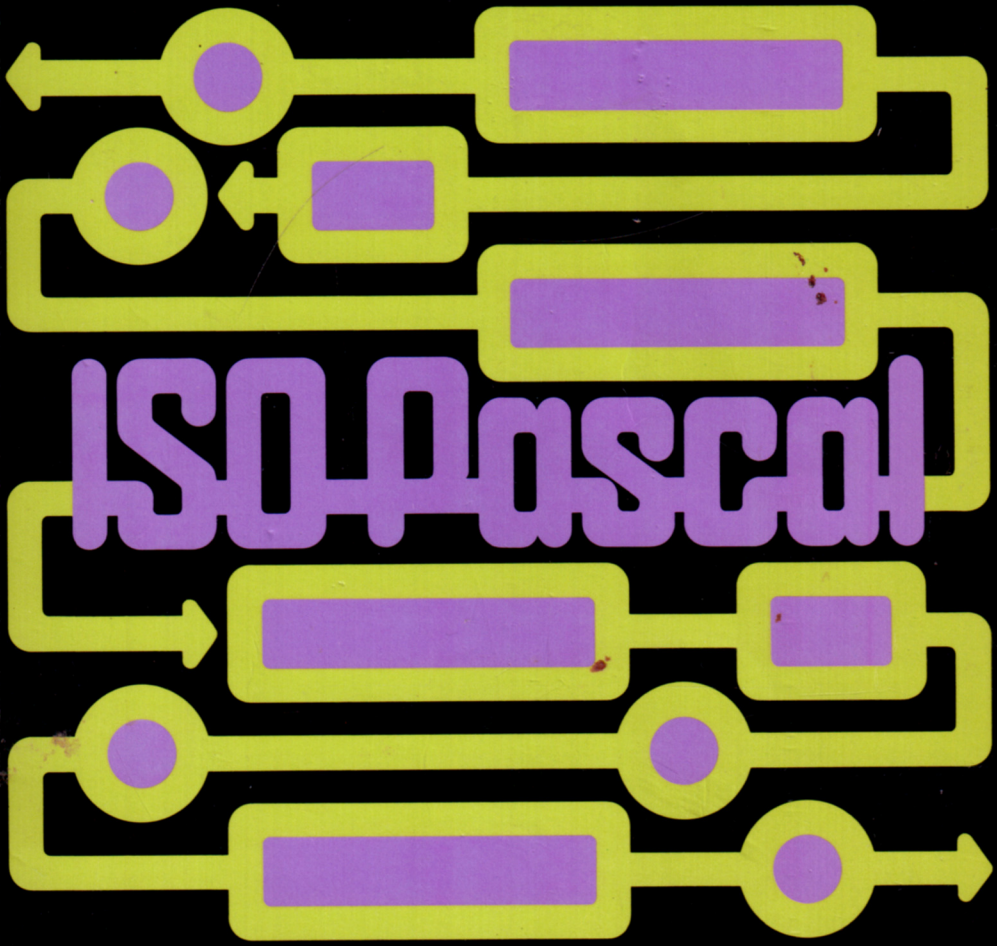
ACORNSOFT

PETE COCKERELL



on the BBC Microcomputer
and Acorn Electron

ISO-Pascal

on the BBC Microcomputer and Acorn Electron

PETE COCKERELL

ACORNSOFT

First published in 1984 by Acornsoft Limited

FIRST EDITION

ISBN 0 907876 97 8

Note: British Broadcasting Corporation has been abbreviated to BBC in this publication.

Acornsoft Limited, Betjeman House, 104 Hills Road,
Cambridge CB2 1LQ, England. Telephone (0223) 316039

Contents

1 Introduction

This manual describes the use of the Acornsoft ISO-Pascal system. It does not teach the language, but explains the steps necessary to create and compile Pascal programs on the BBC Microcomputer and Acorn Electron. For tutorial information, see *Pascal from BASIC*, which is provided as part of the system. The ISO specification for Pascal is included as Appendix A and provides a summary of the Pascal commands.

There are three versions of Pascal that may be used with this manual: BBC Microcomputer ROM Pascal, BBC Microcomputer with 6502 Second Processor disc Pascal and Acorn Electron ROM cartridge Pascal. In most respects the versions behave in exactly the same way, but there are some important 'machine dependent' differences.

1.1 BBC Microcomputer ROM Pascal

This version comes on two ROMs which may be fitted by the user (in accordance with the instructions included in the pack) or by an Acorn-approved dealer. The version of Pascal implemented is ISO-standard level zero with minor restrictions (see Appendices A and B) and Acornsoft extensions.

All of the Pascal commands listed in section 1.7 are implemented, in particular it is possible to use memory files when using the COMPILE command; see chapter 3.

If you are planning to use Pascal on a BBC Microcomputer and don't have previous experience of the machine, you are advised to read the introductory chapters in the *BBC Microcomputer System User Guide*.

1.2 BBC Microcomputer disc Pascal

This version is for use on BBC Microcomputers fitted with a 6502 Second Processor. The version of Pascal implemented is ISO-standard level one with extensions. The compiler and interpreter (which take the place of the two ROMs in the system described above) are disc files present on the disc that comes with the BBC Microcomputer version of Acornsoft Pascal.

To call up the language, the command

*DPASCAL

must be issued (with the Second Processor switched on, of course). All of the commands described in this manual may be used, with the exception of 'COMPILE'. To compile a program, the Pascal program on the system disc called 'DCOMP' must be used. This is described in further detail in section 1.6.

1.3 Acorn Electron ROM cartridge Pascal

This version is very similar in use to the BBC Microcomputer ROM Pascal. The main differences are the lack of a Teletext (MODE 7) display, and a different keyboard layout for the Pascal editor. A pseudo-MODE 7 display is available which saves some memory over MODE 6 on the Electron. The keyboard layout for the editor is described in chapter 2 for both the BBC Microcomputer and Acorn Electron versions and a function key card is provided in the pack.

Users who have not had experience with the Acorn Electron are advised to read the introductory material in the machine's User Guide before proceeding with Pascal.

1.4 Components of the system

The items which comprise the Pascal system are:

1. The Pascal language and the text editor contained in two 16K ROMs (BBC Microcomputer version) or a ROM cartridge pack (Acorn Electron version).

2. A 'ROM installation leaflet' (BBC Microcomputer version only).

3. A function key card relating to the text editor.

4. A floppy disc containing the error message file, some Pascal demonstration programs and the extended compiler/interpreter for 6502 Second Processors (BBC Microcomputer version only).

5. A reference card.

6. The book *Pascal from BASIC* by P J Brown.

7. This manual, *ISO-Pascal on the BBC Microcomputer and Acorn Electron*.

1.5 Installing the Pascal system

Installing Pascal in a BBC Microcomputer involves inserting the ROMs into two of the sideways ROM sockets underneath the keyboard. This should only be attempted if you are confident about delving into the machine's innards. If not, your Acorn dealer will be happy to fit the ROMs for a nominal charge.

Instructions for fitting the ROMs are given in the leaflet supplied with the BBC Microcomputer version of the Pascal system.

To use Pascal on an Acorn Electron, first make sure the machine is turned off – then you simply have to insert the Pascal ROM cartridge into a slot on the Plus 1. When the machine is switched on, Pascal will announce itself with the '%' prompt. If you are planning to use Pascal as the main language on your Acorn Electron, it is advisable to insert it into the rear socket, where it may be left permanently. If you then want to use a games cartridge, for example, this may be placed in the other socket, where it will take precedence.

1.6 Entering the Pascal system

The two ROMs in the Pascal system are language ROMs, just like BASIC. This means that Pascal can be selected either by issuing the appropriate operating system command, or by placing one of the ROMs in the rightmost socket in the computer. The latter ensures that Pascal is the language selected by the operating system when the computer is turned on, or when a hard reset (CTRL BREAK) is performed.

To enter Pascal from another language, it must be possible to issue an operating system command. Most languages (eg BASIC, VIEW and BCPL) enable users to do this simply by prefixing the command with an asterisk (*). So to call Pascal from one of the languages mentioned above, the line:

*PASCAL

should be typed when the language prompt has been given. Note that it is normally possible to abbreviate the word 'PASCAL' to 'P.' and to use either upper or lower case letters.

For languages such as FORTH and LISP, it is slightly more difficult to issue an operating system command; see the appropriate user guide for details.

The disc that comes with the BBC Microcomputer version of the Pascal system contains a version of the compiler and interpreter for use on 6502 Second Processors. The interpreter is a 'high' version which loads at &B800. The compiler is an extended version of the ROM compiler that conforms to ISO level one class A requirements. To enter the disc Pascal system, the command:

*DPASCAL

should be issued. Note that this may not be abbreviated unless the file is renamed on the disc. The only difference between using the ROM and disc systems is that the disc interpreter does not have the COMPILE command (see below) built in. The disc compiler is called DCOMP. Thus to compile a program using the disc compiler, a command of the form

must be issued (see chapter 3 for further details). This means that when using the disc Pascal system, it is only possible to compile to and from disc (ie memory files may not be used).

Most examples in this book assume that you are using the ROM-based system. Chapter 3 explains how to use the disc-based compiler, and the differences between the two systems.

1.7 The command mode

Once the Pascal system has been activated, it prints a prompt and waits for the user to type a command. The Pascal prompt is '%' to distinguish it from BASIC and the other languages. At this stage the user is in command mode; he or she can type Pascal commands and operating system commands.

Because Pascal is a compiled language, it is not possible to type sections of Pascal and have them executed in command mode. In this it differs from BASIC, which is 'interactive' and executes immediate statements as soon as they are entered. There is a fairly small set of commands which may be used in Pascal. These are listed below with their minimum abbreviations. They may be in upper or lower case.

CLOSE (C.)

This command closes all open files on the current filing system. It is useful in situations where BREAK is pressed in the middle of a compilation, resulting in the source and/or object file being left open. The file(s) will be inaccessible until closed.

COMPILE (CO.)

This compiles a source program and generates an object program which can subsequently be executed. The source and object files can be in memory or on disc (if available). If tape is being used the 'memory file' facility provided by Pascal should be used. COMPILE is described in detail in chapter 3.

EDIT (E.)

This command activates the screen editor. This editor may be used to edit any text file, though its main use will obviously be in the creation and amendment of Pascal source programs. If a filename is specified after EDIT, this will be loaded before the editor is called, otherwise the current memory text file will be used. The editor is discussed in detail in chapter 2.

GO (G.)

This command executes the code file currently in memory, if there is one. It is discussed in greater detail in chapters 3 and 6.

LOAD (L.)

This command loads an object or code file into memory. It can then be executed using the GO command. See chapter 3.

MODE (M.)

MODE is used to change the BBC Microcomputer or Acorn Electron display mode. Its main use is to enable larger files to be compiled, edited or executed by changing to a low memory-usage mode. MODE 7 is the least greedy of the displays, and should be used when compiling large programs. MODE 3 is the most convenient mode in which to edit text, but reduces the amount of free memory to about 9000 bytes. Note that when a 6502 Second Processor is used, the display mode does not affect the amount of memory available to Pascal.

On the Acorn Electron, typing MODE 7 will cause MODE 6 to be used as usual. In addition, however, a small text window of 9 lines by 40 columns will be defined at the bottom of the screen. The top 16 lines are used as workspace when compilation is taking place. This will cause 'garbage' to be written on to the screen, but has the advantage of enabling larger programs to be compiled.

RUN (R.)

The RUN command is equivalent to LOAD followed by GO. The command is described with GO in chapter 6.

SAVE (S.)

This saves any object code in memory to the file specified after the command. See chapter 3.

TRACE (T.)

This command enables or disables tracing of programs that were compiled with the debug option turned on (see chapter 4). TRACE 2 causes procedure names and line numbers to be printed, TRACE 1 causes just procedure names to be printed, and TRACE 0 disables tracing entirely.

In addition to the above commands, the user may give operating system commands prefixed by an asterisk, eg '*CAT'. Also, special codes may be sent to the VDU using the CTRL key as from BASIC's command mode. This is useful for turning the printer on and off (CTRL B, CTRL C), turning paged

mode on and off (CTRL N, CTRL O), and clearing the screen (CTRL L). The display mode should not be changed using control codes; the MODE command should be used.

Any other (ie unrecognised) command, say 'FTUMPCH', will be interpreted as the name of a code file which Pascal will attempt to RUN. If, however, the cassette filing system is selected, such a command will cause a 'Bad command' error to be generated. This behaviour is similar to the operating system's response to unrecognised * commands.

2 The Pascal editor

As mentioned in the last chapter, Pascal programs must be prepared using a text editor so that they can be compiled before execution. This chapter describes the use of the EDIT command which is accessible from the Pascal command mode.

Before using the editor, the function key card provided with the Pascal system should be placed under the clear plastic strip above the keyboard (for the BBC Microcomputer) or on top of the 'Acorn Electron' label (for the Acorn Electron).

2.1 Entering EDIT

There are two ways of entering the editor – with or without a filename. If the command:

```
EDIT
```

is given the text file currently stored in memory will be edited. If there is no text file present, a blank screen will be displayed to which the user may add text.

The alternative way of calling the editor is to follow the command with a filename, as in:

```
EDIT ex1
```

where 'ex1' could be any appropriate filename. This will load the text in, then call the editor.

As the minimum abbreviation for 'EDIT' is 'E.', the second example above could have been typed as 'E.ex1'.

For the purposes of the following description, it will be assumed that just 'EDIT' was typed with no text file in the memory. The editor may be used in any of the 40 and 80 column display modes, though operation is very slightly different in MODE 7. The difference is in the way control codes are displayed and is described in section 2.10.

When the EDIT command is given, the screen clears apart from the very top and bottom lines. At the top line is an inverse video '*' which marks the end of text. (This is just a white square in MODE 7.) The fact that it is at the start of the

page implies that currently there is no text being edited. Just below the marker is the flashing cursor. This always marks where the characters will be put into the text.

Note for Electron users:

If MODE 7 has been selected from command mode (see chapter 1), upon entering the editor MODE 6 will be used, with a full 40 by 25 screen. The system will remain in MODE 6 when the editor is exited.

2.2 The status line

The bottom line of the screen contains the 'status line'. This line gives the user information about various modes of operation. The status line looks like this when the editor is first entered:

```
#Insert  0 mark(s)
```

This indicates that the editor is in insert mode, cursor editing is inactive and there are no marks set (these terms are explained below). When the editor is in overtype mode with cursor editing active and two marks set, the command line would look like this:

```
#Over * 2 mark(s)
```

the '*' indicating cursor edit mode.

The status line also gives the number of matches found after a replace command.

2.3 Using insert mode

When the editor is first entered it is always in insert mode where text is added simply by typing it in. The end of text marker or anything else to the right of the cursor will be moved along to make room for the new text. There is another mode called overtype and this is discussed in section 2.9. To illustrate some of the features of the editor, the entry of the simple Pascal program listed below will be described in detail:

```
program charSet(output);
{ Prints the ASCII character set }
var
   ch : char;

begin
   for ch:=chr(32) to chr(126) do
      write(ch);
   writeln
end.
```

First type the top line exactly as shown above. If you make a mistake, press the DELETE key to erase the incorrect characters. Notice that pressing DELETE removes the character before the flashing cursor. Pressing COPY deletes the character at the cursor position. When you reach the end of the line, press RETURN. This will move the cursor to the start of the next line. Type the next two lines in the same way. Note that in MODE 7 on the BBC Microcomputer the character '{' appears as '¼' and '}' is displayed as '¾'.

The fourth line of text is indented. To achieve this, simply press the Space Bar the correct number of times. It is also possible to use the right-arrow cursor key, though pressing the Space Bar is probably more convenient.

The fifth line is blank. Producing blank lines simply involves pressing RETURN when the cursor is at the start of the line. In this example, it means pressing RETURN twice after the ';' of line four instead of just once.

Line six can be typed as normal, as can line seven, remembering to press the Space Bar to achieve the indentation. Line eight is indented further. Instead of typing six spaces, press the TAB key (FUNC A on the Acorn Electron). The cursor will move in three spaces automatically. This is a property of TAB (FUNC A) in the editor: it moves the cursor to the same column as the next non-space character in the line above the current one. If the cursor is already below a non-space, it moves forward until it is below a space, then it skips the spaces above it. For example, if the cursor was below the first character of this line:

```
1234    5678 90AB   CDEF
```

successive TABs would move it to below the '5', below the '9', and finally below the 'C'.

Using TAB isn't that useful in this example as you still have to type in three spaces yourself. However, in most Pascal (and assembly language) programs, there are sequences where many lines have the same indentation. Using TAB (FUNC A) to achieve this can save quite a lot of typing.

Note that TAB (FUNC A) does not actually insert any spaces: it simply moves the cursor so that it is below the first non-space character above the cursor. Spaces are not inserted until a key is pressed.

The last two lines can be typed in as normal.

2.3.1 Executing the program

At this stage, assuming no errors have been made, the file can be compiled. If you are keen to try this immediately, before learning more about the editor, type the commands listed below and the screen will appear as shown, otherwise go straight to section 2.4.

SHIFT f4 (FUNC 5 on the Electron)

```
%COMPILE <RETURN>
ISO-Pascal compiler V. 1.00

   1  0 - program charSet(output);
   2  0 - { Prints the ASCII character set }
   3  0 - var
   4  0 -     ch : char;
   5  0 -
   6  0 - begin
   7  0 -     for ch:=chr(32) to chr(126) do
   8  0 -         write(ch);
   9  0 -     writeln
  10  0 - end.

0 Compilation error(s) Code size = 72 bytes
%GO <RETURN>
 !'#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_£abcdefghijklmnopqrstuvwxyz{¦}~
%EDIT <RETURN>
```

The SHIFT f4 (FUNC 5) leaves the editor, and goes back into Pascal command mode. The source file is left intact. The command COMPILE is the simplest form of compilation: it means compile the source file in memory and leave the object code in memory. The meaning of the compilation listing is discussed in the next chapter.

As there were no errors, the object program (in memory) may be executed using the GO command. This produces the output shown.

Having executed the program, you may modify it using the editor simply by typing EDIT (or E.). The rest of this chapter discusses more of the features of the editor.

2.4 Using the cursor keys

BBC editor	Function	Electron editor
Up arrow	Move up a line	Up arrow
Down arrow	Move down a line	Down arrow
Left arrow	Move left a character	Left arrow
Right arrow	Move right a character	Right arrow
SHIFT up	Move up a page	FUNC N
SHIFT down	Move down a page	FUNC M
SHIFT left	Move to start of line	FUNC <
SHIFT right	Move to end of line	FUNC >
CTRL up	Move to top of text	FUNC Z
CTRL down	Move to end of text	FUNC X

Now that we have some text to manipulate, it is possible to start using the various facilities of EDIT. First of all, it is desirable to be able to move around the screen, changing various parts of the text. To do this, the four cursor (arrow) keys are used. If you press one of these keys, the cursor moves one space in the direction indicated, that is one character to the left or right for the left- and right-arrow keys respectively, and one line up or down for the up- and down-arrow keys.

It is not possible to move past the top of text or below the end of text marker. As an example, change the 'write' on line eight above to 'writeln'. To do this, use the arrow keys to position the cursor on the '(' after write. Then simply type the characters 'ln'. The text after the cursor will be shifted to the right to make room for the new text. This happens when the editor is in 'insert mode'. There is another mode called 'overtype mode' which is described in section 2.9.

Deleting small sections of text is equally easy. Move the cursor to the character after the item to be deleted, then press DELETE the appropriate number of times. Again the text will move, this time to the left. Notice that if you move the cursor to the beginning of a line and press DELETE, the current line will join up with the one above it. This happens because you have deleted the (normally invisible) carriage return character which separates the two lines.

2.4.1 Large cursor movements on the BBC Microcomputer

The cursor keys can be used in combination with SHIFT or CTRL to make

larger movements around the screen. If SHIFT is pressed with the up and down keys, the text will scroll up and down by a 'page'. The length of a page is dependent on the screen mode in use. This is useful for moving rapidly over a large region of text.

The CTRL key can be used with cursor up and cursor down to reach the top and bottom of the text respectively. This is particularly useful in Pascal as you might want to check the spelling of a variable which is declared at the top of the program, when the text you are adding is at the bottom.

Pressing SHIFT or CTRL with left or right arrows moves the cursor to the start and end of the current line respectively.

2.4.2 Large cursor movements on the Acorn Electron

To move up and down a 'page' (one screenful of text), the FUNC key must be used. FUNC N moves the cursor up one page, and FUNC M moves it down one page.

Similarly, it is possible to move to the top of the text by pressing FUNC Z, and to the bottom of the text by pressing FUNC X.

To move to the start of the current line on the Electron version of the Pascal editor, press FUNC <. To move to the end of the line, use the combination FUNC >.

2.5 The cursor edit mode

The use of cursor editing with the COPY key in the editor is slightly different from its use in command mode. Normally, to copy some text when typing in a Pascal command, the arrow keys are used to move the copy cursor to the required part of the screen, then COPY is pressed to do the copying. Copy mode ends when RETURN is pressed.

In the editor, the only difference is that instead of one of the cursor keys initiating copy mode, SHIFT COPY (FUNC : on the Acorn Electron) must be pressed. Then the cursor keys can be used to move to the text to be copied, and COPY pressed the appropriate number of times. Copy mode is terminated when ESCAPE (rather than RETURN) is pressed.

This technique is very versatile as it means that the cursor keys may be used to move around the screen when inserting or deleting text, and also to move the copy cursor around as usual. The obvious application of copying is for duplicating identical or nearly identical lines.

When in cursor edit mode, the status line reflects this by having an asterisk in it.

A useful feature of cursor edit mode is that the strings associated with the function keys become available. If the command '*key0procedure ' had been issued at some time, pressing SHIFT-COPY f0 (or FUNC : FUNC 0) would cause the string to be entered. ESCAPE has to be pressed to exit from cursor edit mode.

2.6 Extra functions on the BBC Microcomputer editor

It can be seen that quite a lot of editing may be carried out simply by using the keys discussed so far. However, EDIT provides functions which make the manipulation of program text even easier. These functions are accessed by using the red function keys at the top of the keyboard on the BBC Microcomputer.

As some of the red keys provide more than one function, it is sometimes necessary to press SHIFT at the same time as the key itself. For example, pressing function key 0 searches for a given line number, whereas SHIFT function key 0 switches the display of carriage returns on and off. All these functions are shown on the function key card provided. Below, 'function key n' will be abbreviated to 'fn', where 'n' is 0 to 9, as shown on the keys themselves.

2.7 Extra functions on the Acorn Electron editor

All of the functions available on the BBC Microcomputer version of the editor are accessible on the Acorn Electron version. As there are no red function keys on the latter machine, the editing functions such as 'Goto a line' are activated using combinations of FUNC and other keys. There is a simple relationship between the layout of the BBC Microcomputer functions and the Acorn Electron ones. FUNC Q has the same function as f0, FUNC W as f1, across to FUNC P which has the same function as f9. Similarly, FUNC 1 corresponds to SHIFT f0, across to FUNC 0 which is the same as SHIFT f9.

In the descriptions below, both the BBC Microcomputer and Acorn Electron key combinations are given. In the tables, the BBC Microcomputer version comes first, followed by the description, which is followed by the Acorn Electron version. In the detailed descriptions, the Acorn Electron keys are given in parentheses after the BBC Microcomputer keys.

2.8 Saving, loading and inserting text

It is obviously useful to be able to save and load files from the editor. There are three commands (pressing a function key will be treated as issuing a command in this chapter) for manipulating files from within EDIT. These are:

f2	Load the text in a file	FUNC E
f3	Save text to a file	FUNC R
SHIFT f2	Insert text from a file	FUNC 3

In addition, there are

SHIFT f4	Quit from the editor	FUNC 5
f8	Print the current text	FUNC O
f9	Restore old text	FUNC P
SHIFT f9	Delete the text	FUNC 0

Pressing f3 (FUNC R) produces a prompt like:

```
#Save [prog1]:
```

In response you should give the name of the file into which the text should be saved. If an error occurs, it is reported and you will be prompted to press the Space Bar to continue.

If you press RETURN with no filename, the editor will use the 'current' filename, ie the name last used in a load command. This is given in square brackets after '# Save'. The current filename is updated every time a load command is executed.

It is possible to save only part of the file you are editing. The method is to put a marker at the start of the portion of text to be saved, move the cursor to the end of the section, and then use f3 (FUNC R) as described above. The setting and clearing of markers is described in section 2.11.

Text is loaded using the command f2 (FUNC E). The prompt this time is:

```
#Load []:
```

In this example there is no default filename, and one must be typed before RETURN is pressed. This command wipes out any text already in the machine. Errors are reported in the manner described above. Additionally, if the file is too large to be read into the computer, the error 'File too long' is given.

SHIFT f2 (FUNC 3) inserts a file into a particular place in the text. Its prompt has the form:

```
#Insert [pas_1]:
```

to which you should reply with a filename or just RETURN. The file will be read into the text at the current cursor position. Text before the cursor will be unaffected; characters at and after the cursor will be moved in order to make room for the file, and the cursor will be placed at the start of the text just read in. This command is useful for inserting frequently-used procedures into programs. Note that after an insert operation the current filename will be that of the file inserted.

Pressing SHIFT f4 (FUNC 5) will cause the editor to be exited and Pascal command mode entered. The text will remain intact, so if you forget to save it before leaving the editor, typing the command 'E.' will re-enter the editor with the old text still there. (This assumes you have not issued any 'LOAD' or 'RUN' commands, as these overwrite the text file.)

Pressing f8 (FUNC 0) causes the current text to be printed out. The printer is enabled, then the current text file is listed to the screen (and hence the printer), then the printer is disabled. For disc users this is exactly the same as typing

<CTRL B>
*TYPE source
<CTRL C>

in Pascal command mode. The f8 command is provided mainly for the convenience of cassette users who haven't got the *TYPE facility.

Pressing SHIFT f9 (FUNC 0) causes the current text to be removed. This is fairly drastic, so the command should obviously be used with some caution. If you press SHIFT f9 (FUNC 0) or BREAK by mistake, it is possible to restore the text using f9 (FUNC P) (in the latter case it will be necessary to re-enter the editor first, of course).

2.9 Insert and overtype modes

SHIFT f1 Toggle between insert and over FUNC 2

So far, only insert mode has been discussed. When the editor is first called, text typed at the keyboard is inserted into the document by shifting everything after the cursor to the right. In order to replace a word, it must be deleted and then retyped. It is sometimes more convenient to be able to replace characters simply by overwriting them.

In EDIT, it is possible to switch between insert and overtype mode by pressing SHIFT f1 (FUNC 2). This key acts as a 'toggle' so that if you are in overtype mode pressing it will put you in insert mode, and if you are in insert mode, pressing SHIFT f1 (FUNC 2) will put you in overtype mode.

The bottom (status) line of the display shows the current typing mode as 'Insert' or 'Over'. Try pressing SHIFT f1 (FUNC 2) several times to see the effect on the status line.

To see how overtype mode differs from insert mode, press SHIFT f1 (FUNC 2) until the status line has 'Over' in it and move the cursor to the start of a line already containing some text. Now start to type. The line will be overwritten by what you typed rather than be moved in order to make room for it.

The way in which carriage return characters are treated differs between insert and overtype modes. In insert mode, as we have seen, it is possible to delete a carriage return by moving the cursor to the start of the previous line and pressing DELETE. You may have also discovered that it is possible to split a line into two by moving the cursor to where you want the split to appear and pressing RETURN.

In overtype mode, pressing RETURN will cause the character at the cursor to be overwritten by a carriage return. Pressing DELETE at the first character on a line, or COPY when the cursor is at a carriage return will cause the carriage return to be overwritten by a space. Note that a line may be extended in overtype mode simply by typing past the carriage return. Thus it is not possible to overtype a carriage return.

2.10 Special characters in the text

SHIFT f0 Toggle carriage return display FUNC 1

It is possible to type control characters in the same way as any other character. To distinguish them, they are shown in inverse video in MODEs 0 to 6 and as white 'blobs' in MODE 7.

A special inverse character is RETURN, which is the same as CTRL M. Since a RETURN character appears on the end of every line, showing it as an inverse 'M' would be very distracting. Normally, then, EDIT does not show carriage returns. However, pressing SHIFT f0 (FUNC 1) will make them visible. Pressing it a second time will render the inverse 'M's invisible again, so it acts as a toggle (like SHIFT f1 (FUNC 2) for insert/overtype mode).

By making the RETURNs visible it is possible to see if there are any unnecessary trailing blanks on a line. (These can occur when the cursor is past the end of the normal text and the Space Bar is pressed inadvertently.)

The character DELETE (whose code is 127) is shown as a small white rectangle in all modes. Characters with codes greater than 127 are displayed as normal (so user-defined characters and MODE 7 colour codes have their usual effect). There is, however, no direct way of typing characters with codes between 127 and 255 from the keyboard.

2.11 Dealing with blocks of text

There are three operations which can be performed on a block of text. These are:

SHIFT f8	Delete a block of text	FUNC 9
f7	Copy a block of text	FUNC I
SHIFT f7	Move a block of text	FUNC 8

In addition, two commands are needed to set and reset markers, which are used in conjunction with the above commands. These are:

f6	Set marker	FUNC U
SHIFT f6	Clear marker(s)	FUNC 7

2.11.1 Deleting a block

To delete a block of text, two delimiters are needed, one at the start and one at the end of the block. One of the delimiters is the cursor and the other is a marker. Consider the text:

The quick
brown fox jumps
over the
lazy dog.

Suppose it is required to delete the middle two lines. This is accomplished thus: move the cursor to the 'b' in 'brown'. Press f6 (FUNC U). This sets marker 1 (an inverse space) at the cursor position which will act as the first delimiter. Then move the cursor to the 'l' in lazy and press SHIFT f8 (FUNC 9). This will delete the required two lines.

Notice the exact characters which are deleted: from the first delimiter inclusive to the last delimiter exclusive, so the first delimiter should be placed at the first character in the block and the second delimiter should be just after the last character.

Note also that the marker can equally well be the second delimiter. The two lines could have been deleted by setting the marker at the 'l' in 'lazy' and moving the cursor up to the 'b' in 'brown' before pressing SHIFT f8 (FUNC 9). The result would be the same.

Once a marker has been set, you should not try to insert or delete any characters, otherwise a 'Bad marking' error will occur.

2.11.2 Copying a block

As described earlier, it is possible to copy text using the COPY key. This can be very tedious if there is a lot to copy, so a way is provided of duplicating a whole

block of text. Again there are two delimiters needed to mark the area and a way of marking the destination of the copied block.

Suppose it is required to copy the text

```
lda # ASC">"
jsr oswrch
```

This is done as follows: move the cursor to the 'l' of 'lda' and press f6. This sets marker one. Move the cursor to below the 'j' of 'jsr' and press f6. This sets marker two. Move the cursor to where the block is to be copied to (this must not be between the two markers or a 'Bad marking' error will occur). In this example the cursor should be moved to the line below marker two. Press f7 (FUNC I). This produces a copy of the required lines. Notice again that the block copied lies between delimiter one inclusive and delimiter two exclusive.

2.11.3 Moving a block

Moving is equivalent to copying a block then deleting the original. The block to be moved is delimited by two markers and it is moved to the position of the cursor when SHIFT f7 (FUNC 8) is pressed. Again it is illegal for the cursor to be within the region delimited by the markers when the delete command is given.

2.11.4 Deleting the marker(s)

The three commands described above delete any markers present automatically. However, it is sometimes desirable to delete the markers without having to execute a 'block' command. Pressing SHIFT f6 (FUNC 7) will delete the active marker(s).

It should be noted that there are several operations which can't be performed if there are markers in the text. These are: typing in text, inserting a file, and using the interactive find and replace commands to replace text.

The status line indicates how many markers there are in the text; it ends with the phrase 'd mark(s)' where d is 0, 1 or 2.

2.12 The scroll margins

You may have noticed that when the cursor is moved near to the top or bottom of the screen, the text scrolls and the cursor stays on the same line. This usually happens when the cursor tries to move above the fourth line from the top or below the fourth line from the bottom. These two lines mark the so-called 'scroll margins'.

The bottom scroll line is used by various search commands. For example, the f0 (FUNC Q) and f4 (FUNC T) search commands described later both cause their

targets to be displayed on the bottom scroll line. Also, when CTRL down arrow (FUNC X) is used to move to the end of text, the marker is displayed on the bottom scroll line.

2.13 Finding and replacing text

We now come to one of the most useful features of EDIT. When editing a large file, it is often desirable to find the occurrence of a particular word or phrase, perhaps with a view to changing it to something else. Scanning through by eye is tedious and prone to error. Another requirement is to be able to jump to a given line in the text. This is necessary as compilers and assemblers usually give the line number at which an error occurred. Being able to find this incorrect line quickly greatly aids debugging.

EDIT has the ability to find a given line, find and selectively replace one string with another, and count all occurrences of a string, optionally replacing it with something else. The relevant commands are:

f0	Find a line number	FUNC Q
f4	Find and replace a string	FUNC T
f5	Globally count/replace a string	FUNC Y

2.13.1 Finding a given line

Pressing f0 (FUNC Q) produces the prompt:

```
#Line:
```

to which you should type the number of the line to be found. The top line is number one. If the line specified is greater than the number of lines in the document, a 'Not found' error is generated. Otherwise, the screen is updated so that the line in question is placed on the bottom scroll margin.

2.13.2 Finding and selectively replacing a string

A more general way of searching the file is for a particular string. EDIT lets you search for simple strings such as 'begin', but more powerfully for such things as 'all identifiers beginning with A'. The command f4 (FUNC T) finds strings and, if required, changes them to something else. In this context the string being sought is called the 'pattern' and the string with which it will be replaced is called the 'replacement'.

In response to f4 (FUNC T) EDIT will produce the prompt:

```
#Find and replace:
```

You should type one of two things: a pattern, followed by RETURN, or a pattern, then a '/' as a separator, then a replacement and finally RETURN. Examples are:

begin<RETURN>	(Find occurrences of 'begin')
for/FOR<RETURN>	(Replace (some of) 'for' by 'FOR')
=/:=<RETURN>	(Replace (some of) '=' by ':=')
then /<RETURN>	(Delete (some of) 'then ')

In the last example, the replacement is a null string, which leads to the pattern being deleted.

The search for the pattern begins at the cursor position, so you must move the cursor to the top of the file (CTRL up or FUNC C) if you want to find all occurrences. When the pattern is located, the editor updates the screen so that the pattern is on the bottom scroll line and prompts with:

```
#R(eplace), C(ontinue) or ESCAPE
```

If you press 'R' and specified a replacement then the change is made and the next occurrence sought. If you press 'R' but didn't specify a replacement you will be prompted with

```
#Replace by:
```

so that you can give one, then the change will be made and the next occurrence sought. If you press 'C', this occurrence is skipped and the next one sought. If you press ESCAPE, the search ends. After the last occurrence has been found (or if the search fails), the message '`Not found`' is given.

2.13.3 Globally counting and replacing strings

The command f5 (FUNC Y) acts in a similar way to the last one but assumes that if you specify a replacement string, all occurrences of the pattern should be replaced. If no replacement is given, then the number of times the pattern occurs in the file is counted. The prompt for the pattern and (optional) replacement is:

```
#Global replace:
```

Typical replies are:

FRED/JOHN<RETURN>	(Replace all of 'FRED' with 'JOHN')
HELLO/<RETURN>	(Delete all of 'HELLO')
for<RETURN>	(Count all of 'for')

Notice that the only difference between counting occurrences of the pattern and deleting them is whether a '/' appears at the end of the line. The f5 (FUNC Y) command does the search and replace automatically without any prompts. As this is a 'global' search and replace it starts from the top of the file, independent of the cursor position (but see below). After the search (and replace), the number of times the pattern was found is appended to the status line in the form:

```
1234 found
```

It is possible to limit the scope of the global search and replace by setting a marker before issuing the command. If this is done, only text between the marker and the cursor will be affected. It can be useful when, for example, only the occurrences of an identifier in a particular procedure need to be altered.

2.13.4 Patterns

The patterns used by the search commands may be regarded as expressions. In fact the formal name for patterns is 'regular expression'. They may be thought of as having constant parts (literal text) and variable parts (wildcards, ranges, choices, repeats and inversions). This section describes the different parts in detail.

In the examples given so far, the patterns and their replacements have been simple strings. However, by using special symbols in a pattern, it is possible to specify the variable parts mentioned in the list above. The special characters available in patterns are:

.	matches any character
@	matches any alphanumeric (0–9, A–Z, a–z and _)
#	matches any digit (0–9)
[xyz]	matches any of 'x', 'y' and 'z'
a–z	matches any character between 'a' and 'z'
$	matches the carriage return character
¦c	matches CTRL c
¦?	matches the DELETE character (ASCII 127)
¦!c	matches character code c+128
~c	matches anything but c (which can be a wildcard)
\c	matches c (with no special meaning attached to c)
*c	matches zero or more cs (c can be a wildcard)

A '.' in a pattern matches any single character in the range ASCII 0 to ASCII 255. All the wildcards may be duplicated, so '..' matches any two characters, and so on.

Example. Count the number of character constants in a Pascal program:

```
#Global replace: '.'<RETURN>
```

'@' is slightly more restrictive and matches those characters which are allowed in identifiers (digits, upper and lower case letters and _). '#' matches any of the ten characters in the range '0' to '9', ie the digits.

Example. Find the next three-digit numeric constant:

```
#Find and replace: ###<RETURN>
```

If neither of '@' and '#' provides a suitable range of characters to match, it is possible to define your own range using '–'. Thus 'A–F' matches any valid hexadecimal letter. Another way is to put several choices inside square brackets. Only one of the characters in the brackets will be matched.

Example. Replace all white space with newlines. 'White space' is taken to be the TAB character (CTRL I, written as ↑I in patterns), the space character, or the newline character (written as $ or ↑M in patterns):

```
#Global replace: [↑I$ ]/$<RETURN>
```

It is possible to combine ranges and choices. For example, to match any hexadecimal character, the choice '[0–9A–F]' would be used. Since existing wildcards may be put in a choice, this could also be expressed as '[#A–F]'. It is also possible to mix single characters and ranges so a pattern to match characters which may occur at the start of a Pascal variable name is '[a–zA–Z_]'. This can be read as: match any character in the range 'a' to 'z', 'A' to 'Z' or the character '_'.

'$' is a convenient way of putting the carriage return character in a search string (the RETURN key can't be used for obvious reasons).

The vertical bar '↑' has the same meaning as within *KEY and filename strings, ie 'make the next character a control character'. Thus '↑@' means ASCII 0, '↑A' means ASCII 1 and so on. '↑M' is the same as '$' and '↑[' is the ESCAPE character. Also, '↑?' matches the DELETE character and '↑!c' matches the character with the ASCII code of 'c' plus 128. For example, '↑!↑@' matches chr(128) and '↑!↑?' matches chr(255).

The action of '~' is to match anything but the subpattern that follows. Thus '~A' matches anything but 'A', '~#' matches any non-digit and '~A–Z' matches anything that isn't an upper case letter.

The backslash character is needed to remove any special meaning from the symbol which follows it. Thus '\$' stands for '$', not carriage return, '\¦' prevents the character after the bar from being interpreted as a control character, '\.' means '.' not 'any character', and so on.

Putting a '*' before a pattern means 'match zero or more of' that pattern. So '*.' matches any sequence of characters, '* ' matches zero or more spaces, and '*A–Z' matches zero or more upper case letters.

Example. Delete the trailing spaces from all lines in the text:

```
#Global replace: * $/$<RETURN>
```

This can be read as 'replace zero or more spaces followed by a carriage return with a carriage return'. Preceding a character with an asterisk is called 'forming a closure' over that character, but is usually read, as indicated above, as 'zero or more of'.

A closure pattern will always match the shortest string possible, so if it can it matches nothing at all. For example, the pattern 'proc*.' would match exactly the same strings as 'proc' because the '*.' on the end will do as little work as possible, ie match the null string. To force '*' patterns to do more work, the pattern must end with something 'solid', like the trailing spaces example above.

Example. Find definitions of procedures with parameters:

```
#Find and replace: procedure*[@ ](<RETURN>
```

This says 'find all occurrences of "procedure" followed by a sequence of alphanumerics and spaces followed by a "(" '.

It can be seen that the closure (also called 'multiple match') facility is very powerful, but must be used with care. In order to use '*' to match one or more of a character (as opposed to zero or more) the format:

```
#Find and replace: c*c<RETURN>
```

should be used.

Example. Find all identifiers beginning with A

```
#Find and replace: A*@~@<RETURN>
```

The 'A' matches the first character; the '*@' matches the rest of the identifier up to the non-alphanumeric matched by '~@'.

Example. Find all integer constants

```
#Find and replace: #*#~#<RETURN>
```

The first '#' matches the first digit; the '*#' matches the zero or more following digits. The '~#' ensures that all of the number is matched.

Example. Find all the '\'s in a file

```
#Find and replace: \\<RETURN>
```

Two backslashes are needed as '\' itself is a special character and, therefore, needs a preceding '\' to 'quote' it.

Example. Find all blank lines

```
#Find and replace: $* $<RETURN>
```

A blank line is simply where a carriage return is followed by another one with only zero or more spaces intervening. The first '$' matches a carriage return; the '* ' matches zero or more spaces; the second '$' matches the carriage return of the blank line. Note that this pattern will only find the first of one or more blank lines.

Example. Match any control character

```
#Find and replace: ¦@-¦_<RETURN>
```

'¦@' is CTRL @, the lowest valued control character (ASCII 0) and '¦_' is CTRL _, the highest control character (ASCII 31).

Example. Match a hexadecimal constant

```
#Find and replace: \&[#A-F]*[#A-F]~[#A-F]<RETURN>
```

This is simply '&' followed by one or more hex characters. '&' is escaped as it has a special meaning in replacements which is described below.

Two useful global search without replace commands are:

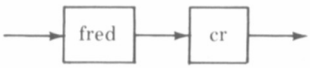
```
#Global replace: .<RETURN>
#Global replace: $<RETURN>
```

These display the number of characters and lines in the file respectively on the status line.

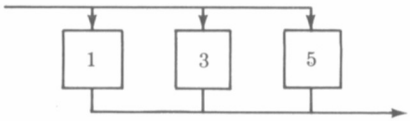
2.13.5 Patterns as syntax diagrams

In a way, patterns specify the 'syntax' of the strings that they will match. Like the syntax of Pascal, patterns may be described in the form of syntax diagrams. If you understand these, thinking of patterns in the same form might make it easier to see what will be matched, and conversely how to specify the pattern to match a given type of string. The syntax of patterns is very straightforward compared to Pascal, closures being the only things to generate any complexity.

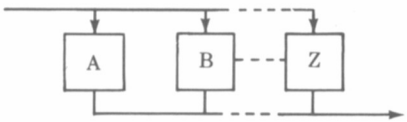
As a simple example, imagine the pattern 'fred$' which matched the word 'fred' at the end of a line. This would be:



Slightly more complex is '[135]', which is a choice. This translates into:



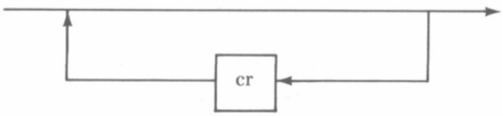
This shows that one, and only one, of the possible characters may be matched. Ranges are similar, but usually have more elements, as in 'A–Z'. This may be depicted as:



The wildcard '#' is exactly equivalent to the range '0–9' and '.' is also equivalent to a range, this time with 256 choices from chr(0) to chr(255).

The closure character produces the most interesting syntax diagram.

Consider the pattern '*$':



From this it is clear that a closure may match no characters at all, by passing straight through the top line, or may loop round through the 'cr' (standing for 'carriage return') an arbitrary number of times. The important thing to remember when viewing the diagram with respect to EDIT's pattern matching is that EDIT will always try to find the shortest path through the diagram.

It may be an interesting exercise to compare the syntax diagram for a Pascal identifier (as given in *Pascal from BASIC*) and the equivalent pattern in EDIT: '[A–Za–z]*@'.

2.13.6 Replacements

Although it is obviously very useful to be able to find patterns of the type described above, it is even more useful to be able to replace them with something else. A replacement can be either a literal string such as 'fred' or may contain various special characters. These are:

$	stands for a carriage return
¦c	means CTRL c
¦?	means DELETE
¦!c	means character code c+128
\c	means c (with no special meaning)
&	whatever was matched by the pattern
%n	field number n

In the list above, 'c' stands for a character and 'n' stands for a digit between 0 and 9. The first four items have already been encountered in patterns.

The ampersand '&' is a character which only has a special meaning in the replacement string (although, as we saw earlier, it still needs to be escaped in patterns). It means 'whatever the pattern matched'. Clearly if the pattern was just a literal such as 'until', then that is what '&' will stand for. However, when wildcards and repeats ('*') are used in the pattern, it is not possible to know exactly what was matched. Suppose it was required to duplicate all digits so that for example '1' becomes '11' and '123' becomes '112233'. This could be achieved with the following global replace:

```
#Global replace: #/&&<RETURN>
```

Whatever is matched by the pattern will become the ampersand in the replacement. Notice that although global replacing was used in the example above, the same replacement string could have been used in a selective replace just as legally.

The final special replacement character acts as a more restricted version of '&'. The percent sign '%' is followed by a digit n between 0 and 9. This combination

stands for the nth field of the pattern. A 'field' is defined as any non-constant item in a pattern, ie a wildcard character, a multiple match (a symbol preceded by '*'), an inverted match (a character preceded by '~'), a range ('a–z') or a choice ('[13579]'). The fields are numbered from zero on the left. Some examples should make this clearer:

```
#Find and replace: A*@+.<RETURN>
```

The characters matched by the '*@' are field 0, and the character matched by the '.' is field 1.

```
#Find and replace: ##~@* <RETURN>
```

Here there are four fields: '#', '#', ' ~@' and '* ' respectively.

```
#Find and replace: *~##*#/
```

This matches any number of non-digits followed by one or more digits. The first field is '*~#'; the second field is '#' and the third one '*#'.

Here are some examples of using the fields in replacement strings:

'Reverse the order of alternate characters'

```
#Global replace: ../%1%0<RETURN>
```

'Delete the % sign from integer variables in BASIC'

```
#Global replace: [a-zA-Z_]*@\%/%0%1<RETURN>
```

(Note the quoted '%' sign in the pattern is for the character at the end of BASIC integer variables.)

'Put a $ after all variables beginning with S__ and delete the S__'

```
#Global replace: S_*@~@/%0\$%1<RETURN>
```

(Again, note the quote sign '\' before the '$' so that it is not treated as a special character by EDIT.)

Notice that when replacing (rather than simply finding) patterns, you have to be very precise about what marks the end of a string. For example, to insert '__1' at the end of all identifiers, it is insufficient to use:

```
#Global replace: @*@/&_1<RETURN>
```

This will in fact cause '_1' to be placed after the first letter of identifiers as the '*@' matches the bare minimum, ie nothing at all. It is necessary to explicitly find the last character in the identifier and use:

```
#Global replace: @*@~@/%0%1_1%2<RETURN>
```

This time the '_1' is placed between the '*@', ie the tail of the identifier, and the '~@', ie the non-identifier character which marks the end.

2.14 Using command 'macros'

f1 Enter operating system command FUNC W

As mentioned earlier, it is possible to generate strings from function keys in cursor editing mode. Since editor commands are really just characters above 127, if these are put into function key definitions it becomes possible to issue several commands at a single keystroke.

Most usefully, this facility may be used to execute several global replaces in quick succession. The following example is for the BBC Microcomputer version, though the same method may be used on the Acorn Electron. The code of the global replace command is 133. To obtain this, the function key string '¦!¦E' is used. Thus to make the string produced by f0 replace all tabs with spaces, the following sequence of commands must be used.

First, press f1 (FUNC W) to enter an operating system command. Then type the following:

KEY0¦!¦E¦¦I/ ¦M<RETURN>
<RETURN>

The first line programs function key zero so that it contains the characters necessary to perform the global replace: '¦!¦E' issues the command (as if you'd typed f5); '¦¦I/' is the pattern and replacement part (there are two '¦'s so the character isn't interpreted by the MOS), and the ¦M is the <RETURN> that would be typed after a global replace command. The <RETURN> on its own gets back to the editor.

Press SHIFT COPY followed by f0. The string defined above will be produced. Obviously it is possible to build up quite useful command strings using the function keys. Here is a list of the strings required to generate various commands:

Command	Alone	+ SHIFT	+ CTRL
f0	¦!¦@	¦!¦P	¦!<SPACE>
f1	¦!¦A	¦!¦Q	¦! !
f2	¦!¦B	¦!¦R	¦!"
f3	¦!¦C	¦!¦S	¦!#
f4	¦!¦D	¦!¦T	¦!$
f5	¦!¦E	¦!¦U	¦!%
f6	¦!¦F	¦!¦V	¦!&
f7	¦!¦G	¦!¦W	¦!'
f8	¦!¦H	¦!¦X	¦! (
f9	¦!¦I	¦!¦Y	¦!)
TAB	¦!¦J	¦!¦Z	¦!*
DELETE	¦?	¦?	¦?
COPY	¦!¦K	¦!¦[	¦!+
left	¦!¦L	¦!¦\	¦!,
right	¦!¦M	¦!¦]	¦!-
down	¦!¦N	¦!¦^	¦!.
up	¦!¦O	¦!¦_	¦! /

Notice that it is possible to generate cursor moves as within the editor the cursor keys are treated as extra function keys. As another example, to set up a key to insert a file called 'decl' the following would be used:

<f1>
KEY0¦!¦Rdecl¦M<RETURN>
<RETURN>

3 Compilation

This chapter describes how Pascal source files are compiled (using the COMPILE and DCOMP commands).

3.1 Memory files

An important part of compilation, particularly when using cassette, is the availability of memory files. These were mentioned briefly in chapter 1. Below is a detailed explanation of which commands use the memory file facility.

At any one time there can be a memory text file and a memory code file present. The text file can be created and manipulated by the editor, or can be loaded from the filing system. The code file can be created by the compiler or loaded from the filing system. The current text file resides at the operating system high water mark (OSHWM or PAGE in BASIC) and the code file will be just above it or at OSHWM depending on how it was created.

3.1.1 The memory text file

The commands which can affect or use the memory text file are: COMPILE, EDIT, RUN and LOAD. This is how the various commands use the file:

COMPILE will use the current text file as its source if no filing system source file was specified. Using memory instead of a normal file speeds up the compilation, especially when compared to the speed of tape. If compiling from a file into memory, the code file will be placed at OSHWM, destroying any text there. If the compilation is from memory to memory then the code file will be placed just after the text, which will remain intact.

EDIT uses the current text file. If a filename is given after the EDIT command, this will be loaded before EDIT is entered, overwriting any text already present. As mentioned above, text files are always loaded at OSHWM.

RUN performs a LOAD so the comments for that command apply here.

LOAD will load a specified object (code) file at OSHWM. As this is where memory text files reside, any current file will be overwritten.

As noted in the last chapter, the memory text file may be loaded and saved from within the editor.

3.1.2 The memory code file

The commands which can use or affect the memory code file are: COMPILE,

EDIT, RUN, LOAD, GO and SAVE. The effects are:

COMPILE can be made to store its object (code) file in memory if a filing system destination isn't given. The code is put either at OSHWM if the source file is on the current filing system, or after the source file if compiling from memory.

EDIT with a filename first loads a file, overwriting the current text and code memory files.

GO executes the current code file, whether it is at OSHWM (eg after a LOAD) or just after the text file (after a COMPILE to and from memory).

LOAD loads in a specified code file at OSHWM, deleting any other memory text or code files present.

RUN executes a LOAD of the filename given and performs a GO. Thus any memory files present are deleted first.

SAVE stores the current code file on to the current filing system under the specified name.

Note that RUN and LOAD delete both types of memory file, so they should be used with caution.

3.2 The COMPILE command

The COMPILE command is used to convert a Pascal source program into the equivalent object program. Its minimum abbreviation is 'CO.'. When using disc Pascal, you should use the command 'DCOMP' wherever 'COMPILE' appears below as this is the name of the disc compiler program. There are four versions of the command:

```
COMPILE src-file obj-file          { Compile to and from disc }
COMPILE                            { Compile to and from memory }
COMPILE src-file                   { Compile from tape/disc to memory }
COMPILE >obj-file                  { Compile from memory to disc }
```

The parts given as 'src-file' and 'obj-file' may be any valid filenames. In addition the command may be followed by some Pascal text which will be compiled as the first line of the program. If present this text must begin with a '{' character. See the discussion of compiler options in chapter 4 for more details.

Only the first form of the command is available on the disc compiler used with 6502 Second Processors, so the command to compile using the disc system is always:

DCOMP src-file obj-file

The source and object filenames may be up to 60 characters long, so the filing system in use will impose the actual limit on filenames.

The four forms of COMPILE are used as follows:

COMPILE source-file code-file

This is used when compiling large programs using the disc filing system. As neither of the files is in memory, there is maximum room for the workspace needed by the compiler. It should be noted, however, that there should be enough space on the disc for the object file and one other (temporary) workfile which is created by the compiler. A compiler error is given if the source and object filenames are the same.

COMPILE

Using this option, both the source and code files will be in memory. This is the quickest method for compiling, but has the disadvantage that the size of programs which may be compiled is limited by the computer's memory. The code file produced may be saved with the SAVE command.

COMPILE source-file

This command will compile a program from the filing system into memory. It is used when compiling large programs from tape. The source file should be created with EDIT and saved. The compiler can then use this file as its input to create a memory code file which can be SAVEd. Note that for compilation from tape to memory the cassette recorder used must be equipped with remote motor control.

COMPILE >code-file

This has a similar use to the last version, but compiles from a source file in memory to a code file on the filing system. It is not as useful because the object code is usually smaller than the source file which produced it, thus it is more efficient for the former to reside in memory. As with the first form of COMPILE, this form needs enough space on the disc for a workfile in addition to the object file, and will not work with tape.

3.3 An example compilation session

The following is a detailed example of compiling a small Pascal program using the second form of the COMPILE command. Suppose it is required to compile the following program:

```
{$W-}
program table(input,output);
{ Prints the squares and cubes of 1..10 }
var i:integer;
begin
  writeln('Number':10,'Square':10,'Cube':10);
  for i:=1 to 10 dp
    writeln(i:10,sqr(i):10,i*sqr(i):10)
end.
```

The comment at the very start of the program is a compiler option, telling the compiler not to wait for RETURN to be pressed after reporting errors. There are several compiler options, or 'switches' and they are described in detail in chapter 4. Although the program is mainly in lower case, the compiler ignores the case of all letters except those between quotes. Thus the first two lines of the program could be:

```
{$w-}
Program Table(Input,Output)
```

without affecting their interpretation by the compiler.

The program can be created using the editor as described in the last chapter. When this has been done to your satisfaction (remember that Pascal is a free-format language, so the layout does not have to follow that presented above), leave the editor using SHIFT f4 (FUNC 5), after saving the program on disc or tape if you are cautious. This will return you to Pascal command mode with the '%' prompt.

In this example we will not save the program but compile to and from memory, so simply type

COMPILE<RETURN>

The following will be produced by the compiler:

```
ISO-Pascal compiler V. R1.00

   1 0 - {$W-}
   2 0 - program table(input,output);
   3 0 - { Prints the squares and cubes of 1..10 }
   4 0 - var i:integer;
   5 0 - begin
   6 0 -   writeln('Number':10,'Square':10,
'Cube':10);
   7 0 -   for i:=1 to 10 dp
   8 0 -     writeln(i:10,sqr(i):10,i*sqr(i):10)
         ^
** Error 32 at line 8 in RAM
   9 0 - end.
1 Compilation error(s)
Code size = 105 bytes
```

The version number and code size may be slightly different, but the number of errors should be the same. If the disc compiler was being used on a 6502 Second Processor, the error message would be printed out under the error number. However, the ROM compiler doesn't print error messages by default (as the messages are held on a file which is inaccessible to tape users), so the fault must be looked up in Appendix E. This tells us that error 32 is 'Do expected'. The error was obviously caused by the misspelling at the end of line 7. To correct it, type E.<RETURN> to get back into the editor. The source program will still be there. Change the incorrect 'p' to an 'o' and exit the editor. Recompiling the program should now produce zero errors.

The line number given in error messages refers to the line within the current source file. If the compile option to chain a source file is used, and an error occurs in the second or subsequent file, the error message line number will not correspond to the number on the left of the listing, as the latter is a cumulative number that is not reset at the start of each source file.

The filename 'RAM' is used in error messages when the source file is in memory; at other times the filename specified in the COMPILE command line is used.

After printing its name and version number, the compiler starts to read the source text. Normally, a compiler listing is produced as shown above. This may be disabled by specifying the 'L-' compiler option described in chapter 4. At the start of each line is a line number. This is used for reporting compilation (and run-time) errors. The second number is the textual level: this is incremented

every time a procedure or function block is entered and decremented at the end of each such block. There are no procedures in this simple program so it remains at zero.

After the textual level there is a dash on most lines. This becomes a 'C' if the whole line is within a comment. The 'C' acts as a warning in some cases that a comment has not been closed, and large amounts of the program are being ignored.

The remainder of the line is simply the text read from the source file. At the end of the listing the number of errors is printed. On the line below, the size of the object file is given in bytes. The size of the source file in this example is 209 bytes so it can be seen that even on small programs the code file can be much smaller than the source. Although the size of the object program is always given, it may only be executed if no compilation errors occurred.

The compiler, like all Pascal programs, may be interrupted at any time by pressing ESCAPE. This will cause the message '`Escape`' to be printed and Pascal command mode to be entered. If the interrupted program was compiled with debugging code on (see below), the line number and procedure being executed when ESCAPE was pressed are printed as well. The compiler is compiled with debugging off, so the extra information is not printed.

Assuming the compilation was successful second time round, a code file will be produced. In the current example this will be in memory, and can be executed using the GO command. If a filename was cited for the code file (eg 'COMPILE >obj'), the object file can be executed using RUN, or simply by giving its name as a command, as long as this isn't the same as a Pascal system command.

Once a code file has been LOADed or RUN it is 'resident' and may be executed again using GO without the need to reload it. As noted above, several commands destroy the code memory file, so if one of these is used the code must be reLOADed or RUN.

The result of executing the example program above is:

```
Number      Square      Cube
     1           1         1
     2           4         8
     3           9        27
     4          16        64
     5          25       125
     6          36       216
     7          49       343
     8          64       512
     9          81       729
    10         100      1000
```

4 Compiler options

There are several ways in which the action of the compiler may be controlled. For example, the listing it usually gives may be suppressed, and various parts of the code produced to aid debugging may be omitted to make working programs smaller or faster. The compiler options are controlled by special comments of the form:

{$L–}

The comment delimiters may be either the braces type as shown or '(*' and '*)'. If the first character in the comment is '$', the compiler expects the following character to be a letter indicating which option is being affected (in upper or lower case). Immediately following that comes either '+' or '–' to turn the option on or off respectively. The '{$L–}' given above, for example, turns the compiler listing off. Some compiler options are followed by numbers instead. An example is '{$C200}' which sets the code size for compiling to memory.

Comments containing options may be given in the COMPILE command, after the source and/or object filenames. For example, to compile a program without a listing the following command would be used:

COMPILE source obj {$L–}

Of course, if the first line of the program proper contained the option '{$L+}', the listing would still be generated. Code given in the COMPILE command line may be regarded as line 'zero' of the source file.

More than one option may be changed at a time by separating the letters with commas. Alternatively they can be given in separate comments, so '{$L–,D–}' and '{$L–}{$D–}' are equivalent. Below is a complete list of options (or 'switches', as they are also known), together with their default settings:

Switch	Default		Meaning
C	2000	*	Specify code size
D	+	*	Generate debugging code
F	–		Produce error messages as well as numbers
G	+	*	Generate object code
I	1000	*	Reserve identifier storage
L	+		Produce a compiler listing
R	+		Generate range-checking code
S	N/A		Chain in a given source file
T	–	*	Enable the rest of the command line to be read
U	+		Generate unassigned checking code
W	+		Wait for RETURN to be pressed after errors
X	+		Enable Acornsoft extensions to be used

The switches marked with a '*' must be given before the 'program' line in the source; the rest may occur anywhere within the file. In addition, the 'C' option must occur before the first 'U' option to have any effect. The detailed meanings of these options are:

4.1 {$Ccode_size} – Specify code size

This option is used when compiling to memory. It should be given before the 'program' line. If no 'C' option is specified, the compiler will claim 2000 bytes in which to place the object code when compiling to memory. If this is not enough, or too much, a different figure may be used by specifying it in the comment. For example, suppose it is known that a program compiles into about 4000 bytes, and it is desired to compile it from tape into memory. The following option:

{$C4500}

would guarantee that the compiler reserved enough space.

4.2 {$D+} – Generate debugging code

By default the compiler produces code for each line of source which is used when reporting run-time errors. It also generates code so that the name of each procedure which is called is stored. The result is that when a run-time error occurs (eg 'Division by zero') it is possible for the interpreter to print a message giving the location of the incorrect statement.

Inserting this code takes up room and slows the program down slightly. If the compiler option '{$D–}' is given (before the program proper) the debugging code is not generated. It should be stressed that debugging programs when the location of errors is not known is very tedious, so this option should not be used until the program has been fully debugged.

4.3 {$F+} – Produce full error messages

If this switch is set, full error messages are printed out at compilation, in addition to the error numbers. The option defaults to + for the disc-based version of the compiler and to– for the ROM version. In order to make the ROM compiler print out full error messages, the option '{$F+}' must be specified. The file 'errutil' must be present on the disc in the currently selected directory. Alternatively, the disc compiler may be prevented from printing full error messages by giving the '{$F−}' option. The switch may be changed as often as desired throughout the source file.

4.4 {$G+} – Generate object code

Naturally it is usually desirable to produce a code file after a successful compilation. However, if a long program is known to contain many errors (for example, if it has been written in a non-standard version of Pascal), the important thing is to produce the error messages as quickly as possible. If the option '{$G–}' is given before the 'program' (ie first) line of the program, the compiler will check the syntax of the program, but not attempt to generate any code. This speeds up the compilation considerably when compiling to certain filing systems (eg the Econet filing system).

4.5 {$Iid_storage} – Reserve identifier storage

When a program is being compiled, all identifiers that are encountered are stored in an area of the heap (see chapter 7) claimed at the start of compilation. By default, the compiler claims 1000 bytes for this table. If we assume that the average length of an identifier is seven characters, there is room for 1000/(7+1)=125 of them. The additional byte is for the space that separates identifiers. If this limit is not sufficient, it is possible to specify the amount of space reserved by identifier names. To reserve 4000 bytes, for example, the option:

```
{$I4000}
```

would be given before the 'program' line. Another reason for changing the identifier storage would be to give the compiler more workspace by decreasing it to say 500 bytes.

4.6 {$L+} – Produce a compiler listing

It might be desired to list only certain sections (or none at all) of the source file during compilation. The 'L' option may be used to turn the listing on and off as required.

4.7 {$R+} – Generate range-checking code

Much of Pascal is concerned with the 'robustness' of programs. To this end, it is important that a variable or subscript which is declared to be in the range, say, 0 . . 255 stays there. By default, the Pascal compiler will produce code which checks such things, but again these checks make the object programs slower and larger. Giving the option '{$R–}' will turn off the generation of range-checking code until a '{$R+}' is encountered. It is recommended that range checking be left 'on' until the program has been fully debugged.

4.8 {$S'filename'} – Chain in a given source file

It is wise to split a very large source program into several files. There are several advantages in doing this: the accidental deletion of one of the files won't be as serious as if the whole program were lost; it is possible to group connected procedures and functions into a self-contained module, which can aid debugging. Finally, there is the very practical consideration that otherwise the whole of the source cannot be loaded into the computer's memory at once, and therefore can't be edited.

The switch 'S', which has a slightly different form from the rest, enables a new file to be 'chained' in. It is usually given at the end of a source file as everything after the switch is ignored. Notice that the filename is given in single quotation marks like a Pascal string, for example:

```
{$S'source2'}
```

The existing compiler options are preserved when a new file is chained in, so you don't have to disable range checking (for example) at the start of every file.

4.9 {$T+} – Enable the command line 'tail' to be read

When characters or numbers are read from the file 'input' (which is usually the keyboard) the computer waits for something to be typed. A typical example would be:

```
........
  write('How many eggs? ');
  readln(numEggs);
........
```

In this case the prompt would be displayed and the computer would wait for a line to be typed, followed by RETURN. (For the technically minded, Pascal uses OSWORD 0 to do its input, as does BASIC.) When a program needs only one or two items of input from the user, it is more convenient to have them typed

after the name of the command. This is common with disc utility commands, for example:

*FORM80 1

Using this method, the FORM80 command doesn't have to prompt the user '`Format which drive?`'; it simply reads it from the rest of the command line. Pascal programs have a similar facility when they are compiled with the '{$T+}' option. When this is done, the initial line of input to the program is taken from the text following the name of the program in the command line. If, for example, the program was called with the line:

TABLE 100 200

the file 'TABLE' would be executed and the first two Pascal reads of integers would produce 100 and 200. After the 200 is read, eoln becomes true, as usual, and further input would be sought from the keyboard.

If simply:

TABLE

was issued as the command, and the file had been compiled with the 'T' option on, then eoln would be true before any reads had been performed.

A concrete example will make the use of this option clearer. Suppose the 'table' example given earlier was amended so that it takes the first and last number in the table from the user. The user can type the values either once the program is running or on the command line. The new version of the program is:

```
{$T+}
program table(input,output);
{ Prints the squares and cubes of first..last }
var i:integer;
    first, last:integer;
begin
  if eoln then
    write('Type first-number last-number: ');
  readln(first,last);
  writeln('Number':10,'Square':10,'Cube':10);
  for i:=first to last do
    writeln(i:10,sqr(i):10,i*sqr(i):10)
end.
```

If the program is compiled with the line

COMPILE >table

it may be executed with 'RUN table' or simply 'table'. Executions subsequent to the first one require just 'GO'. If you type

table<RETURN>

the prompt '`Type first-number last-number: `' will be given. This is because, as described above, eoln is true on entry to a program which has been compiled with '{$T+}' and has no 'tail' in the command line. However, if you instead invoke the command with:

table 1 20 <RETURN>

then eoln will be false on entry so the prompt won't be printed. Furthermore, the two numbers required by the program will be read from the command line tail, so the table will be printed immediately. Once the program is in memory, it can be rerun by lines of the form:

GO 5 25

Contemplate what happens if you type:

table 1<RETURN>

There is a tail to the command so eoln will not be true on entry and the prompt will not be printed. The first number will be read from the command line, as above. The second number, however, will have to be typed in as usual by the user. This is clearly quite nasty as the machine appears to 'die' while waiting for input. A way round this is to check for eoln at each input (rather than just the first one as in the current program), and print a prompt if it is true.

When performing Pascal character reads on the command tail, spaces after the name of the command are skipped, as are trailing spaces, so the command

file abcde <RETURN>

will produce the following results:

State of eoln	Character read
false	a
false	b
false	c
false	d
false	e
true	<SPACE>

After the end of line space character, further input will result in the keyboard being used as normal.

4.10 {$U+} – Generate unassigned checking code

It is a rule in Pascal that variables' values should not be used until they have been assigned in some way. So in the program:

```
program test(input,output);
var i:integer;
begin
  write(i)
end.
```

an error will be produced when the program is executed. In Acornsoft ISO-Pascal, unassigned checking will be performed by default on all variable accesses. This is implemented by setting every byte in the variable to hex &B8 (184 in decimal) at the start of the program (or procedure). This means that for every type, there is a single 'illegal' value. If that value is assigned to a variable in the course of a program, eg:

```
c := chr(184);
```

then future attempts to reference the variable will lead to 'unassigned variable' errors. This is only a problem with single-byte types, eg characters and enumerated, as the likelihood of the illegal integer and real values (−1195853640 and −5.199449944E16 respectively) occurring in a program is very remote. A pointer with internal value &B8B8 can never occur in the current Pascal system.

Since it is important to enable the whole of the character range to be assigned, a compromise unassigned checking option is available. If the option:

{$U*}

is given, unassigned checking will be applied to all variables except single-byte ones. This provides enough 'security' for most programs. Unassigned checking may be disabled altogether using the option '{$U–}'.

It should be noted that having this type of checking does not significantly alter the size of the object program; it merely adds two bytes at the start of the object file to tell the interpreter to perform checking. This is the reason that the flag should be specified before the 'program' declaration.

4.11 {$W+} – Wait for RETURN to be pressed

When the compiler detects a syntax error, it reports the fact to the user. It then waits for the user to press RETURN before it starts recompiling. This prevents the error messages from being scrolled off the screen before the user gets a chance to read them.

It may be that the user is sending the error messages to a printer (by issuing a CTRL B before starting the compilation) or to a *SPOOL file. If this is true, they may want to disable the 'wait for RETURN' facility so that the whole program may be compiled without human intervention. This can be achieved by giving the compile option '{$W–}'. Once it has been given, error messages will be printed with no pauses until the option '{$W+}' is issued.

Note that the switch does not have to be given before the program header, so the facility can be used selectively throughout the source file. This is true of all the switches except for C, D, G, I and T. The S switch can be used anywhere, but only once per file as anything after it in the source file is ignored. The S option should not be used in the COMPILE command line.

4.12 {$X+} – Enable Acornsoft extensions

A compilation with no options specified will result in a debuggable object file of an ISO-Pascal program being produced. It has been seen that some of the options make the program less easy to debug (eg by turning off range checking) and some make the program less portable (ie less likely to run on other Pascal implementations, for example by enabling the command 'tail' to be accessed). The last option to be described falls into the latter category – it enables various Acornsoft-specific extensions to be recognised by the compiler. The option defaults to 'on' so in order to compile a program that is guaranteed to be standard, the first line should be '{$X–}', or the extensions may be disabled in the command line, for example:

```
compile source {$x-}
```

The extensions provided are discussed in detail in the next chapter.

5 Acornsoft language extensions

The extensions provided can be categorised in different ways. There are new procedures, functions and extensions to the syntax. The new features relate mainly to the BBC Microcomputer's graphics and sound facilities. There is also a facility for accessing machine code programs from Pascal.

The new procedures and functions are described below, along with the way they would be headed if being defined explicitly.

5.1 Graphics procedures and functions

```
procedure mode(modeNumber : integer);
procedure vdu(byte1, byte2, ...byte64 : integer);
procedure plot(plotOption, x, y : integer);
function  point(xCoord, yCoord : integer) :
integer;
```

All of these have direct equivalents in BBC BASIC. The only slightly unusual procedure is 'vdu' which can take a variable number of parameters. There can be between 1 and 63 parameters and these are sent as bytes to the VDU drivers. Note that there is no equivalent to the semicolon in BBC BASIC's VDU statement.

All the other graphics commands built into BBC BASIC can be emulated using the 'vdu' procedure. Below is the heading of a program which defines all of these, though typically only a couple will need to be used:

```
program graphics(input,output);
{ Defines the BBC BASIC graphics functions }

const
  store_    = 0;
  or_       = 1;
  and_      = 2;
  eor_      = 3;
  not_      = 4;
  nop_      = 5;

  black     = 0;
  red       = 1;
  green     = 2;
  yellow    = 3;
  blue      = 4;
  magenta   = 5;
  cyan      = 6;
  white     = 7;
  fBlack    = 8;
  fRed      = 9;
  fGreen    = 10;
  fYellow   = 11;
  fBlue     = 12;
  fMagenta  = 13;
  fCyan     = 14;
  fWhite    = 15;


procedure move(x, y : integer);
const
  moveCode = 4;
begin
  plot(moveCode, x, y)
end;

procedure draw(x, y : integer);
const
  drawCode = 5;
begin
  plot(drawCode, x, y)
end;
```

```
procedure colour(col : integer);
const
  colourCode = 17;
begin
  vdu(colourCode, col)
end;

procedure gcol(plotMode : integer; col : integer);
const
  gcolCode = 18;
begin
  vdu(gcolCode, plotMode, col)
end;

procedure clg;
const
  clgCode = 16;
begin
  vdu(clgCode)
end;

procedure cls;
const
  clsCode = 12;
begin
  vdu(clsCode) { or just 'page' }
end;

procedure palette(logical : integer; actual :
integer);
const
  paletteCode = 19;
begin
  vdu(paletteCode, logical, actual, 0, 0, 0)
end;
```

```
procedure tab(x, y : integer);
const
  tabCode = 31;
begin
  vdu(tabCode, x, y)
end;

begin { main }
end.
```

There are a couple of extra procedures defined, namely 'palette' and 'tab'. The former is known as 'VDU 19' under BASIC, and is used to assign particular actual colours to the logical colours. The 'tab' procedure moves the cursor to a given tab position.

Notice that enumerated types are used to specify the types of plotting modes (in 'gcol') and the actual colours (in 'palette'). Thus to change the graphics colour to logical colour 2, using exclusive-or plotting, the statement '`gcol(eor_, 2)`' would be used. The underline after the plotting modes stops them from clashing with Pascal reserved words (in the case of 'and_', 'or_' and 'not_').

Similarly, when setting an actual colour using 'palette', statements may be of the form '`palette(2, cyan)`'. The colours prefixed by 'f' are the flashing colours.

There is no reason why other vdu sequences, such as 'define graphics origin', should not be given procedures of their own. The section on VDU in the *BBC Microcomputer System User Guide* lists the appropriate codes.

5.2 Sound procedures

```
procedure sound(channel, amplitude, pitch,
                duration : integer);
procedure envelope(param1, param2, ...param14 :
                   integer);
```

The order of the parameters in sound and envelope is identical to that in BBC BASIC.

5.3 The oscli procedure

```
procedure oscli(command : string);
```

This procedure sends the string to the command line interpreter (as the OSCLI statement does in BASIC II). The type 'string' in this context should be defined as:

```
type string = packed array[1..len] of char;
```

where len is an integer constant between 2 and 255. Alternatively, of course, a string constant may be given as in: `oscli('cat 1')`. The program below repeatedly takes a string from the user and sends it to the operating system. Notice that there is no need to put a carriage return character on the end of the string – Pascal does this automatically. Note that the procedure call `oscli('.')` is illegal as `'.'` is of type char, not string. The call `oscli('. ')` is OK.

```
program oscli(input,output);

const
  len = 10;

type
  subscript = 1..len;
  string = packed array[subscript] of char;

var
  lin : string;
  ch  : char;
  i   : integer;

begin
  repeat
    lin := '          ';{len spaces}
    write('Command: ');
    i := 1;
    while not eoln and (i <= len) do
      begin
        read(ch);
        lin[i] := ch;
        i := i + 1
      end;
    readln;
    oscli(lin)
  until lin = '          '{len spaces}
end.
```

5.4 Miscellaneous operating system facilities

```
function  adval(channel : integer) : integer;
function  inkey(delay : integer) : integer;
function  time : integer;
procedure settime(timeVal : integer);
```

These four also have direct equivalents in BBC BASIC. The function 'time' and the procedure 'settime' emulate the BASIC psuedo-variable TIME. The 'adval' and 'inkey' functions give exactly the same result as the BBC BASIC functions of the same name for any argument. Thus 'inkey' may be used to wait for a key to be pressed in a given time, or examine the status of a particular key. 'adval' may be used to interrogate the analogue to digital converters, or examine the status of the various buffers. See the *BBC Microcomputer System User Guide* or the *Acorn Electron User Guide* for details. Below is a short program to demonstrate the use of 'time' and 'settime'.

```
program time(input,output);

const
  tab = 31;

var
  t,h,m,s  : integer;

function secs(t : integer) : integer;
begin
  secs := t div 100 mod 60
end;

function mins(t : integer) : integer;
begin
  mins := t div 6000 mod 60
end;

function hrs(t : integer) : integer;
begin
  hrs := t div 360000 mod 24
end;
```

```
begin
  page;
  writeln;
  write('Type in the time as hours minutes
        seconds ');
  readln(h,m,s);
  settime(100*(s+60*m+3600*h));
  repeat
    repeat until time mod 100 = 0;
    t := time;
    vdu(tab,0,5);
    write(hrs(t):2,':',mins(t):2,':',secs(t):2)
  until inkey(0)=ord(' ')
end.
```

5.5 String conversion functions

```
function  ival(str : string) : integer;
function  rval(str : string) : real;
```

These functions provide fast conversion between strings and numeric types. The string parameter is interpreted as an integer or real in 'ival' and 'rval' respectively. The form of the number allowed is exactly that which is valid on numeric reads. The string will be read up to the first non-numeric character. If this occurs before any numeric characters have been read, an error will be generated, otherwise the 'result so far' will be returned, for example:

`rval('1.23fred')`	returns 1.23 and no error
`ival('fred1')`	generates a run-time error

5.6 Extended file procedures

```
procedure   reset(fileName : fileType;
                  externalName : string);
procedure rewrite(fileName : fileType;
                  externalName : string);
```

These extended versions of the standard procedures 'reset' and 'rewrite' enable file variables to be linked with external (ie filing system) names. The first parameter is the same as in the standard procedures. The second parameter is a string giving the name to be used by the current filing system. An example is:

```
reset(infile, ':1.text1');
rewrite(outfile, 'text2');
```

Of course, the strings may be variables, for example:

```
reset(data,usersName);
```

Note that there are two types of files: temporary and permanent. Temporary files have external names of the form 'pas___0', 'pas___1' and so on. They are created when the standard forms of 'reset' and 'rewrite' are used, and are deleted at the end of the file variable's scope.

Permanent (named) files are accessed using the extended forms of 'reset' and 'rewrite' and are not deleted at the end of the program. File variables which are linked with permanent files must appear in the program header.

A special case of the extended forms for 'reset' and 'rewrite' is when the string parameter is a null or its first character is a carriage return, eg '`reset(file,'')`'. In this case, the file is bound to the console, ie the screen, for output and the keyboard for input. The file must be a text file. This is useful as it enables users to specify the console as a file, so that, for example, text can be seen on the screen instead of being sent to a disc file which has to be examined later.

Programs which process data until the 'eof' condition becomes true for a file can still work when the keyboard is used. If the user enters a character with ASCII code &80, eof is set to true for the file 'input' (or any file reset as the console as described above). To obtain this code, SHIFT f0 (FUNC A) should be used. Note that as Acornsoft Pascal's input is line-buffered, the actual sequence required is SHIFT f0<RETURN> (FUNC A <RETURN>).

5.7 Extended dynamic storage handling

```
procedure release(address, bytes : integer);
function  claim(bytes : integer) : integer;
function  free : integer;
```

These functions and the procedure enable the programmer to obtain areas of storage whose size is determined at run-time and later release them. The function 'claim' takes a parameter which is the size of the block required in bytes. It returns an integer which is the machine address of the start of the block.

The procedure 'release' takes two parameters: the address of the block to be returned to the heap and the number of bytes being given back. This should

obviously be the same as the number of bytes claimed in the first place.

The function 'free' returns the number of bytes which are free for use by dynamic data structures. In theory, it gives the size of the biggest block that can be 'claimed', but in practice it is wise to use only, say, 75 per cent of this amount to leave the stack room to grow.

It may seem rather strange that the function 'claim' returns a machine address rather than a pointer. The problem is that pointers have to be bound to particular types, and there is no type called 'chunk of memory'. To use the memory obtained by 'claim' some nefarious dealing with variant records has to be undertaken. As a simple example, here is the code to emulate 'peek' and 'poke' found on some machines:

```
{$U-}
{ It is necessary to turn  unassigned checking
off in case address &B8B8 is accessed as the
address &B8B8 is the 'unassigned' pointer value.
Thus when  fudge.addr^ is attempted, the
unassigned checking code will object.}

program peek_poke(input,output);

type
   byte = 0..255;
   word = 0..65535;
   fudgerec =
     packed record
       case boolean of
         false : (addr : ^byte);
         true  : (int  : word)
     end;

var
   i : word;
  pi : byte;

function peek(address : word) : byte;
var
   fudge : fudgerec;
begin
   fudge.int:=address;
   peek:=fudge.addr^
end;
```

```
procedure poke(address : word; value : byte);
var
   fudge : fudgerec;
begin
   fudge.int:=address;
   fudge.addr^:=value
end;

begin
   for i := &8000 to &8100 do
      begin
        pi := peek(i);
        write(~i, ~pi);
        if pi > 31 then
           writeln(chr(pi) : 4)
        else
           writeln
      end
end.
```

It can be seen that to set the address to be peeked (or poked), the 'int' field is used and to access the contents, the indirected 'addr' field is used.

5.8 Machine code linkage functions

```
function  code0(address, a, x, y : integer) :
                integer;
function  code1(address, a : integer;
                var variable : anyType) : integer;
```

These functions allow 6502 machine code routines to be used from Pascal. The first version allows the user to call a given address, specifying the contents of the A, X and Y registers on entry. The machine code should return with an 'rts' instruction. The result of the function is an integer whose four bytes could be expressed as &PYXA in hexadecimal, ie the least significant byte is the value of A on exit, the next byte is X, then Y and the most significant byte is the value of the status register on exit.

The function 'code1' enables a Pascal variable to be accessed from the machine code. The first two parameters are the address of the routine and the value to be put in A on entry. The third parameter can be any type. The start address of the variable is passed to the machine code in X and Y (low byte, high byte respectively). For the format and size of different types see chapter 8.

The address in X and Y is of the variable specified as the third parameter, not a copy of it. This is analogous to the parameter being specified as a 'var' type in the declaration of the function 'code1', so that by changing the contents of the address (and subsequent bytes) the variable may be altered by the machine code.

The main use of these functions is to access operating system facilities which aren't supported through existing Pascal extensions. An example is the BASIC 'GET' function, which returns the character code of the next key in the buffer. A Pascal function to do this is shown in the program below:

```
program fgetp(input,output);
const
  cr  = 13;
  etx = 128;

type
  byte = 0..&FF;

var
  i    : byte;

function fget : byte;
const
  { OS get char function. Returns code in A }
  { A, X and Y on entry are immaterial }
  osrdch = &FFE0;
begin
  fget := code0(osrdch,0,0,0) mod &100
end;

begin
  repeat
    i := fget;
    if i=cr then
      writeln
    else
      vdu(i)
  until i=etx
end.
```

The function is named 'fget' so that its declaration doesn't redefine the Pascal standard procedure 'get' (although this would not matter in the current program as 'get' is not used).

The code1 function is used in situations where a parameter block is required by the OS routine. An example is the 'read character definition' call, which is osword 10. The program below prints the character set in large characters:

```
program big(input,output);
var
  ch : char;

procedure printChar(ch : char);
const
{ A=10 for char. definition. XY points to par.
  block }
  osword  = &FFF1;
  getPatt = 10;
  width   = 8;
  height  = 8;

type
  byte   = 0..&FF;
  xCoord = 1..width;
  yCoord = 1..height;
  parBlk = packed
    record
      ch   : char;
      patt : packed array[yCoord] of byte
    end;

var
  block    : parBlk;
  row      : yCoord;
  col      : xCoord;
  thisbyte : byte;
  dummy    : integer;
```

```
begin
  block.ch := ch;
  dummy := code1(osword, getPatt, block);
  for row := 1 to height do
    begin
      thisByte := block.patt[row];
      for col := 1 to width do
        begin
          if thisByte >= 128 then
            write('*')
          else
            write(' ');
          thisByte := thisByte mod &80 * 2
        end;
      writeln
    end
end;

begin {main}
  for ch := ' ' to '~' do
    begin
      printChar(ch);
      writeln
    end
end.
```

Note that you must take care when defining the record to be used as the parameter block. There must be a one-to-one correspondence between the structure and the bytes that the operating system routine expects to find. For example, if the array in 'block' had not been packed, its elements would have been four bytes each instead of the desired one byte. Chapter 8 details the sizes of the various packed and unpacked data types in Acornsoft ISO-Pascal.

5.9 The 'otherwise' clause

The last extension is an addition to the syntax of the case statement. The 'otherwise' clause enables action to be taken if none of the case constants matches the case expression. This obviates the necessity to check that the case expression is in a given set before executing the statement. The syntax of the extension is best illustrated with a trivial example:

```
program trivial(input,output);
var
  i:integer;
begin
  for i:=1 to 5 do
    case i of
     1: writeln('One');
     2: writeln('Two')
    end
    otherwise
     writeln('Greater than two')
end.
```

The 'otherwise' follows the 'end' of the case statement and is followed by the statement to be executed if none of the other cases match. The output of the program is:

```
One
Two
Greater than two
Greater than two
Greater than two
```

5.10 Hexadecimal numbers

With the extensions enabled, it becomes possible to deal with hexadecimal numbers in the source program. Hex constants are specified in the usual Acorn way by preceding them with an ampersand '&'. Hex numbers may contain the digits '0'–'9', 'A'–'F' and 'a'–'f'. For example:

```
const
   return = &0D;
   delete = &7f
```

Furthermore, the standard procedures 'write' and 'writeln' are extended to enable integers to be printed in hex by prefixing expressions with tilde, '~', for example:

```
write(~free, 'bytes free');
```

5.11 The underscore character

When extensions are enabled, it is permitted for identifiers to contain the underscore character '_' in addition to digits and letters. Note that there is no

limit on the length of Acornsoft ISO-Pascal identifiers, and all characters are significant. The underscore character may not be the first character of an identifier.

5.12 Packed var parameters

Normally, it is illegal to pass an element of a packed record or array to a procedure or function as a var parameter. In Acornsoft ISO-Pascal with extensions enabled, it is possible to do this as long as the size of the element does not change when packed, ie anything but integer and subrange of integer is allowed. Thus, in the declaration:

```
var pearl :
    packed record
       john : real;
       pete : 0..26;
       deb  : integer
    end;
```

The field pearl.john could be passed as an actual variable parameter but pearl.pete and pearl.deb could not.

6 Executing programs

Once a program has been compiled without errors it may be executed. There are three ways of doing this. If the object code is on the filing system, it may be executed using either

RUN filename<RETURN> or simply filename<RETURN>

In the latter case, the file shouldn't be called something like 'COMPILE' as Pascal will interpret it as a command.

Once a program is in the machine, it can be executed simply by issuing the command:

GO<RETURN>

In each case, any command line parameters to be read by the program should be put just before the <RETURN>, as in:

RUN filename 1 2 3 4 5<RETURN>
filename 1 2 3 4 5<RETURN>
GO 1 2 3 4 5<RETURN>

If an error occurs during execution a message is given describing the fault, eg 'Division by zero'. Providing the 'D' option wasn't disabled at compile time, the location of the error (line number and procedure name) will also be printed.

6.1 Tracing execution

Using the 'TRACE' command, it is possible to trace the execution of programs. There are three levels of tracing: off, procedures only, and line numbers and procedures. These levels are set by following 'TRACE' with the digit 0, 1 and 2 respectively. Thus, if you issue the command 'TRACE 2', the next program to be executed will have the name of each procedure executed as it is entered, and every line number encountered will be printed.

Note that it is only possible to trace programs that were compiled with the 'generate debugging code' option enabled (the default setting). Programs compiled with the option '{$D–}' will be unaffected by enabling the trace facility.

For example, consider the program listed below

```
program tracer(output);

var
   i : integer;

procedure inc(var i : integer);
begin
  i := succ(i)
end;

begin
  i:=0;
  repeat
   inc(i)
  until i=10
end.
```

When this is executed after a 'TRACE 1' command, the following output is produced:

```
(TRACER) (INC) (INC) (INC) (INC) (INC) (INC)
(INC) (INC) (INC) (INC)
```

After 'TRACE 2', the traced output from the program looks like this:

```
(TRACER) [12] [14] [14] (INC) [8] [15] [14] (INC)
[8] [15] [14] (INC) [8] [15] [14] (INC) [8] [15]
[14] (INC) [8] [15] [14] (INC) [8] [15] [14]
(INC) [8] [15] [14] (INC) [8] [15] [14] (INC) [8]
[15] [14] (INC) [8] [15]
```

Notice the procedure names are in round brackets and the line numbers (corresponding to those given in the compiler listing) are printed between square brackets.

The compiler produces trace instructions at the start of each statement, after calls to user-defined functions, and when UNTIL keywords are encountered. This ensures that all lines of program source have debugging information generated for them, unless an expression extends over more than one line.

7 Memory organisation

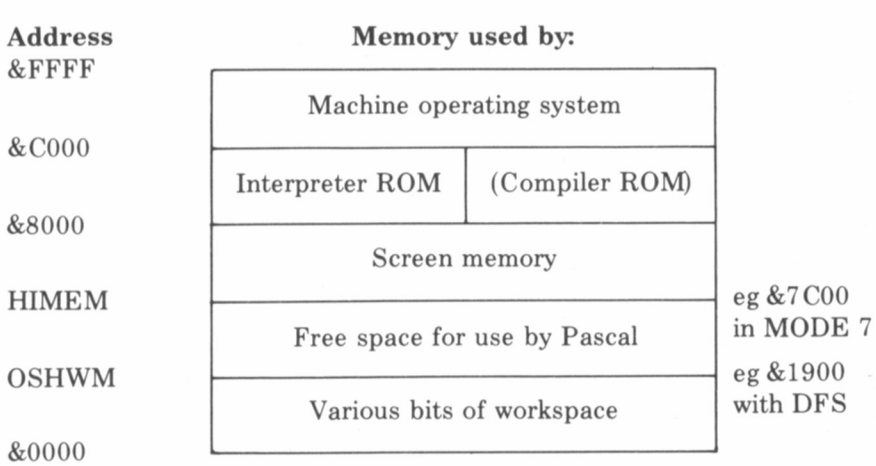
As mentioned in chapter 1, Pascal comes in two 16K ROMs (contained within a cartridge pack on the Acorn Electron). One of them contains the Pascal compiler (which is a large Pascal object program) and the other holds the Pascal run-time system, the editor and the command line interpreter. The code in the interpreter ROM is 6502 machine code.

7.1 The memory map in the I/O processor (BBC Microcomputer or Acorn Electron)

When the command:

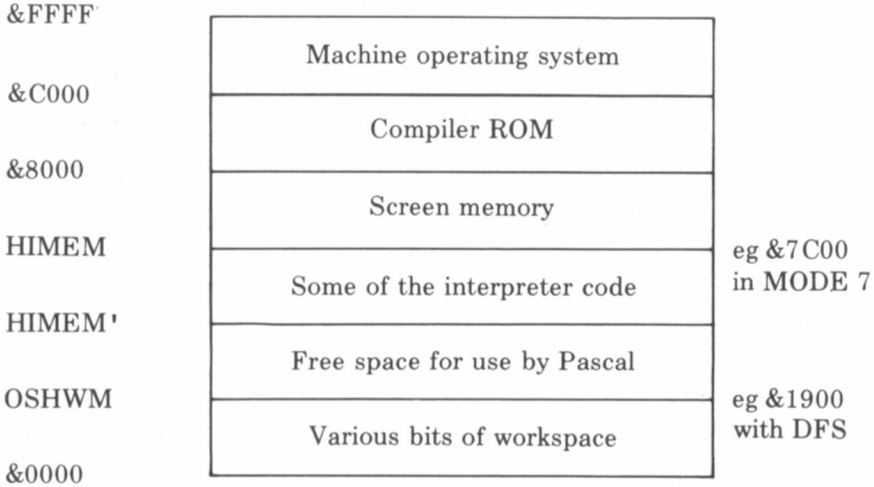
*PASCAL<RETURN>

is issued, it is the interpreter ROM which is called. The memory map in the BBC Microcomputer without a 6502 Second Processor under Pascal is shown below diagrammatically, but not to scale:



The label 'Compiler ROM' is in brackets as when normal Pascal programs are running, or the editor or command line interpreter is active, the interpreter ROM is paged in and the compiler ROM is 'invisible'. It can be seen that on a disc machine in MODE 7, there are &7C00–&1900 or 25344 bytes for use by Pascal. The way in which this space is used is described in section 7.3 'Use of free space'.

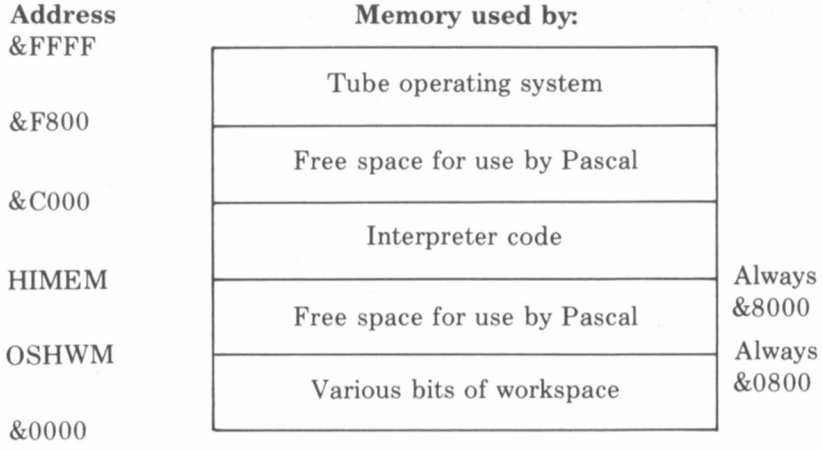
When the COMPILE command is issued, the memory map alters as shown below:



The differences are that a new version of HIMEM, HIMEM', marks the upper limit of free memory and the compiler ROM is paged in. Between HIMEM' and HIMEM is a copy of some of the code of the interpreter– enough to execute the compiler ROM. The size of this section of the interpreter is about 8 kilobytes.

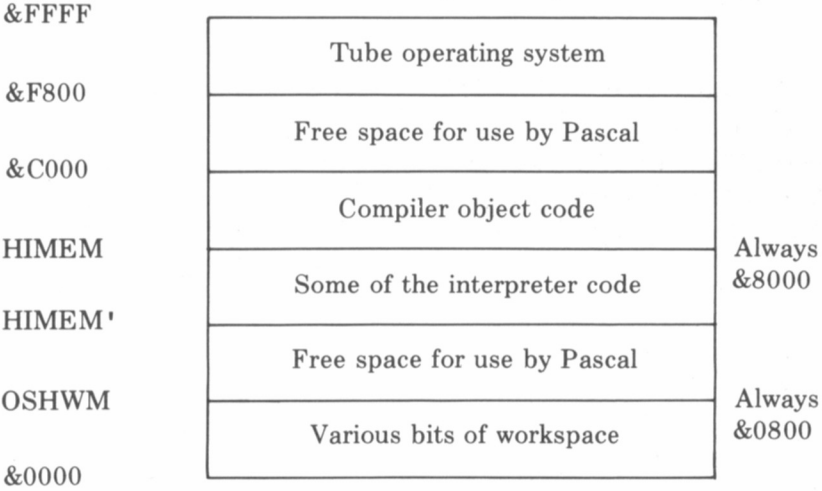
7.2 The memory map in the 6502 Second Processor

There are slight differences in the layout of the memory map when Pascal is running in the 6502 Second Processor. These relate to the different values of OSHWM and HIMEM and the lack of an MOS ROM. The memory map when running a normal Pascal program is:



It is easy to see that there is much more free space when a 6502 Second Processor is used – a total of 44 kilobytes. Also, the different display modes do not affect the amount of free memory, so it is possible to edit large programs in, say, MODE 3.

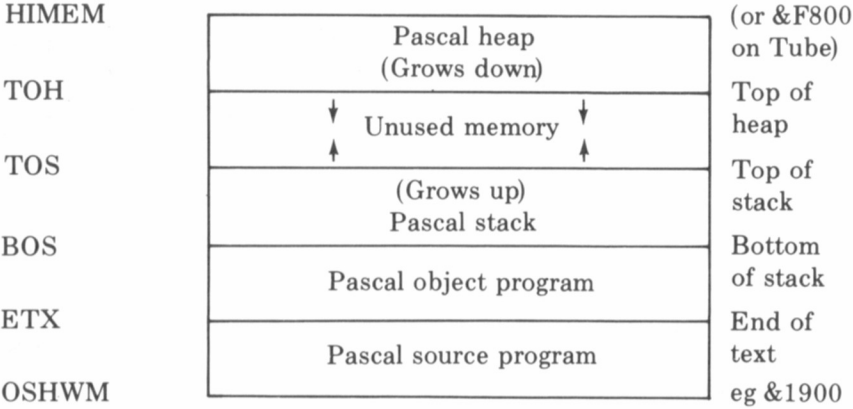
The memory map changes yet again when the compiler is called in a 6502 Second Processor system. This time it is:



This time the compiler ROM is copied across to the 6502 Second Processor and the interpreter code required to run it is again put just below HIMEM.

7.3 Use of free space

The sections of memory that aren't being used by the operating system or Pascal itself are free to be used by the user's Pascal programs. The layout of the free space is:



As can be seen, there are four structures involved, two of which (the heap and stack) dynamically 'grow' towards one another when a program is running. The area 'Pascal source program' isn't always present, so the 'Pascal object program' will sometimes start at OSHWM.

The heap grows from the top of free space (either HIMEM or &F800 on a non-Tube and Tube machine respectively). It contains dynamic variables created by the 'new' procedure. As more variables are created, the heap grows down. When a dynamic variable is disposed of, the heap does not shrink, but the storage being released is inserted into a 'free list' so that it may be re-used. Acornsoft ISO-Pascal has some powerful extensions for dealing with the heap: 'free', 'claim', and 'release' (see section 5.7).

8 Internal formats

All versions of Pascal have details which are specific to the implementation, eg size of integers, limits on enumerated types, etc. This chapter describes these details for Acornsoft ISO-Pascal. They are useful to know when writing machine code routines which will be called from within Pascal (see section 5.8 and chapter 9).

8.1 Size and ranges of simple types

The table below gives the number of bytes required for each of the simple types, and what their ordinal ranges are.

Type	Minimum val	Maximum val	Size in bytes
char	0	255	1
boolean	0	1	1
integer	−2147483648	2147483647	4
real	−1.70141183E38	1.70141183E38	5
set	0 . . 0	0 . . 255	32
enumerate	0	255	1
pointer	0	65535	2
file	–	–	5+element size

8.1.1 Character types

The character set used in Acornsoft ISO-Pascal is that used by the BBC Microcomputer. This is a slightly modified form of ASCII in that chr(96) gives a pound sign ('£') instead of a grave accent. The character chr(128) is used to mark the end of file 'input' if encountered on a 'read', 'readln' or 'get(input)'. This code may be obtained from the keyboard by pressing SHIFT f0. The size of a character is, of course, one byte.

8.1.2 Boolean types

The boolean constants 'false' and 'true' have the ordinal values 0 and 1 respectively (as required by the Standard) thus 'false < true' is true. They are stored as single bytes internally, the value of the byte being the ordinal value of the boolean.

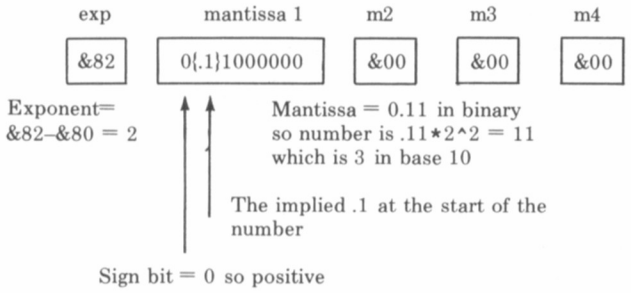
8.1.3 Integer types

Integers are four bytes long and are stored in twos-complement form. This gives

a range of −2^31 to +2^31−1. The constant 'maxint' has the value 2147483647 in Acornsoft ISO-Pascal. In memory and on files integers are stored with the least significant byte first.

8.1.4 Real types

Reals are stored in five bytes: a one byte exponent followed by four bytes of mantissa. The exponent is held in excess-128 form. The four bytes of the mantissa are stored with the most significant byte first. Bit seven of the first byte of the mantissa is the sign bit of the whole number – 0 for positive, 1 for negative. Reals are normalised such that there is a notional binary '.1' before the most significant bit of the mantissa proper. The diagram below makes this clearer:



It shows how the sequence of bytes &82, &40, &00, &00, &00 represents the number 3.

8.1.5 Set types

Although sets are not really simple types, they have a uniform size in Acornsoft ISO-Pascal so will be included here. Set types are 32 bytes (256 bits) long. Thus a set can have up to 256 elements, one bit in the set representing the presence or absence of a member. The ordinal values in a set must be in the range 0 . . 255, so the following are legal:

```
type
set1   = set of char;      {ordinal values 0..255}
colour = (red,green,blue);
set2   = set of colour;    {ordinal values 0..2}
set3   = set of 50..150;   {ordinal values 50..150}
set4   = set of 'A'..'Z'; {ordinal values 65..90}
```

The following, however, are not:

```
type
set5 = set of integer;      {ordinal values
                              -maxint-1..maxint}
set6 = set -1..254;         {ordinal values -1..254}
```

It is an important property of Acornsoft ISO-Pascal that it is legal to have sets of char, as these are used frequently. In storage, bit 0 of the first byte of the set marks the presence or absence of the member with ordinal value 0; bit 7 of the last byte marks the presence or absence of the member with ordinal value 255.

8.1.6 Enumerated types

Enumerated types are stored as single bytes. The first identifier in the enumeration has an ordinal value of 0, the second is 1 and so on. The maximum number of identifiers in an enumeration is 256. Again, this is a very realistic limit, and allows sets of all possible enumerated types to be valid.

8.1.7 Pointer types

Pointers are two bytes long. Their contents are not accessible directly, though the two bytes are in fact 6502 addresses stored low byte first. The address held in a pointer is of the first byte of the object to which it points. These objects are located in the heap as mentioned earlier, and when a pointer object is disposed of it becomes linked into a free list so the memory can be re-used.

8.2 Complex types

Arrays, files, records and sets are constructed from the simple types mentioned above. Sets have already been covered and won't be mentioned further. The complex types may be specified as 'packed'. In Acornsoft ISO-Pascal packing affects only the two following integer subranges:

low.. high where 0<=low<=255 and low<=high<=255 is stored as one byte, and
low.. high where 0<=low<=65535 and low<=high<=65535 is stored as two bytes

For example:

```
type
   byte   = 0..255;
   word   = 0..65535;
   {256 one-byte elements}
   codes = packed array[byte]   of byte;
   {10  two-byte elements}
   addrs = packed array[1..10] of word;
   brkp  = packed record
                    locn : word;      {2 bytes}
                    val  : byte       {1 byte}
                  end;
```

Unpacked structures or integer ranges outside those cited above result in the same storage being taken as normal integers. For example:

```
type
   big = packed array[1..10] of -100..0;
```

will result in four-byte integers being used even though the ordinal range of the base type is only 101, because neither of the packed integer subranges apply.

8.2.1 Array types

The elements of arrays are stored contiguously, the first element appearing lowest in memory. Consider the simple array:

```
var
   simpleArray : array[1..10] of integer;
```

The ten elements are stored with simpleArray[1] in the first four bytes, followed immediately by the four bytes of simpleArray[2], and so on. Thus the whole structure occupies 40 bytes.

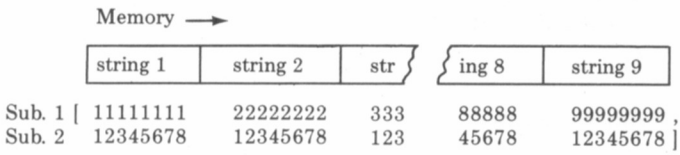
A slightly more complex example is:

```
var
   textArray : packed array[1..9,1..8] of char;
```

As this is equivalent to the declaration:

```
var
   textArray : packed array[1..9] of
                  packed array[1..8] of char;
```

it can be seen that the structure is an array of nine strings of eight characters each. These are stored thus:



It can be seen that the second subscript increases more rapidly than the first; it is the general case with multi-dimensional arrays that the later subscripts increase first. This is known as storing the arrays in row-major order.

The total amount of storage required for an array can be calculated as follows. Suppose an n-dimensional array is declared:

```
var
  eg : array[low1..high1,low2..high2, ...
             lown..highn] of t;
```

The number of bytes needed is:

(high1 − low1 + 1) * (high2 − low2 + 1) * . . . (highn − lown + 1) * size(t)

where size(t) is the size of the base type of the array (taking any 'packed' modifiers into account). For the string array given earlier, the size is:

(9 − 1 + 1) * (8 − 1 + 1) * size(char) = 9 * 7 * 1 = 63 bytes

Another example is:

```
var
   anArray : packed array[-10..9,20..30] of 1..31;
```

The appropriate calculation is:

(9 − (−10) + 1) * (30 − 20 + 1) * size(1 . . 31) = 20 * 11 * 1 = 220 bytes.

Note that size(1 . . 31) is one since the array is packed. If it wasn't, the size would be four.

8.2.2 Record types

The organisation of a record can be viewed as simply the concatenation of its component types. As a record may be packed, subranges of integers may occupy 1, 2 or 4 bytes. Consider this declaration:

```
type
  rec = record
          x,y : 0..99;
          name : packed array[1..10] of char;
          shape : (circle, ellipse, square)
        end;
```

The total size of this structure is 19 bytes. The first four are the field 'x'; the next four are the field 'y'; the next ten are the field 'name' and the last byte is the field 'shape'.

If the record were specified as being packed, the fields 'x' and 'y' would only occupy one byte each as they are in suitable subranges. The size of the record would then only be 13 bytes.

8.2.3 File types

Files are declared in the program header and also in the variable declaration part of the program. The exceptions are the predefined text files 'input' and 'output' which only appear in the program header, and the so-called temporary files that are local to procedures and functions. The latter appear only in the declaration part of the appropriate procedure.

Once again, the only effect of declaring a file to be packed is when its base type is a subrange of integer with upper and lower bounds in the range 0 . . 255 (one byte) and 0 . . 65535 (two bytes). The way in which data is stored in a file is an exact copy of how it appears in memory. Hence, a file declared thus:

```
type
   recFile = file of rec; {rec defined above}
```

would consist of records of 19 bytes, arranged in the same order as described above. Note that altering the definition to:

```
type
   recFile = packed file of rec;
```

would not alter the size of each record: only making 'rec' packed would do that.

File variables are stored in two parts: a five-byte file descriptor followed by the file buffer variable. If the code1 function is used to access a file variable, the part of the variable pointed to by XY depends on the call:

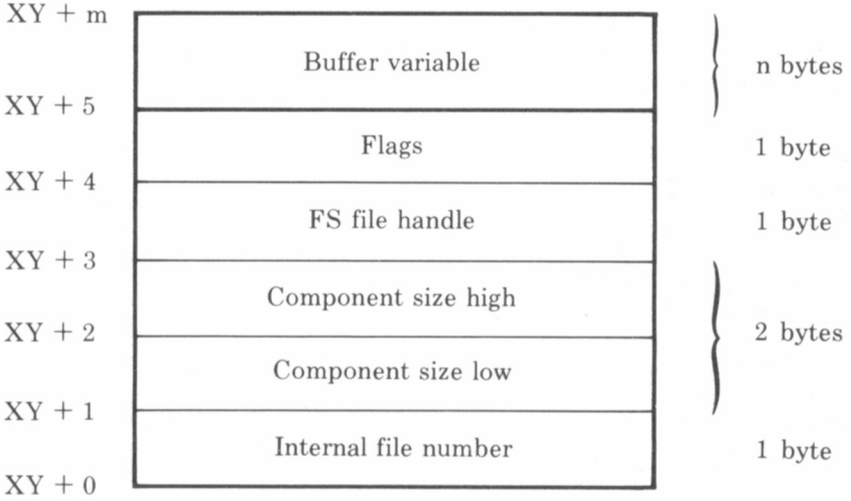
```
res := code1(addr,1,myFile);
```

will pass the address of the file descriptor in XY, and the buffer variable is found five bytes further on, whereas,

```
res := code1(addr,1,myFile^);
```

will pass the address of the buffer variable in XY and the file descriptor is found five bytes lower in memory.

The format of the file descriptor is:



The internal file number is zero for permanent files (whose names appear in the program header), and 1–8 for temporary files. If the value is n, the name of the temporary file is 'pas___m' where m is n−1. Thus the first temporary file has a file number 1 and name 'pas___0'.

The component size is a two-byte value giving the size of each component of the file. Thus a file of integer would have a component size of 4 and a file of packed array[1..100] of boolean would have a component size of 100. This is the size of the buffer variable that follows the file descriptor.

The filing system file handle is the channel number returned by the filing system when the file was opened. Its value depends on the filing system type and how many files are already open.

The flags are stored in a single byte, of which 6 bits are currently used:

bit 0 is unused.

bit 1 is unused.

bit 2 is 1 if a 'get' is pending on the (text) file. This is used to provide interactive input files where a line does not have to be input before text may be output. This is also called 'lazy input'.

bit 3 is 1 if the filing system's EOF flag is set for the (input) file.

bit 4 is 1 if the Pascal eof function is set for the file.

bit 5 is 1 if eoln is true for the (text) file.

bit 6 is 1 if the buffer variable for the file is currently valid.

bit 7 is 1 if the file is 'read', 0 if it is 'write'.

The buffer variable's format is exactly the same as the file's component type, including variations on integer sizes for packing.

8.2.4 Text files

The predefined file type 'text' has special properties. It is similar to 'file of char', but it is possible to read and write entities other than char. In addition to characters, reals and integers may be read from and written to text files, and booleans and strings may be written. The formats are as follows:

Char	: c	where c is a character
String	: ccccc	for as many characters as are in the string
Integer	: iiiiiiiiiii	where i is a digit or leading space
Real	: ddddddddddddd	where d is a digit, '+', '–', 'e' or leading space
Boolean	: 'FALSE' or 'TRUE'	

Thus the types have default field widths of 1, length of the string, 12, 12 and 5 characters for characters, strings, integers, reals and booleans respectively. The total print width and number of decimal digits (for reals) may be altered using the special forms of write (and writeln) as in:

```
write(123 : 4);
write(1.23 : 6 :3);
write(1<>2 : 7);
write(' ' : 40);
write('Number of eggs:' : 34);
```

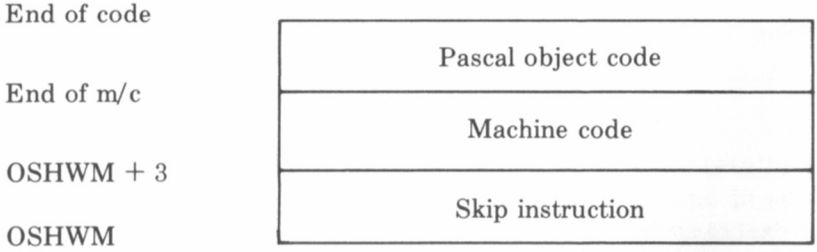
The print widths may be expressions, of course. See *Pascal from BASIC* for more details on print formatting. The maximum print width allowed is 255 and the largest number of decimal places (for reals) is 48.

When reading from text files, only reals, integers and characters may be read. Numbers should be separated by spaces or ends of lines.

9 Using machine code from Pascal

This chapter presents an example of how to incorporate machine code subroutines into a Pascal program. It is easier if you assume that the Pascal object file (and, therefore, the embedded machine code) will always be loaded to the same place, but it is possible with a little effort to make totally relocatable programs (pure Pascal object code is always relocatable). Machine code routines are called using the functions 'code0' and 'code1' which were discussed in section 5.8.

The basic idea when incorporating machine code is to put the code before the Pascal object code, and incorporate an instruction to tell the Pascal interpreter to skip the machine code. It looks like this:



Pascal object programs are always loaded at OSHWM. Since this varies on different machines (depending on the filing system, presence of Second Processor etc), it is obviously important to assemble the machine code to run at the correct address. The skip instruction at the start of the file is a Pascal object code (BL-code) instruction telling the interpreter to branch by a given displacement. It is three bytes long. The first byte (&F1) is the opcode and the next two bytes are the displacement.

9.1 An example routine

As a simple example, we will produce a small routine which shifts a Pascal integer variable by a given number of places to the left. Shifting n times is equivalent to multiplying the number by two to the power of n. This could be done by a 'for' loop in Pascal, but would be quite slow.

The function 'code1' will be used to call the machine code. Remember that this takes three parameters: the address of the routine, the value the accumulator should take on entry, and a Pascal variable. The routine will, therefore, be called by an instruction of the form:

```
dummy:=code1(shiftAddr, count, int);
```

It should be noted that there is no way of telling what type the variable is from within the machine code, so if called incorrectly the routine will quite happily shift, say, the first four bytes of a set variable.

Machine code routines may use zero-page memory locations between &70 and &8F, in the same way as under BASIC.

9.1.1 BASIC II version

The program to perform the shift operation is given below in BBC BASIC II assembler:

```
1000 REM  Example machine code routine for Pascal
1010
1020 REM BL-code skip instruction
1030 jump_code_base = &F1
1040 REM For disc machine
1050 oshwm          = &1900
1060 REM Pointer to variable
1070 var_ptr        = &70
1080 REM Counter for bytes
1090 count          = &72
1100 REM For machine code
1110 DIM code 40
1120
1130 FOR opt=4 TO 6 STEP 2
1140 P%=oshwm
1150 O%=code
1160 [ opt opt
1170 \ Put skip at start of m/c
1180   EQUB jump_code_base
1190 \ Displacement to Pascal entry point
1200   EQUW end_of_mc - oshwm
```

```
1210 .shift
1220 \ Store address of var
1230   stx var_ptr
1240 \ Count in range 0..31
1250   sty var_ptr + 1
1260   and #&1F
1270 \ Count of shifts
1280   sta count
1290 \ Done last one
1300 .shift_loop_count
1310   dec count
1320   bmi end_shift
1330 \ Shift four bytes
1340   ldy #0
1350 \ X is counter for bytes
1360   ldx #4
1370 \ Shift in zero for first byte
1380   clc
1390 \ Get a byte
1400 .shift_loop_y
1410   lda (var_ptr),Y
1420 \ One bit to the left
1430   rol A
1440 \ Store it back
1450   sta (var_ptr),Y
1460 \ Next byte
1470   iny
1480 \ Next bit
1490   dex
1500   bne shift_loop_y
1510   beq shift_loop_count
1520 \ All bits done
1530 .end_shift
1540 \ For disp calculation
1550   rts
1560 .end_of_mc
1570 ]
1580 NEXT opt
1590
1600 PRINT " *SAVE SHIFTMC ";~code" ";
1610 PRINT ;~0%" ";~shift" ";~oshwm
1620 END
```

The important lines are:

1030
This defines the opcode of the BL jump instruction.

1050
This sets the base address and for non-relocatable routines should be altered to suit the circumstances, eg on a Second Processor it should be &800.

1070, 1090
These define the zero-page locations that are used. &70–&8F are available for Pascal users.

1180
This puts the BL-code jump at the start of the machine code as this is where Pascal will start to execute when a program is RUN.

1200
This is the two-byte displacement needed to skip the machine code.

1210 to 1550
This is the program to perform the shift operation.

1560
This is the label marking the end of machine code (and the start of Pascal when the files are combined) used in line 1200.

1600, 1610
These print a line which, if copied, will save the machine code. The third number gives the execution address (where the codel function should jump to) and the last address is OSHWM.

9.1.2 BASIC I version

If you haven't got BASIC II, it is a little more tricky to assemble the machine code at OSHWM (PAGE). The best method is to increase PAGE to, say, &1A00 before the program is loaded and set P% to OSHWM. O% wouldn't be used and line 1080 would be:

```
1080 FOR opt=0 TO 2 STEP 2
```

Notice that the program is relocatable as there are no references to addresses within it, except for branches. This means that it will work at any value of OSHWM. If there were any absolute references (for example, if the program contained any subroutines) then it would only work at the address for which it was assembled.

9.1.3 Using the routine

The next step is to write a Pascal program which uses the routine. A simple one is given below. It simply sets an integer to 1, then shifts it by 0 to 31 places in turn, printing its new value each time. The program is:

```
program shift(input,output);

{ This demonstrates how the machine code routine
at the start of the file is called. It uses
OSBYTE to deduce its address (by finding OSHWM)
and calls the code three bytes above that. If the
machine code were not relocatable, its address
might as well be declared in a constant
declaration, as the program would not work at any
other location. }

const
    osbyte     = &FFF4;     { Operating system
                              call }
    findOshwm  = 131;       { Code to find OSHWM }
    mcOffset   = 3;         { Offset from OSHWM to
                              code }

var
    count      : integer;   { Looping variable }
    mcAddr     : integer;   { Address of machine
                              code }
    oshwm      : integer;   { Current OSHWM value }
    anInteger  : integer;   { A variable accessed by
                              'shift' }
    dummy      : integer;   { A dummy variable for
                              code1 }

begin
 { Find OSHWM - Returned in XY hence the mod and
   div }
 oshwm :=
code0(osbyte,findOshwm,0,0) mod &1000000 div &100;

 mcAddr := oshwm + mcOffset;
```

```
 { Simply print the integer and shift it left, 32
   times}
 for count := 0 to 31 do
   begin
      anInteger := 1;
      dummy := code1(mcAddr,count,anInteger);
      write(~anInteger : 10)
   end;
 writeln
end.
```

The only complex bit is the way that OSHWM is obtained. Remember that code0 and code1 return their integer results in the form &PYXA. The desired bytes (Y and X) are therefore in the middle. The expression:

```
code0(osbyte,find0shwm,0,0) mod &1000000 div &100
```

first masks off the top byte (P) then shifts the remainder one byte to the right so that we are left with YX.

If the program above were saved in the file 'code1', it could be compiled thus:

COMPILE code1 O.code1

We cannot run the object file yet as the machine code needs to be inserted at the start of the file. To do this, another Pascal program is used. This asks for the names of the machine code file, the Pascal object file and the final, composite object file. It creates the final file simply by concatenating the first two files. The three filenames may be typed immediately after the command to run the concatenate program, or may be typed in response to the prompts that the program issues.

```
{$T+}
program  merge(input,output,codeFile,blFile,
                  pascalFile);

{ This asks for the filenames of a machine code
file, a bl-code (Pascal object) file and a final
file. It merges the first two files to create the
last. }

const
   nameLen = 10;
```

```
type
    byte = 0..255;
    name = packed array[1..nameLen] of char;
    binary = packed file of byte;

var
    codeFile : binary;
    blFile   : binary;
    pascalFile : binary;
    codeName : name;
    blName   : name;
    pascalName : name;
    prompt : boolean;


procedure readName(var fileName : name);
var
    i : 1..nameLen;

begin
    for i:=1 to nameLen do
      fileName[i] := ' ';
    while input^ = ' ' do
      get(input);
    i := 1;
    repeat
      read(fileName[i]);
      i := i+1
    until eoln or (fileName[i-1]=' ');
    if eoln then
      readln
end;

procedure  copy(var sourceFile, destFile :
                  binary);
var
    b : byte;
    begin
    while not eof(sourceFile) do
       begin
          read(sourceFile, b);
          write(destFile, b)
       end
end;
```

```
begin { Main Program }

    prompt := eoln;

    if prompt then
      write('Name of machine code file: ');
    readName(codeName);

    if prompt then
      write('Name of object file: ');
    readName(blName);

    if prompt then
      write('Name of final object file: ');
    readName(pascalName);

    reset(codeFile, codeName);
    reset(blFile, blName);
    rewrite(pascalFile, pascalName);

    copy(codeFile, pascalFile);
    copy(blFile, pascalFile);

    writeln('Finished')

end.
```

In fact the program may be used to concatenate any two files into a single, large one. It simply treats all three files as a sequence of bytes.

The composite file produced by the program is a RUNable Pascal object file. In the current example, the replies to the three prompts are:

SHIFTMC<RETURN>
O.code1<RETURN>
fin<RETURN>

Once Pascal is re-entered, the file 'fin' may be RUN (or simply typed as a command since no Pascal command has the same name). The output from the program consists of the powers of two between 0 and 31, printed in hexadecimal in a field width of ten characters.

Appendix A
ISO specification

This appendix contains the full ISO specification for Pascal, which is suitable for use as a reference section for Acornsoft ISO-Pascal.

The specification, BS6192: 1982 is reproduced by permission of the British Standards Institution. Copies of the standard can be obtained from BSI at Linford Wood, Milton Keynes MK14 6LE.

BS 6192 : 1982

UDC 681.3.06 PASCAL : 519.682

Specification for

Computer programming language Pascal

Spécification du langage de programmation de calculateur Pascal

Spezifikation für die Computerprogrammiersprache Pascal

British Standards Institution

Preface

This standard has been prepared under the direction of the Office and Information Standards Committee. The text following this preface has been prepared as an international standard to be published by the International Organization for Standardization (ISO). Following a precedent set by other programming languages (e.g. ISO 1539 'Programming language—FORTRAN') the intention is that the draft international standard, ISO/DIS 7185, and the resulting standard, ISO 7185, will refer to this British Standard for the whole of the technical content.

NOTE. A list of the cooperating organizations involved in the preparation of this standard is given on the back cover.

Terminology and conventions. For ease of reproduction, the actual text produced for the draft international standard, with minor editorial changes, has been used for this British Standard. As a result of using the international text, some terminology and certain conventions are not identical with those used in British Standards; attention is especially drawn to the following.

Wherever 'ISO 7185' appears, it should be read as 'BS 6192'.

Cross-reference

International standard	Corresponding British Standard
ISO 646-1973	BS 4730 : 1974 Specification for the United Kingdom 7-bit data code (Technically equivalent)

Contents

Foreword

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

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

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

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

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

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

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

Editorial note. It is normal convention to use italic type for algebraic quantities. Since the status of such quantities contained in this standard may or may not directly represent true variable quantities, this convention has not been adopted in this standard.

British Standard Specification for

Computer programming language Pascal

0. Introduction

The appendices are included for the convenience of the reader of this standard. They do not form a part of the requirements of this standard.

1. Scope

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

1.2 This standard does not specify:

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

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

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

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

(e) the method for reporting errors or warnings;

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

2. Reference

ISO 646 : The 7-bit coded character set for information processing interchange

3. Definitions

For the purposes of this standard, the following definitions apply.

NOTE. To draw attention to language concepts, some terms are printed in italics on their first mention in this standard.

3.1 **error**. A violation by a program of the requirements of this standard that a processor is permitted to leave undetected.

NOTES

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

2. Processors should attempt the detection of as many errors as possible, and to as complete a degree as possible. Permission to omit detection is provided for implementations in which the detection would be an excessive burden, or which are not of the highest quality.

3.2 **extension**. A modification to clause **6** of this standard that neither invalidates any program complying with the requirements of this standard, as defined by **5.2**, except by prohibiting the use of one or more particular spellings of identifiers, nor alters the status of any implementation-dependent feature or error.

3.3 **implementation-defined**. Possibly differing between processors, but defined for any particular processor.

3.4 **implementation-dependent**. Possibly differing between processors and not necessarily defined for any particular processor.

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

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

4. Definitional conventions

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

Table 1. Metalanguage symbols

Metasymbol	Meaning
=	shall be defined to be
>	shall have as an alternative definition
|	alternatively
.	end of definition
[x]	0 or 1 instance of x
{ x }	0 or more instances of x
(x | y)	grouping: either of x or y
"xyz"	the terminal symbol xyz
meta-identifier	a non-terminal symbol

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

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

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

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

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

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

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

the y *closest-containing* an x: that y which contains an x but does not contain another y containing that x.

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

5. Compliance

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

5.1 Processors. A processor complying with the requirements of this standard shall:

(a) if it complies at level 0, accept all the features of the language specified in

clause 6, except for 6.6.3.6(e), 6.6.3.7 and 6.6.3.8, with the meanings defined in clause 6;

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

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

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

(e) be able to determine whether or not a program violates any requirement of this standard, where such a violation is not designated an error, and report the result of this determination to the user of the processor; in the case where the processor does not examine the whole program, the user shall be notified that the determination is incomplete whenever no violations have been detected in the program text examined;

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

(1) there shall be a statement in an accompanying document that the error is not reported;

(2) the processor shall report during preparation of the program for execution that an occurrence of that error was possible;

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

(4) the processor shall report the error during execution of the program, and terminate execution of the program;

and if any violations that are designated as errors are treated in the manner described in 5.1(f)(1), then a note referencing each such treatment shall appear in a separate section of the accompanying document;

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

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

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

NOTES

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

2. In cases where the compilation is aborted due to some limitation of tables, etc., an incomplete determination of the kind 'No violations were detected, but the examination is incomplete.' will satisfy the requirements of clause 5.1(e). In a similar manner, an interpretive or direct execution processor may report an incomplete determination for a program of which all aspects have not been examined.

A processor that purports to comply, wholly or partially, with the requirements of this

standard shall do so only in the following terms. A *compliance statement* shall be produced by the processor as a consequence of using the processor, or shall be included in accompanying documentation. If the processor complies in all respects with the requirements of this standard the compliance statement shall be:

‹*This processor*› complies with the requirements of level ‹*number*› of ISO 7185.

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

‹*This processor*› complies with the requirements of level ‹*number*› of ISO 7185, with the following exceptions:
‹*followed by a reference to, or a complete list of, the requirements of the standard with which the processor does not comply*›.

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

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

5.2 **Programs**. A program complying with the requirements of this standard shall:

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

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

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

NOTES

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

2. The requirements for compliant programs and compliant processors do not require that the results produced by a compliant program are always the same when processed by a compliant processor. They may be, or they may differ, depending on the program. A simple program to illustrate this is:
program x(output); begin writeln(maxint) end.

6. Requirements

6.1 Lexical tokens

NOTE. The syntax given in this subclause (6.1) describes the formation of lexical tokens from characters and the separation of these tokens, and therefore does not adhere to the same rules as the syntax in the rest of this standard.

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

```
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" |
         "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" |
         "u" | "v" | "w" | "x" | "y" | "z" .
```

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

6.1.2 *Special-symbols*. The special-symbols are tokens having special meanings and shall be used to delimit the syntactic units of the language.

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

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

6.1.3 *Identifiers*. Identifiers may be of any length. All characters of an identifier shall be significant in distinguishing between identifiers. No identifier shall have the same spelling as any word-symbol. Identifiers that are specified to be *required* shall have special significance (see 6.2.2.10 and 6.10).

```
identifier = letter { letter | digit } .
```

Examples:

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

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

```
directive = letter { letter | digit } .
```

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

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

```
signed-number = signed-integer | signed-real .
signed-real = [ sign ] unsigned-real .
signed-integer = [ sign ] unsigned-integer .
unsigned-number = unsigned-integer | unsigned-real .
sign = "+" | "-" .
unsigned-real = unsigned-integer "." fractional-part [ "e" scale-factor ] |
                unsigned-integer "e" scale-factor .
unsigned-integer = digit-sequence .
fractional-part = digit-sequence .
scale-factor = signed-integer .
digit-sequence = digit { digit } .
```

Examples:

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

6.1.6 *Labels*. Labels shall be digit-sequences and shall be distinguished by their apparent integral values and shall be in the closed interval 0 to 9999.

```
label = digit-sequence
```

6.1.7 *Character-strings*. A character-string containing a single string-element shall denote a value of the required char-type (see **6.4.2.2**). A character-string containing more than one string-element shall denote a value of a string-type (see **6.4.3.2**) with the same number of components as the character-string contains string-elements. Each string-character shall denote an implementation-defined value of the required char-type, subject to the restriction that no such value shall be denoted by more than one string-element.

```
character-string = "'" string-element { string-element } "'" .
string-element = apostrophe-image | string-character .
apostrophe-image = "''" .
string-character = one-of-a-set-of-implementation-defined-characters .
```

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

Examples:

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

6.1.8 *Token separators*. The construct

```
"{" any-sequence-of-characters-and-separations-of-lines-not-containing-right-brace "}"
```

shall be a comment if the { does not occur within a character-string or within a comment. The substitution of a space for a comment shall not alter the meaning of a program.

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

6.1.9 *Lexical alternatives*. The representation for lexical tokens and separators given in **6.1.1** to **6.1.8** shall constitute a *reference representation* for these tokens and separators. The reference representation shall be used for program interchange.

To facilitate the use of Pascal on processors that do not support the reference representation, the following alternatives have been defined. All processors that have the required characters in their character set shall provide both the reference

representations and the alternative representations, and the corresponding tokens or separators shall not be distinguished.

The alternative representations for the tokens shall be:

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

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

The alternative forms of comment shall be all forms of comment where one or both of the following substitutions are made:

Delimiting character	Alternative delimiting pair of characters
{	(*
}	*)

NOTES

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

3. If the sequence *(*)* occurs in a comment, it is equivalent to *{}* and marks the end of the comment, because the substitution is only for a delimiting character.

4. See also 1.2(f).

6.2 Blocks, scope and activations

6.2.1 *Block.* A block closest-containing a label-declaration-part in which a label occurs shall closest-contain exactly one statement in which that label occurs. The occurrence of a label in the label-declaration-part of a block shall be its defining-point as a label for the region that is the block.

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

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

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

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

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

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

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

```
statement-part = compound-statement .
```

6.2.2 *Scope*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.3 *Activations*

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

6.2.3.2 The activation of a block shall contain:

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

(b) for each label in a statement having a defining-point in the label-declaration-part of the block, a program-point in the algorithm of the activation of that statement;

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

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

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

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

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

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

(a) the activation containing the procedure or function; *and*

(b) all activations that that containing activation is within.

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

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

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

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

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

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

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

6.4 Type-definitions

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

```
type-definition = identifier "=" type-denoter
type-denoter = type-identifier ¦ new-type .
new-type = new-ordinal-type ¦ new-structured-type ¦ new-pointer-type .
```

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

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

```
simple-type-identifier = type-identifier .
structured-type-identifier = type-identifier .
pointer-type-identifier = type-identifier .
type-identifier = identifier .
```

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

6.4.2 *Simple-types*

6.4.2.1 *General.* A simple-type shall determine an ordered set of values. The values of each ordinal-type shall have integer ordinal numbers. An ordinal-type-identifier shall denote an ordinal-type.

```
simple-type = ordinal-type ¦ real-type-identifier .
ordinal-type = new-ordinal-type ¦ ordinal-type-identifier .
new-ordinal-type = enumerated-type ¦ subrange-type .
ordinal-type-identifier = type-identifier .
real-type-identifier = type-identifier .
```

6.4.2.2 *Required simple-types.* The following types shall exist:

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

(b) *real-type.* The required real-type-identifier *real* shall denote the real-type. The

values shall be an implementation-defined subset of the real numbers denoted as specified in **6.1.5** by signed-real.

(c) *Boolean-type*. The required ordinal-type-identifier *Boolean* shall denote the Boolean-type. The values shall be the enumeration of truth values denoted by the required constant-identifiers *false* and *true*, such that false is the predecessor of true. The ordinal numbers of the truth values denoted by *false* and *true* shall be the integer values 0 and 1 respectively.

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

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

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

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

(4) The ordering relationship between any two character values shall be the same as between their ordinal numbers.

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

6.4.2.3 *Enumerated-types*. An enumerated-type shall determine an ordered set of values by enumeration of the identifiers that denote those values. The ordering of these values shall be determined by the sequence in which their identifiers are enumerated, i.e. if x precedes y then x is less than y. The ordinal number of a value that is of an enumerated-type shall be determined by mapping all the values of the type on to consecutive non-negative values of integer-type starting from zero. The mapping shall be order preserving.

```
enumerated-type = "(" identifier-list ")" .
identifier-list = identifier { "," identifier } .
```

The occurrence of an identifier in the identifier-list of an enumerated-type shall constitute its defining-point as a constant-identifier for the region that is the block closest-containing the enumerated-type.

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

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

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

Examples:

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

6.4.3 *Structured-types*

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

```
structured-type = new-structured-type | structured-type-identifier .
new-structured-type = [ "packed" ] unpacked-structured-type .
unpacked-structured-type = array-type | record-type | set-type | file-type .
```

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

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

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

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

```
array-type = "array" "[" index-type { "," index-type } "]" "of" component-type .
index-type = ordinal-type .
component-type = type-denoter .
```

Examples 1:

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

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

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

Example 2:

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

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

Example 3:

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

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

(v[m], ... ,v[n]).

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

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

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

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

6.4.3.3 *Record-types*. The structure and values of a record-type shall be the structure and values of the field-list of the record-type.

```
record-type = "record" field-list "end" .
field-list = [ ( fixed-part [ ";" variant-part ] | variant-part ) [ ";" ] ] .
fixed-part = record-section { ";" record-section } .
record-section = identifier-list ":" type-denoter .
variant-part = "case" variant-selector "of" variant { ";" variant } .
variant-selector = [ tag-field ":" ] tag-type .
tag-field = identifier .
variant = case-constant-list ":" "(" field-list ")" .
tag-type = ordinal-type-identifier .
case-constant-list = case-constant { "," case-constant } .
case-constant = constant .
```

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

The occurrence of an identifier in the identifier-list of a record-section of a fixed-part of a field-list shall constitute its defining-point as a field-identifier for the region that is the record-type closest-containing the field-list, and shall associate the field-identifier with a distinct component, which shall be designated a *field*, of the record-type and of the field-list. That component shall have the type denoted by the type-denoter of the record-section.

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

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

(V1, V2, ..., Vm).

NOTE 1. If the type of the i-th component has Fi values, then the cardinality of the set of values of the field-list shall be (F1 * F2 * ... * Fm).

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

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

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

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

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

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

(k, Xk)

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

NOTES

2. If there are n values specified by the selector-type, and if the field-list of the variant associated with the i-th value has Ti values, then the cardinality of the set of values of the variant-part is (T1 + T2 + ... + Tn). There is no component of a value of a variant-part corresponding to any non-active variant of the variant-part.

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

Examples:

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

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

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

6.4.3.4 *Set-types*. A set-type shall determine the set of values that is structured as the powerset of the base-type of the set-type. Thus each value of a set-type shall be a set whose members shall be unique values of the base-type.

```
set-type = "set" "of" base-type .
base-type = ordinal-type .
```

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

Examples:

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

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

For every ordinal-type S, there exists an unpacked set type designated the *unpacked canonical set-of-T type* and there exists a packed set type designated the *packed canonical set-of-T type*. If S is a subrange-type then T is the host type of S; otherwise T is S. Each value of the type set of S is also a value of the unpacked canonical set-of-T type, and each value of the type packed set of S is also a value of the packed canonical set-of-T type.

6.4.3.5 *File-types*

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

file-type = "file" "of" component-type .

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

Examples:

file of real
file of vector

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

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

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

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

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

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

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

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

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

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

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

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

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

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

There shall be a file-type that is denoted by the required structured-type-identifier *text*. The structure of the type denoted by *text* shall define an additional sequence-type whose values shall be designated *lines*. A line shall be a sequence cs~S(e), where cs is a sequence of components having the char-type, and e represents a special component value, which shall be designated an *end-of-line*, and which shall be indistinguishable from the char value space except by the required function *eoln* (see **6.6.6.5**) and by the required procedures *reset* (see **6.6.5.2**), *writeln* (see **6.9.4**), and *page* (see **6.9.5**). If l is a line then no component of l other than l.last shall be an end-of-line. This definition shall not be construed to determine the underlying representation, if any, of an end-of-line component used by a processor.

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

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

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

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

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

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

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

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

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

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

```
pointer-type = new-pointer-type ¦ pointer-type-identifier .
new-pointer-type = "↑" domain-type .
domain-type = type-identifier .
```

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

6.4.5 *Compatible types*. Types T1 and T2 shall be designated *compatible* if any of the following four statements is true.

(a) T1 and T2 are the same type.

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

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

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

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

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

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

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

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

(e) T1 and T2 are compatible string-types.

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

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

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

6.4.7 *Example of a type-definition-part*

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

```
persondetails = record
                  name, firstname : charsequence;
                  age : integer;
                  married : Boolean;
                  father, child, sibling : person;
                  case s : sex of
                    male :
                      (enlisted, bearded : Boolean);
                    female :
                      (mother, programmer : Boolean)
                end;
FileOfInteger = file of integer;
```

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

6.5 Declarations and denotations of variables

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

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

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

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

Example of a variable-declaration-part:

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

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

NOTE. Variables occurring in examples in the remainder of this standard should be assumed to have been declared as specified in 6.5.1.

6.5.2 *Entire-variables*

```
entire-variable = variable-identifier .
variable-identifier = identifier .
```

6.5.3 *Component-variables*

6.5.3.1 *General.* A component of a variable shall be a variable. A component-variable shall denote a component of a variable. A reference, or access to a component of a variable shall constitute a reference, or access, respectively, to the variable. The value, if any, of the component of a variable shall be the same component of the value, if any, of the variable.

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

6.5.3.2 *Indexed-variables.* A component of a variable possessing an array-type shall be denoted by an indexed-variable.

```
indexed-variable = array-variable "[" index-expression { "," index-expression } "]" .
array-variable = variable-access .
index-expression = expression .
```

An array-variable shall be a variable-access that denotes a variable possessing an array-type. For an indexed-variable closest-containing a single index-expression, the value of the index-expression shall be assignment-compatible with the index-type of the array-type. The component denoted by the indexed-variable shall be the component that corresponds to the value of the index-expression by the mapping of the type possessed by the array-variable (see **6.4.3.2**).

Examples 1:

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

If the array-variable is itself an indexed-variable an abbreviation may be used. In the abbreviated form, a single comma shall replace the sequence *] [* that occurs in the full form. The abbreviated form and the full form shall be equivalent.

The order of evaluation of the index-expressions of an indexed-variable shall be implementation-dependent.

Examples 2:

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

NOTE. These two examples denote the same component variable.

6.5.3.3 *Field-designators.* A field-designator either shall denote that component of the record-variable of the field-designator associated with the field-identifier of the field-specifier of the field-designator, by the record-type possessed by the record-variable; or shall denote the variable denoted by the field-designator-identifier (see **6.8.3.10**) of the field-designator. A record-variable shall be a variable-access that denotes a variable possessing a record-type.

The occurrence of a record-variable in a field-designator shall constitute the

defining-point of the field-identifiers associated with components of the record-type possessed by the record-variable, for the region that is the field-specifier of the field-designator.

```
field-designator = record-variable "." field-specifier | field-designator-identifier .
record-variable = variable-access .
field-specifier = field-identifier .
field-identifier = identifier .
```

Examples:

```
p2↑.mother
coord.theta
```

An access to a component of a variant of a variant-part, where the selector of the variant-part is not a field, shall attribute to the selector that value specified by its type associated with the variant. It shall be an error unless a variant is active for the entirety of each reference and access to each component of the variant.

When a variant becomes not active, all of its components shall become totally-undefined.

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

6.5.4 *Identified-variables*. An identified-variable shall denote the variable (if any) identified by the value of the pointer-variable of the identified-variable (see **6.4.4** and **6.6.5.3**).

```
identified-variable = pointer-variable "↑" .
pointer-variable = variable-access .
```

A variable created by the required procedure *new* (see **6.6.5.3**) shall be accessible until the termination of the activation of the program-block or until the variable is made inaccessible (see the required procedure *dispose*, **6.6.5.3**).

NOTE. The accessibility of the variable also depends on the existence of a pointer-variable that has attributed to it the corresponding identifying value.

A pointer-variable shall be a variable-access that denotes a variable possessing a pointer-type. It shall be an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined. It shall be an error to remove from its pointer-type the identifying-value of an identified variable (see **6.6.5.3**) when a reference to the identified variable exists.

Examples:

```
p1↑
p1↑.father↑
p1↑.sibling↑.father↑
```

6.5.5 *Buffer-variables*. A file-variable shall be a variable-access that denotes a variable possessing a file-type. A buffer-variable shall denote a variable associated with the variable denoted by the file-variable of the buffer-variable. A buffer-variable associated with a textfile shall possess the char-type; otherwise, a buffer-variable shall possess the component-type of the file-type possessed by the file-variable of the buffer-variable.

```
buffer-variable = file-variable "↑" .
file-variable = variable-access .
```

Examples:

```
input↑
pooltape[2]↑
```

It shall be an error to alter the value of a file-variable f when a reference to the buffer-variable f↑ exists. A reference or access to a buffer-variable shall constitute a reference or access, respectively, to the associated file-variable.

6.6 Procedure and function declarations

6.6.1 *Procedure-declarations*

```
procedure-declaration = procedure-heading ";" directive |
                        procedure-identification ";" procedure-block |
                        procedure-heading ";" procedure-block .
procedure-heading = "procedure" identifier [ formal-parameter-list ] .
procedure-identification = "procedure" procedure-identifier .
procedure-identifier = identifier .
procedure-block = block .
```

The occurrence of a formal-parameter-list in a procedure-heading of a procedure-declaration shall define the formal parameters of the procedure-block, if any, associated with the identifier of the procedure-heading to be those of the formal-parameter-list.

The occurrence of an identifier in the procedure-heading of a procedure-declaration shall constitute its defining-point as a procedure-identifier for the region that is the block closest-containing the procedure-declaration.

Each identifier having a defining-point as a procedure-identifier in a procedure-heading of a procedure-declaration closest-containing the directive *forward* shall have exactly one of its applied occurrences in a procedure-identification of a procedure-declaration, and that shall be closest-contained by the procedure-and-function-declaration-part closest-containing the procedure-heading.

The occurrence of a procedure-block in a procedure-declaration shall associate the procedure-block with the identifier in the procedure-heading, or with the procedure-identifier in the procedure-identification, of the procedure-declaration.

Examples of a procedure-and-function-declaration-part:

Example 1:

NOTE. This example is not for level 0.

```
procedure AddVectors (var A, B, C : array [low..high : natural] of real);
var
  i : natural;
begin
  for i := low to high do A[i] := B[i] + C[i]
end { of AddVectors };
```

Example 2:

```
procedure readinteger (var f : text; var x : integer);
var
   i : natural;
begin
   while f↑ = ' ' do get(f);
   {The file buffer contains the first non-space char}
   i := 0;
   while f↑ in ['0'..'9'] do begin
      i := (10 * i) + (ord(f↑) - ord('0'));
      get(f)
   end;
   {The file buffer contains a non-digit}
   x := i
   {Of course if there are no digits, x is zero}
end;

procedure bisect (function f(x : real) : real; a, b : real; var result : real);
{This procedure attempts to find a zero of f(x) in (a,b) by the method of bisection. It
is assumed that the procedure is called with suitable values of a and b such that
   (f(a) < 0) and (f(b) > 0)
The estimate is returned in the last parameter.}
const
   Eps = 1e-10;
var
   midpoint : real;
begin
   {The invariant P is true by calling assumption}
   midpoint := a;
   while abs(a - b) > Eps * abs(a) do begin
      midpoint := (a + b) / 2;
      if f(midpoint) < 0 then a := midpoint
      else b := midpoint
      {Which re-establishes the invariant:
         P = (f(a) < 0) and (f(b) > 0)
      and reduces the interval (a,b) provided that the value of midpoint is
      distinct from both a and b.}
   end;
   {P together with the loop exit condition assures that a zero is contained in
   a small sub-interval. Return the midpoint as the zero.}
   result := midpoint
end;
```

```
procedure PrepareForAppending (var f : FileOfInteger);
{This procedure takes a file in an arbitrary state and sets it up in a
condition for appending data to its end. Simpler conditioning is only
possible if assumptions are made about the initial state of the file.}
var
  LocalCopy : FileOfInteger;

  procedure CopyFiles (var from, into : FileOfInteger);
  begin
    reset(from); rewrite(into);
    while not eof(from) do begin
      into↑ := from↑;
      put(into); get(from)
    end
  end { of CopyFiles };

begin { of body of PrepareForAppending }
  CopyFiles(f, LocalCopy);
  CopyFiles(LocalCopy, f)
end { of PrepareForAppending };
```

6.6.2 *Function-declarations*

```
function-declaration = function-heading ";" directive |
                       function-identification ";" function-block |
                       function-heading ";" function-block .
function-heading = "function" identifier [ formal-parameter-list ] ":" result-type
function-identification = "function" function-identifier .
function-identifier = identifier .
result-type = simple-type-identifier | pointer-type-identifier .
function-block = block .
```

The occurrence of a formal-parameter-list in a function-heading of a function-declaration shall define the formal parameters of the function-block, if any, associated with the identifier of the function-heading to be those of the formal-parameter-list. The function-block shall contain at least one assignment-statement such that the function-identifier of the assignment-statement is associated with the block (see **6.8.2.2**).

The occurrence of an identifier in the function-heading of a function-declaration shall constitute its defining-point as a function-identifier associated with the result type denoted by the result-type for the region that is the block closest-containing the function-declaration.

Each identifier having a defining-point as a function-identifier in the function-heading of a function-declaration closest-containing the directive *forward* shall have exactly one of its applied occurrences in a function-identification of a function-declaration, and that shall be closest-contained by the procedure-and-function-declaration-part closest-containing the function-heading.

The occurrence of a function-block in a function-declaration shall associate the function-block with the identifier in the function-heading, or with the function-identifier in the function-identification, of the function-declaration; the block of the function-block shall be associated with the result type that is associated with the identifier or function-identifier, respectively.

Example of a procedure-and-function-declaration-part:

```
function Sqrt (x : real) : real;
{This function computes the square root of x (x > 0) using Newton's method.}
var
   old, estimate : real;
begin
   estimate := x;
   repeat
      old := estimate;
      estimate := (old + x / old) * 0.5;
   until abs(estimate - old) < Eps * estimate;
   {Eps being a global constant}
   Sqrt := estimate
end { of Sqrt };

function max (a : vector) : real;
{This function finds the largest component of the value of a.}
var
   largestsofar : real;
   fence : indextype;
begin
   largestsofar := a[1];
   {Establishes largestsofar = max(a[1])}
   for fence := 2 to limit do begin
      if largestsofar < a[fence] then largestsofar := a[fence]
      {Re-establishing largestsofar = max(a[1], ... ,a[fence])}
   end;
   {So now largestsofar = max(a[1], ... ,a[limit])}
   max := largestsofar
end { of max };


function GCD (m, n : natural) : natural;
begin
   if n=0 then GCD := m else GCD := GCD(n, m mod n);
end;
```

```
{The following two functions analyse a parenthesized expression and convert it to an
internal form. They are declared forward since they are mutually recursive, i.e. they
call each other.}

function ReadExpression : formula;
  forward;
function ReadOperand : formula;
  forward;

function ReadExpression; {See forward declaration of heading.}
  var
    this : formula;
    op : operation;
  begin
  this := ReadOperand;
  while IsOperator(nextsym) do
    begin
    op := ReadOperator;
    this := MakeFormula(this, op, ReadOperand);
    end;
  ReadExpression := this
  end;

function ReadOperand; {See forward declaration of heading.}
  begin
  if IsOpenParenthesis(nextsym) then
    begin
    SkipSymbol;
    ReadOperand := ReadExpression;
    {nextsym should be a close-parenthesis}
    SkipSymbol
    end
  else ReadOperand := ReadElement
  end;
```

6.6.3 *Parameters*

6.6.3.1 *General.* The identifier-list in a value-parameter-specification shall be a list of value parameters. The identifier-list in a variable-parameter-specification shall be a list of variable parameters.

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

formal-parameter-section > value-parameter-specification |
                           variable-parameter-specification |
                           procedural-parameter-specification |
                           functional-parameter-specification .
```

NOTE 1. There is also a syntax rule for formal-parameter-section in 6.6.3.7

```
value-parameter-specification = identifier-list ":" type-identifier .
variable-parameter-specification = "var" identifier-list ":" type-identifier .
procedural-parameter-specification = procedure-heading .
functional-parameter-specification = function-heading .
```

An identifier defined to be a parameter-identifier for the region that is the formal-parameter-list of a procedure-heading shall be designated a formal parameter of the block of the procedure-block, if any, associated with the identifier of the

procedure-heading. An identifier defined to be a parameter-identifier for the region that is the formal-parameter-list of a function-heading shall be designated a formal parameter of the block of the function-block, if any, associated with the identifier of the function-heading.

The occurrence of an identifier in the identifier-list of a value-parameter-specification or a variable-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal parameter.

The occurrence of the identifier of a procedure-heading in a procedural-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated procedure-identifier for the region that is the block, if any, of which it is a formal parameter.

The occurrence of the identifier of a function-heading in a functional-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated function-identifier for the region that is the block, if any, of which it is a formal parameter.

NOTE 2 If the formal-parameter-list is contained in a procedural-parameter-specification or a functional-parameter-specification, there is no corresponding procedure-block or function-block.

6.6.3.2 *Value parameters.* The formal parameter and its associated variable-identifier shall denote the same variable. The formal parameter shall possess the type denoted by the type-identifier of the value-parameter-specification. The type possessed by a formal parameter shall be one that is permitted as the component-type of a file-type. The actual-parameter (see **6.7.3** and **6.8.2.3**) shall be an expression whose value is assignment-compatible with the type possessed by the formal parameter. The current value of the expression shall be attributed upon activation of the block to the variable that is denoted by the formal parameter.

6.6.3.3 *Variable parameters.* The actual-parameter shall be a variable-access. The type possessed by the actual-parameters shall be the same as that denoted by the type-identifier of the variable-parameter-specification, and the formal parameters shall also possess that type. The actual-parameter shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall denote the referenced variable during the activation.

An actual variable parameter shall not denote a field that is the selector of a variant-part. An actual variable parameter shall not denote a component of a variable where that variable possesses a type that is designated packed.

6.6.3.4 *Procedural parameters.* The actual-parameter (see **6.7.3** and **6.8.2.3**) shall be a procedure-identifier that has a defining-point contained by the program-block. The procedure denoted by the actual-parameter and the procedure denoted by the formal parameter shall have congruous formal-parameter-lists (see **6.6.3.6**) if either has a formal-parameter-list. The formal parameter and its associated procedure-identifier shall denote the actual parameter during the entire activation of the block.

6.6.3.5 *Functional parameters.* The actual-parameter (see **6.7.3** and **6.8.2.3**) shall be a function-identifier that has a defining-point contained by the program-block. The function denoted by the actual-parameter and the function denoted by the formal parameter shall have the same result-type and shall have congruous formal-parameter-lists (see **6.6.3.6**) if either has a formal-parameter-list. The formal

parameter and its associated function-identifier shall denote the actual parameter during the entire activation of the block.

6.6.3.6 *Parameter list congruity.* Two formal-parameter-lists shall be congruous if they contain the same number of formal-parameter-sections and if the formal-parameter-sections in corresponding positions match. Two formal-parameter-sections shall match if any of the following statements is true.

(a) They are both value-parameter-specifications containing the same number of parameters and the type-identifier in each value-parameter-specification denotes the same type.

(b) They are both variable-parameter-specifications containing the same number of parameters and the type-identifier in each variable-parameter-specification denotes the same type.

(c) They are both procedural-parameter-specifications and the formal-parameter-lists of the procedure-headings thereof are congruous.

(d) They are both functional-parameter-specifications, the formal-parameter-lists of the function-headings thereof are congruous, and the type-identifiers of the result-types of the function-headings thereof denote the same type.

(e) They are either both value-conformant-array-specifications or both variable-conformant-array-specifications; and in both cases the conformant-array-parameter-specifications contain the same number of parameters and equivalent conformant-array-schemas. Two conformant-array-schemas shall be equivalent if all of the following four statements are true.

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

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

(3) Either the (component) conformant-array-schemas of the conformant-array-schemas are equivalent or the type-identifiers of the conformant-array-schemas denote the same type.

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

NOTES

1. The abbreviated conformant-array-schema and its corresponding full form are equivalent (see 6.6.3.7).

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

6.6.3.7 *Conformant array parameters*

NOTE. For the status of this clause see 5.1(a), 5.1(b), 5.1(c), 5.2(a) and 5.2(b).

6.6.3.7.1 *General.* The occurrence of an identifier in the identifier-list contained by a conformant-array-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal parameter. A variable-identifier so defined shall be designated a *conformant-array-parameter*.

The occurrence of an identifier in an index-type-specification shall constitute its defining-point as a bound-identifier for the region that is the formal-parameter-list

closest-containing it and for the region that is the block, if any, whose formal parameters are specified by that formal-parameter-list.

```
formal-parameter-section > conformant-array-parameter-specification .
conformant-array-parameter-specification = variable-conformant-array-specification |
                                           value-conformant-array-specification .
value-conformant-array-specification = identifier-list ":" conformant-array-schema .
variable-conformant-array-specification = "var" identifier-list ":"
                                          conformant-array-schema .
conformant-array-schema = packed-conformant-array-schema |
                          unpacked-conformant-array-schema .
packed-conformant-array-schema = "packed" "array" "[" index-type-specification "]"
                                 "of" type-identifier .
unpacked-conformant-array-schema = "array" "[" index-type-specification
                                   { ";" index-type-specification } "]" "of"
                                   ( type-identifier |
                                   conformant-array-schema ) .
index-type-specification = identifier ".." identifier ":" ordinal-type-identifier .
factor > bound-identifier
bound-identifier = identifier
```

NOTE 1. There is also a syntax rule for formal-parameter-section in **6.6.3.1**. There is also a syntax rule for factor in **6.7.1**.

If a conformant-array-schema closest-contains a conformant-array-schema, then an abbreviated form of definition may be used. In the abbreviated form, a single semicolon shall replace the sequence *] of array [* that occurs in the full form. The abbreviated form and the full form shall be equivalent.

Examples:

```
array [u..v : T1] of array [j..k : T2] of T3
array [u..v : T1; j..k : T2] of T3
```

During the entire activation of the block, applied occurrences of the first identifier of an index-type-specification shall denote the smallest value specified by the corresponding index-type (see **6.6.3.8**) possessed by each actual-parameter, and applied occurrences of the second identifier of the index-type-specification shall denote the largest value specified by that index-type.

NOTE 2. The object denoted by a bound-identifier is neither a constant nor a variable.

The actual-parameters (see **6.7.3** and **6.8.2.3**) corresponding to formal parameters that occur in a single conformant-array-parameter-specification shall all possess the same type. The type possessed by the actual-parameters shall be conformable (see **6.6.3.8**) with the conformant-array-schema, and the formal parameters shall possess an array-type that shall be distinct from any other type, and which shall have a component-type that shall be the fixed-component-type of the conformant-array-parameters defined in the conformant-array-parameter-specification and which shall have the index-types of the type possessed by the actual-parameters that correspond (see **6.6.3.8**) to the index-type-specifications contained by the conformant-array-schema contained by the conformant-array-parameter-specification. The type denoted by the type-identifier contained by the conformant-array-schema contained by a conformant-array-parameter-specification shall be designated the *fixed-component-type* of the conformant-array-parameters defined by that conformant-array-parameter-specification.

NOTE 3. The type possessed by the formal parameter cannot be a string-type (see **6.4.3.2**) because it is not denoted by an array-type.

6.6.3.7.2 *Value conformant arrays.* The identifier-list in a value-conformant-array-specification shall be a list of value conformant arrays. The actual-parameter shall be an expression. The value of the expression shall be attributed before activation of the block to an auxiliary variable that the program does not otherwise contain. The type possessed by this variable shall be the same as that possessed by the expression. This variable shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall represent the referenced variable during the activation. The fixed-component-type of a value conformant array shall be one that is permitted as the component-type of a file-type.

If the actual-parameter contains an occurrence of a conformant-array-parameter, then for each occurrence of the conformant-array-parameter contained by the actual-parameter, *either*:

(a) the occurrence of the conformant-array-parameter shall be contained by a function-designator contained by the actual-parameter; *or*

(b) the occurrence of the conformant-array-parameter shall be contained by an indexed-variable contained by the actual-parameter, such that the type possessed by that indexed-variable is the fixed-component-type of the conformant-array-parameter.

NOTE. This ensures that the type possessed by the expression and the auxiliary variable will always be known and that, as a consequence, the activation record of a procedure can be of a fixed size.

6.6.3.7.3 *Variable conformant arrays.* The identifier-list in a variable-conformant-array-specification shall be a list of variable conformant arrays. The actual-parameter shall be a variable-access. The actual-parameter shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall represent the referenced variable during the activation.

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

6.6.3.8 *Conformability*

NOTE 1. For the status of this clause see **5.1(a)**, **5.1(b)**, **5.1(c)**, **5.2(a)** and **5.2(b)**.

Given a type denoted by an array-type closest-containing a single index-type, and a conformant-array-schema closest-containing a single index-type-specification, then the index-type and the index-type-specification shall be designated as corresponding. Given two conformant-array-schemas closest-containing a single index-type-specification, then the two index-type-specifications shall be designated as corresponding. Let T1 be an array-type with a single index-type and let T2 be the type denoted by the ordinal-type-identifier of the index-type-specification of a conformant-array-schema closest-containing a single index-type-specification, then T1 shall be conformable with the conformant-array-schema if all the following four statements are true.

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

(b) The smallest and largest values specified by the index-type of T1 lie within the closed interval specified by T2.

(c) The component-type of T1 denotes the same type as that denoted by the type-identifier of the conformant-array-schema, or is conformable to the

conformant-array-schema in the conformant-array-schema.

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

NOTE 2 The abbreviated and full forms of a conformant-array-schema are equivalent (see 6.6.3.7). The abbreviated and full forms of an array-type are equivalent (see 6.4.3.2).

It shall be an error if the smallest or largest value specified by the index-type of T1 lies outside the closed interval specified by T2.

6.6.4 *Required procedures and functions*

6.6.4.1 *General.* The required procedure-identifiers and function-identifiers and the corresponding required procedures and functions shall be as specified in 6.6.5 and 6.6.6 respectively.

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

6.6.5 *Required procedures*

6.6.5.1 *General.* The required procedures shall be file handling procedures, dynamic allocation procedures and transfer procedures.

6.6.5.2 *File handling procedures.* Except for the application of *rewrite* or *reset* to the program parameters denoted by *input* or *output*, the effects of applying each of the file handling procedures *rewrite*, *put*, *reset* and *get* to a file-variable f shall be defined by pre-assertions and post-assertions about f, its components f.L, f.R, and f.M, and about the associated buffer-variable f↑. The use of the variable f0 within an assertion shall be considered to represent the state or value, as appropriate, of f prior to the operation, while f (within an assertion) shall denote the variable after the operation, and similarly for f0↑ and f↑.

It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the defined operation. It shall be an error if any variable explicitly denoted in an assertion of equality is undefined. The post-assertion shall hold prior to the next subsequent access to the file, its components, or its associated buffer-variable. The post-assertions imply corresponding activities on the external entities, if any, to which the file-variables are bound. These activities, and the point at which they are actually performed, shall be implementation-defined.

rewrite(f) pre-assertion: true.

post-assertion: (f.L = f.R = S0) and (f.M = Generation) and (f↑ is totally-undefined).

put(f) pre-assertion: (f0.M = Generation) and (f0.L is not undefined) and (f0.R = S0) and (f0↑ is not undefined).

post-assertion: (f.M = Generation) and (f.L = (f0.L~S(f0↑))) and (f.R = S0) and (f↑ is totally-undefined).

reset(f) pre-assertion: The components f0.L and f0.R are not undefined.

post-assertion: (f.L = S0) and (f.R = (f0.L~f0.R~X)) and (f.M = Inspection) and (if f.R = S0 then (f↑ is totally-undefined) else (f↑ = f.R.first)),

where, if f possesses the type denoted by the required structured-type-identifier *text* and if f0.L^f0.R is not empty and if (f0.L^f0.R).last is not designated an end-of-line, then X shall be a sequence having an end-of-line component as its only component; otherwise X = S0.

get(f) pre-assertion: (f0.M = Inspection) and
(neither f0.L nor f0.R are undefined) and (f0.R ◇ S0).

post-assertion: (f.M = Inspection) and (f.L = (f0.L^S(f0.R.first))) and
(f.R = f0.R.rest) and (if f.R = S0 then
(f↑ is totally-undefined) else (f↑ = f.R.first)).

When the file-variable f possesses a type other than that denoted by *text*, the required procedures *read* and *write* shall be defined as follows.

read Read(f,v1,...,vn) where v1...vn denote variable-accesses shall be equivalent to

```
begin read(f,v1); ... ; read(f,vn) end
```

Read(f,v) where v denotes a variable-access shall be equivalent to

```
begin v := f↑; get(f) end
```

NOTE 1. The variable-access is not a variable parameter. Consequently it may be a component of a packed structure and the value of the buffer-variable need only be assignment-compatible with it.

write Write(f,e1,...,en), where e1...en denote expressions shall be equivalent to

```
begin write(f,e1); ... ; write(f,en) end
```

Write(f,e), where e denotes an expression shall be equivalent to

```
begin f↑ := e; put(f) end
```

NOTES.

2. The required procedures *read*, *write*, *readln*, *writeln* and *page*, as applied to textfiles, are described in 6.9.

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

6.6.5.3 *Dynamic allocation procedures*

new(p)
shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the type that is the domain-type of the pointer-type possessed by p.

new(p,c1,...,cn)
shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the record-type that is the domain-type of the pointer-type possessed by p and shall have nested variants that correspond to the case-constants c1,...,cn. The case-constants shall be listed in order of increasing nesting of the variant-parts. Any variant not specified shall be at a deeper level of nesting than that specified by cn. It shall be an error if a

variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

dispose(q)
shall remove the identifying-value denoted by the expression q from the pointer-type of q. It shall be an error if the identifying-value had been created using the form *new(p,c1,...,cn)*.

dispose(q,k1,...,km)
shall remove the identifying-value denoted by the expression q from the pointer-type of q. The case-constants k1,...,km shall be listed in order of increasing nesting of the variant-parts. It shall be an error if the variable had been created using the form *new(p,c1,...,cn)* and m is not equal to n. It shall be an error if the variants in the variable identified by the pointer-value of q are different from those specified by the case-constants k1,...,km.

NOTE. The removal of an identifying-value from the pointer-type to which it belongs renders the identified variable inaccessible (see 6.5.4) and makes undefined all variables and functions that have that value attributed (see 6.6.3.2 and 6.8.2.2).

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

It shall be an error if a variable created using the second form of new is accessed by the identified-variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

6.6.5.4 *Transfer procedures.* Let a be a variable possessing a type that can be denoted by
array [s1] of T,
let z be a variable possessing a type that can be denoted by
packed array [s2] of T,
and u and v be the smallest and largest values of the type s2, then the statement *pack(a,i,z)* shall be equivalent to

```
begin
k := i;
for j := u to v do
   begin
   z[j] := a[k];
   if j <> v then k := succ(k)
   end
end
```

and the statement *unpack(z,a,i)* shall be equivalent to

```
begin
k := i;
for j := u to v do
   begin
   a[k] := z[j];
   if j <> v then k := succ(k)
   end
end
```

where j and k denote auxiliary variables that the program does not otherwise contain. The type possessed by j shall be s2, the type possessed by k shall be s1, and i shall be an expression whose value shall be assignment-compatible with s1.

6.6.6 *Required functions*

6.6.6.1 *General.* The required functions shall be arithmetic functions, transfer functions, ordinal functions and Boolean functions.

6.6.6.2 *Arithmetic functions.* For the following arithmetic functions, the expression x shall be either of real-type or integer-type. For the functions *abs* and *sqr*, the type of the result shall be the same as the type of the parameter, x. For the remaining arithmetic functions, the result shall always be of real-type.

Function	Result
abs(x)	shall compute the absolute value of x.
sqr(x)	shall compute the square of x. It shall be an error if such a value does not exist.
sin(x)	shall compute the sine of x, where x is in radians.
cos(x)	shall compute the cosine of x, where x is in radians.
exp(x)	shall compute the value of the base of natural logarithms raised to the power x.
ln(x)	shall compute the natural logarithm of x, if x is greater than zero. It shall be an error if x is not greater than zero.
sqrt(x)	shall compute the non-negative square root of x, if x is not negative. It shall be an error if x is negative.
arctan(x)	shall compute the principal value, in radians, of the arctangent of x.

6.6.6.3 *Transfer functions*

trunc(x)

From the expression x that shall be of real-type, this function shall return a result of integer-type. The value of trunc(x) shall be such that if x is positive or zero then $0 \leqslant x-\text{trunc}(x) < 1$; otherwise $-1 < x-\text{trunc}(x) \leqslant 0$. It shall be an error if such a value does not exist.

Examples:

trunc(3.5) yields 3
trunc(−3.5) yields −3

round(x)

From the expression x that shall be of real-type, this function shall return a result of integer-type. If x is positive or zero, round(x) shall be equivalent to trunc(x+0.5), otherwise round(x) shall be equivalent to trunc(x−0.5). It shall be an error if such a value does not exist.

Examples:

round(3.5) yields 4
round(−3.5) yields −4

6.6.6.4 *Ordinal functions*

ord(x)

From the expression x that shall be of an ordinal-type, this function shall return a result of integer-type that shall be the ordinal number (see **6.4.2.2** and **6.4.2.3**) of the value of the expression x.

chr(x)

From the expression x that shall be of integer-type, this function shall return a result of char-type that shall be the value whose ordinal number is equal to the value of the expression x if such a character value exists. It shall be an error if

such a character value does not exist. For any value, ch, of char-type, it shall be true that:

chr(ord(ch)) = ch

succ(x)

From the expression x that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression (see **6.7.1**). The function shall yield a value whose ordinal number is one greater than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

pred(x)

From the expression x that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression (see **6.7.1**). The function shall yield a value whose ordinal number is one less than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

6.6.6.5 *Boolean functions*

odd(x)

From the expression x that shall be of integer-type, this function shall be equivalent to the expression

(abs(x) mod 2 = 1).

eof(f)

The parameter f shall be a file-variable; if the actual-parameter-list is omitted, the function shall be applied to the required textfile *input* (see **6.10**). When eof(f) is activated, it shall be an error if f is undefined; otherwise the function shall yield the value true if f.R is the empty sequence (see **6.4.3.5**), otherwise false.

eoln(f)

The parameter f shall be a textfile; if the actual-parameter-list is omitted, the function shall be applied to the required textfile *input* (see **6.10**). When eoln(f) is activated, it shall be an error if f is undefined or if eof(f) is true; otherwise the function shall yield the value true if f.R.first is an end-of-line component (see **6.4.3.5**), otherwise false.

6.7 Expressions

6.7.1 *General.* An expression shall denote a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use shall be an error. The use of a variable-access as a factor shall denote the value, if any, attributed to the variable accessed thereby. Operator precedences shall be according to four classes of operators as follows. The operator *not* shall have the highest precedence, followed by the multiplying-operators, then the adding-operators and signs, and finally, with the lowest precedence, the relational-operators. Sequences of two or more operators of the same precedence shall be left associative.

```
expression = simple-expression [ relational-operator simple-expression ] .
simple-expression = [ sign ] term { adding-operator term } .
term = factor { multiplying-operator factor } .
factor = variable-access | unsigned-constant | function-designator |
         set-constructor | "(" expression ")" | "not" factor .
```

NOTE 1. There is also a syntax rule for factor in **6.6.3.7**.

```
unsigned-constant = unsigned-number | character-string | constant-identifier |
                    "nil"
```

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

Any factor whose type is S, where S is a subrange of T, shall be treated as of type T. Similarly, any factor whose type is set of S shall be treated as of the unpacked canonical set-of-T type, and any factor whose type is packed set of S shall be treated as of the canonical packed set-of-T type.

NOTE 2. Consequently, an expression that consists of a single factor of type S is itself of type T, and an expression that consists of a single factor of type set of S is itself of type set of T, and an expression that consists of a single factor of type packed set of S is itself of type packed set of T.

A set-constructor shall denote a value of a set-type. The set-constructor *[ ]* shall denote that value in every set-type that contains no members. A set-constructor containing one or more member-designators shall denote either a value of the unpacked canonical set-of-T type or, if the context so requires, the packed canonical set-of-T type, where T is the type of every expression of each member-designator of the set-constructor. The type T shall be an ordinal-type. The value denoted by the set-constructor shall contain zero or more members each of which shall be denoted by at least one member-designator of the set-constructor.

The member-designator x, where x is an expression, shall denote the member that shall have the value x. The member-designator x..y, where x and y are expressions, shall denote zero or more members that shall have the values of the base-type in the closed interval from the value of x to the value of y. The order of evaluation of the expressions of a member-designator shall be implementation-dependent. The order of evaluation of the member-designators of a set-constructor shall be implementation-dependent.

NOTE 3. The member-designator x..y denotes no members if the value of x is greater than the value of y.

Examples:

(a) *Factors*:
```
x
15
(x + y + z)
sin(x + y)
[red, c, green]
[1, 5, 10..19, 23]
not p
```

(b) *Terms*:
```
x * y
i / (1 - i)
(x <= y) and (y < z)
```

(c) *Simple expressions*:
```
p or q
x + y
-x
hue1 + hue2
i * j + 1
```

(d) *Expressions*:
```
x = 1.5
p <= q
p = q and r
(i < j) = (j < k)
c in hue1
```

6.7.2 *Operators*

6.7.2.1 *General*

multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

adding-operator = "+" | "-" | "or" .

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

A factor, or a term, or a simple-expression shall be designated an operand. The order of evaluation of the operands of a dyadic operator shall be implementation-dependent.

NOTE. This means, for example, that the operands may be evaluated in textual order, or in reverse order, or in parallel or they may not both be evaluated.

6.7.2.2 *Arithmetic operators.* The types of operands and results for dyadic and monadic operations shall be as shown in tables 2 and 3 respectively.

Table 2. Dyadic arithmetic operations

Operator	Operation	Type of operands	Type of result
+	addition	integer-type or real-type	integer-type if both operands are of integer-type; otherwise real-type
-	subtraction	integer-type or real-type	
*	multiplication	integer-type or real-type	
/	division	integer-type or real-type	real-type
div	division with truncation	integer-type	integer-type
mod	modulo	integer-type	integer-type

Table 3. Monadic arithmetic operations

Operator	Operation	Type of operand	Type of result
+	identity	integer-type real-type	integer-type real-type
-	sign-inversion	integer-type real-type	integer-type real-type

NOTE 1. The symbols +, - and * are also used as set operators (see 6.7.2.4).

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

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

abs(i) − abs(j) < abs((i div j) * j) ≤ abs(i)

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

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

0 ≤ i mod j < j.

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

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

(a) All integral values in the closed interval from *−maxint* to *+maxint* shall be values of the integer-type.

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

(c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.

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

The results of the real arithmetic operators and functions shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined.

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

6.7.2.3 *Boolean operators.* Operands and results for Boolean operations shall be of Boolean-type. Boolean operators *or*, *and* and *not* shall denote respectively the logical operations of disjunction, conjunction and negation.

Boolean-expression = expression .

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

6.7.2.4 *Set operators.* The types of operands and results for set operations shall be as shown in table 4.

Table 4. Set operations

Operator	Operation	Type of operands	Type of result
+	set union	A canonical set-of-T type (see **6.7.1**)	same as the operands
−	set difference		
*	set intersection		

6.7.2.5 *Relational-operators.* The types of operands and results for relational operations shall be as shown in table 5.

Table 5. Relational operations

Operator	Type of operands	Type of result
= <>	any simple, pointer or string-type or canonical set-of-T type	Boolean-type
< >	any simple or string-type	Boolean-type
<= >=	any simple or string-type or canonical set-of-T type	Boolean-type
in	left operand: any ordinal type T right operand: a canonical set-of-T type	Boolean-type

The operands of =, <>, <, >, >=, and <= shall be either of compatible types, the same canonical set-of-T type, or one operand shall be of real-type and the other shall be of integer-type.

The operators =, <>, <, > shall stand for *equal to, not equal to, less than* and *greater than* respectively.

Except when applied to sets, the operators <= and >= shall stand for *less than or equal to* and *greater than or equal to* respectively.

Where u and v denote operands of a set-type, u <= v shall denote the inclusion of u in v and u >= v shall denote the inclusion of v in u.

NOTE. Since the Boolean-type is an ordinal-type with false less than true, then if p and q are operands of Boolean-type, p = q denotes their equivalence and p <= q means p implies q.

When the relational operators = , <> , < , > , <= , >= are used to compare operands of compatible string-types (see **6.4.3.2**), they denote lexicographic relations defined below. Lexicographic ordering imposes a total ordering on values of a string-type. If s1 and s2 are two values of compatible string-types, then

```
s1 = s2 iff for all i in [1..n]: s1[i] = s2[i]

s1 < s2 iff there exists a p in [1..n]:
          (for all i in [1..p-1]: s1[i] = s2[i]) and s1[p] < s2[p]
```

The operator *in* shall yield the value true if the value of the operand of ordinal-type is a member of the value of the set-type, otherwise it shall yield the value false.

6.7.3 *Function-designators.* A function-designator shall specify the activation of the block of the function-block associated with the function-identifier of the function-designator, and shall yield the value of the result of the activation upon completion of the algorithm of the activation; it shall be an error if the result is undefined upon completion of the algorithm. If the function has any formal parameters the function-designator shall contain a list of actual-parameters that shall be bound to their corresponding formal parameters defined in the function-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be

equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by **6.6.3**. The order of evaluation, accessing and binding of the actual-parameters shall be implementation-dependent.

```
function-designator = function-identifier [ actual-parameter-list ] .
actual-parameter-list = "(" actual-parameter { "," actual-parameter } ")" .
actual-parameter =   expression ¦ variable-access ¦ procedure-identifier ¦
                       function-identifier .
```

Examples:

```
Sum(a, 63)
GCD(147, k)
sin(x + y)
eof(f)
ord(ff)
```

6.8 Statements

6.8.1 *General*. Statements shall denote algorithmic actions, and shall be executable.

NOTE 1. Statements may be prefixed by a label.

A label, if any, of a statement S shall be designated as prefixing S, and shall be allowed to occur in a goto-statement G (see **6.8.2.4**) if and only if any of the following three conditions is satisfied.

(a) S contains G.

(b) S is a statement of a statement-sequence containing G.

(c) S is a statement of the statement-sequence of the compound-statement of the statement-part of a block containing G.

```
statement = [ label ":" ] ( simple-statement ¦ structured-statement ) .
```

NOTE 2. A goto-statement within a block may refer to a label in an enclosing block, provided that the label prefixes a simple-statement or structured-statement at the outermost level of nesting of the block.

6.8.2 *Simple-statements*

6.8.2.1 *General*. A simple-statement shall be a statement not containing a statement. An empty-statement shall contain no symbol and shall denote no action.

```
simple-statement = empty-statement ¦ assignment-statement ¦
                     procedure-statement ¦ goto-statement .
empty-statement = .
```

6.8.2.2 *Assignment-statements*. An assignment-statement shall attribute the value of the expression of the assignment-statement either to the variable denoted by the variable-access of the assignment-statement, or to the activation result that is denoted by the function-identifier of the assignment-statement; the value shall be assignment-compatible with the type possessed, respectively, by the variable or by the activation result. The function-block associated (see **6.6.2**) with the function-identifier of an assignment-statement shall contain the assignment-statement.

```
assignment-statement = ( variable-access ¦ function-identifier ) ":=" expression .
```

The decision as to the order of accessing the variable and evaluating the expression

shall be implementation-dependent; the access shall establish a reference to the variable during the remaining execution of the assignment-statement.

The state of a variable or activation result when the variable or activation result does not have attributed to it a value specified by its type shall be designated *undefined*. If a variable possesses a structured-type, the state of the variable when every component of the variable is totally-undefined shall be designated *totally-undefined*. Totally-undefined shall be synonymous with undefined for an activation result or a variable that does not possess a structured-type.

Examples:

```
x := y + z
p := (1 <= i) and (i < 100)
i := sqr(k) - (i * j)
hue1 := [blue, succ(c)]
p11.mother := true
```

6.8.2.3 *Procedure-statements*. A procedure-statement shall specify the activation of the block of the procedure-block associated with the procedure-identifier of the procedure-statement. If the procedure has any formal parameters the procedure-statement shall contain an actual-parameter-list, which is the list of actual-parameters that shall be bound to their corresponding formal parameters defined in the procedure-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by **6.6.3**. The order of evaluation, accessing and binding of the actual-parameters shall be implementation-dependent.

The procedure-identifier in a procedure-statement containing a read-parameter-list shall denote the required procedure *read*; the procedure-identifier in a procedure-statement containing a readln-parameter-list shall denote the required procedure *readln*; the procedure-identifier in a procedure-statement containing a write-parameter-list shall denote the required procedure *write*; the procedure-identifier in a procedure-statement containing a writeln-parameter-list shall denote the required procedure *writeln*.

```
procedure-statement = procedure-identifier ( [ actual-parameter-list ] |
                        read-parameter-list | readln-parameter-list |
                        write-parameter-list | writeln-parameter-list ) .
```

Examples:

```
printheading
transpose(a, n, m)
bisect(fct, -1.0, +1.0, x)
AddVectors(m[1], m[2], m[k])
```

NOTE. The fourth example is not for level 0.

6.8.2.4 *Goto-statements*. A goto-statement shall indicate that further processing is to continue at the program-point denoted by the label in the goto-statement and shall cause the termination of all activations except:

(a) the activation containing the program-point; *and*

(b) any activation containing the activation-point of an activation required by exceptions (a) or (b) not to be terminated.

```
goto-statement = "goto" label .
```

6.8.3 *Structured-statements*

6.8.3.1 *General*

```
structured-statement = compound-statement | conditional-statement|
                       repetitive-statement | with-statement .
statement-sequence = statement { ";" statement } .
```

The execution of a statement-sequence shall specify the execution of the statements of the statement-sequence in textual order, except as modified by execution of a goto-statement.

6.8.3.2 *Compound-statements*. A compound-statement shall specify execution of the statement-sequence of the compound-statement.

```
compound-statement = "begin" statement-sequence "end" .
```

Example: begin z := x ; x := y; y := z end

6.8.3.3 *Conditional-statements*

```
conditional-statement = if-statement | case-statement .
```

6.8.3.4 *If-statements*

```
if-statement = "if" Boolean-expression "then" statement [ else-part ] .
else-part = "else" statement .
```

If the Boolean-expression of the if-statement yields the value true, the statement of the if-statement shall be executed. If the Boolean-expression yields the value false, the statement of the if-statement shall not be executed and the statement of the else-part (if any) shall be executed.

An if-statement without an else-part shall not be immediately followed by the token *else*.

NOTE. An else-part is thus paired with the nearest preceding otherwise unpaired *then*.

Examples:

```
if x < 1.5 then z := x + y else z := 1.5

if p1 <> nil then p1 := p1↑.father

if j = 0 then
   if i = 0 then writeln('indefinite')
   else writeln('infinite')
else writeln( i / j )
```

6.8.3.5 *Case-statements*. The values denoted by the case-constants of the case-constant-lists of the case-list-elements of a case-statement shall be distinct and of the same ordinal-type as the expression of the case-index of the case-statement. On execution of the case-statement the case-index shall be evaluated. That value shall then specify execution of the statement of the case-list-element closest-containing the case-constant denoting that value. One of the case-constants shall be equal to the value of the case-index upon entry to the case-statement, otherwise it shall be an error.

NOTE. Case-constants are not the same as statement labels.

```
case-statement = "case" case-index "of" case-list-element
                    { ";" case-list-element } [ ";" ] "end" .
case-list-element = case-constant-list ":" statement .
case-index = expression .
```

Example:

```
case operator of
   plus:    x := x + y;
   minus:   x := x - y;
   times:   x := x * y
end
```

6.8.3.6 *Repetitive-statements*. Repetitive-statements shall specify that certain statements are to be executed repeatedly.

```
repetitive-statement = repeat-statement | while-statement | for-statement .
```

6.8.3.7 *Repeat-statements*

```
repeat-statement = "repeat" statement-sequence "until" Boolean-expression .
```

The statement-sequence of the repeat-statement shall be repeatedly executed (except as modified by the execution of a goto-statement) until the Boolean-expression of the repeat-statement yields the value true on completion of the statement-sequence. The statement-sequence shall be executed at least once, because the Boolean-expression is evaluated after execution of the statement-sequence.

Example:

```
repeat k := i mod j;
   i := j;
   j := k
until j = 0
```

6.8.3.8 *While-statements*

```
while-statement = "while" Boolean-expression "do" statement .
```

The while-statement

```
while b do body
```

shall be equivalent to

```
begin
if b then
   repeat
      body
   until not (b)
end
```

Examples:

```
while i > 0 do
  begin if odd(i) then z := z * x;
    i := i div 2;
    x := sqr(x)
  end

while not eof(f) do
  begin process(ff); get(f)
  end
```

6.8.3.9 *For-statements*. The for-statement shall specify that the statement of the for-statement is to be repeatedly executed while a progression of values is attributed to a variable that is designated the control-variable of the for-statement.

```
for-statement = "for" control-variable ":=" initial-value ( "to" | "downto" )
                  final-value "do" statement .
control-variable = entire-variable .
initial-value = expression .
final-value = expression .
```

The control-variable shall be an entire-variable whose identifier is declared in the variable-declaration-part of the block closest-containing the for-statement. The control-variable shall possess an ordinal-type, and the initial-value and final-value shall be of a type compatible with this type. The initial-value and the final-value shall be assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed. After a for-statement is executed (other than being left by a goto-statement leading out of it) the control-variable shall be undefined. Neither a for-statement nor any procedure-and-function-declaration-part of the block that closest-contains a for-statement shall contain a statement threatening the variable denoted by the control-variable of the for-statement.

A statement S shall be designated as *threatening* a variable V if one or more of the following statements is true.

(a) S is an assignment-statement and V is denoted by the variable-access of S.

(b) S contains an actual variable parameter that denotes V.

(c) S is a procedure-statement that specifies the activation of the required procedure *read* or the required procedure *readln*, and V is denoted by an actual parameter contained by S.

(d) S is a for-statement and the control-variable of S denotes V.

Apart from the restrictions imposed by these requirements, the for-statement

```
for v := e1 to e2 do body
```

shall be equivalent to

```
begin
temp1 := e1;
temp2 := e2;
if temp1 <= temp2 then
   begin
   v := temp1;
   body;
   while v <> temp2 do
      begin
      v := succ(v);
      body
      end
   end
end
```

and the for-statement

```
for v := e1 downto e2 do body
```

shall be equivalent to

```
begin
temp1 := e1;
temp2 := e2;
if temp1 >= temp2 then
   begin
   v := temp1;
   body;
   while v <> temp2 do
      begin
      v := pred(v);
      body
      end
   end
end
```

where temp1 and temp2 denote auxiliary variables that the program does not otherwise contain, and that possess the type possessed by the variable v if that type is not a subrange-type; otherwise the host type of the type possessed by the variable v.

Examples:

```
for i := 2 to 63 do
   if a[i] > max then max := a[i]

for i := 1 to 10 do
   for j := 1 to 10 do
      begin
      x := 0;
      for k := 1 to 10 do
         x := x + m1[i,k] * m2[k,j];
      m[i,j] := x
      end

for i:= 1 to 10 do
   for j := 1 to i - 1 do
      m[i][j] := 0.0
```

```
for c := blue downto red do q(c)
```

6.8.3.10 *With-statements*

```
with-statement = "with" record-variable-list "do" statement .
record-variable-list = record-variable { "," record-variable } .
field-designator-identifier = identifier .
```

A with-statement shall specify the execution of the statement of the with-statement. The occurrence of a record-variable as the only record-variable in the record-variable-list of a with-statement shall constitute a defining-point of each of the field-identifiers associated with components of the record-type possessed by the record-variable as a field-designator-identifier for the region that is the statement of the with-statement; each applied occurrence of a field-designator-identifier shall denote that component of the record-variable that is associated with the field-identifier by the record-type. The record-variable shall be accessed before the statement of the with-statement is executed, and that access shall establish a reference to the variable during the entire execution of the statement of the with-statement.

The statement

```
with v1,v2, ...,vn do s
```

shall be equivalent to

```
with v1 do
  with v2 do
    ...
      with vn do s
```

Example:

```
with date do
if month = 12 then
  begin month := 1; year := year + 1
  end
else month := month+1
```

has the same effect on the variable *date* as

```
if date.month = 12 then
  begin date.month := 1; date.year := date.year+1
  end
else date.month := date.month+1
```

6.9 Input and output

6.9.1 *The procedure read*. The syntax of the parameter list of *read* when applied to a textfile shall be:

```
read-parameter-list = "(" [ file-variable "," ] variable-access
                          { "," variable-access } ")" .
```

If the file-variable is omitted, the procedure shall be applied to the required textfile *input*.

The following requirements shall apply for the procedure *read* (where f denotes a textfile and v1...vn denote variable-accesses possessing the char-type (or a subrange of char-type), the integer-type (or a subrange of integer-type), or the real-type).

(a) read(f,v1,....,vn) shall be equivalent to

```
begin read(f,v1); ... ; read(f,vn) end
```

(b) If v is a variable-access possessing the char-type (or subrange thereof), read(f,v) shall be equivalent to

```
begin v := f↑; get(f) end
```

NOTE. The variable-access is not a variable parameter. Consequently it may be a component of a packed structure and the value of the buffer-variable need only be assignment-compatible with it.

(c) If v is a variable-access possessing the integer-type (or subrange thereof), read(f,v) shall cause the reading from f of a sequence of characters. Preceding spaces and end-of-lines shall be skipped. It shall be an error if the rest of the sequence does not form a signed-integer according to the syntax of **6.1.5**. Reading shall cease as soon as the buffer-variable f↑ does not have attributed to it a character contained by the signed-integer. The value of the signed-integer thus read shall be assignment-compatible with the type possessed by v, and shall be attributed to v.

(d) If v is a variable-access possessing the real-type, read(f,v) shall cause the reading from f of a sequence of characters. Preceding spaces and end-of-lines shall be skipped. It shall be an error if the rest of the sequence does not form a signed-number according to the syntax of **6.1.5**. Reading shall cease as soon as the buffer-variable f↑ does not have attributed to it a character contained by the signed-number. The value denoted by the number thus read shall be attributed to the variable v.

(e) When *read* is applied to f, it shall be an error if the buffer-variable f↑ is undefined or the pre-assertions for *get* do not hold (see **6.4.3.5**).

6.9.2 *The procedure readln.* The syntax of the parameter list of *readln* shall be:

```
readln-parameter-list = [ "(" ( file-variable | variable-access )
                          { "," variable-access } ")" ] .
```

Readln shall only be applied to textfiles. If the file-variable or the entire readln-parameter-list is omitted, the procedure shall be applied to the required textfile *input*.

Readln(f,v1,....,vn) shall be equivalent to

```
begin read(f,v1,....,vn); readln(f) end
```

Readln(f) shall be equivalent to

```
begin while not eoln(f) do get(f); get(f) end
```

NOTE. The effect of *readln* is to place the current file position just past the end of the current line in the textfile. Unless this is the end-of-file position, the current file position is therefore at the start of the next line.

6.9.3 *The procedure write.* The syntax of the parameter list of *write* when applied to a textfile shall be:

```
write-parameter-list = "(" [ file-variable "," ] write-parameter
                           { "," write-parameter } ")" .
write-parameter = expression [ ":" expression [ ":" expression ] ] .
```

If the file-variable is omitted, the procedure shall be applied to the required textfile

output. When *write* is applied to a textfile f, it shall be an error if f is undefined or f.M = Inspection (see **6.4.3.5**). An application of *write* to a textfile f shall cause the buffer-variable ff to become undefined.

Write(f,p1,....,pn) shall be equivalent to

```
begin write(f,p1); ... ; write(f,pn) end
```

where f denotes a textfile, and p1,....,pn denote write-parameters.

6.9.3.1 *Write-parameters*. The write-parameters p shall have the following forms:

```
e : TotalWidth : FracDigits
e : TotalWidth
e
```

where e is an expression whose value is to be written on the file f and may be of integer-type, real-type, char-type, Boolean-type or a string-type, and where *TotalWidth* and *FracDigits* are expressions of integer-type whose values are the field-width parameters. The values of *TotalWidth* and *FracDigits* shall be greater than or equal to one; it shall be an error if either value is less than one.

Write(f,e) shall be equivalent to the form write(f,e : TotalWidth), using a default value for *TotalWidth* that depends on the type of e; for integer-type, real-type and Boolean-type the default values shall be implementation-defined.

Write(f,e : TotalWidth : FracDigits) shall be applicable only if e is of real-type (see **6.9.3.4.2**).

6.9.3.2 *Char-type*. If e is of char-type, the default value of *TotalWidth* shall be one. The representation written on the file f shall be:

(TotalWidth - 1) spaces, the character value of e.

6.9.3.3 *Integer-type*. If e is of integer-type, the decimal representation of e shall be written on the file f. Assume a function

```
function IntegerSize ( x : integer ) : integer ;
  { returns the number of digits, z, such that
  10 to the power (z-1) ≤ abs(x) < 10 to the power z }
```

and let *IntDigits* be the positive integer defined by:

```
if e = 0
then IntDigits := 1
else IntDigits := IntegerSize(e);
```

then the representation shall consist of:

(a) if TotalWidth ≥ IntDigits + 1:
(TotalWidth - IntDigits - 1) spaces,
the sign character: '-' if e < 0, otherwise a space,
IntDigits digit-characters of the decimal representation of abs(e).

(b) If TotalWidth < IntDigits + 1:
if e < 0 the sign character '-',
IntDigits digit-characters of the decimal representation of abs(e).

6.9.3.4 *Real-Type*. If e is of real-type, a decimal representation of the number e, rounded to the specified number of significant figures or decimal places, shall be

written on the file f.

6.9.3.4.1 *The floating-point representation.* Write(f,e : TotalWidth) shall cause a floating-point representation of e to be written. Assume functions

```
function TenPower ( Int : integer ) : real ;
   { Returns 10.0 raised to the power Int }

function RealSize ( y : real ) : integer ;
   { Returns the value, z, such that TenPower(z-1) ≤ abs(y) < TenPower(z) }

function Truncate ( y : real ; DecPlaces : integer ) : real ;
   { Returns the value of y after truncation to DecPlaces decimal places }
```

let *ExpDigits* be an implementation-defined value representing the number of digit-characters written in an exponent;

let *ActWidth* be the positive integer defined by:

```
if TotalWidth >= ExpDigits + 6
   then ActWidth := TotalWidth
   else ActWidth := ExpDigits + 6;
```

and let the non-negative number *eWritten*, the positive integer *DecPlaces* and the integer *ExpValue* be defined by:

```
DecPlaces := ActWidth - ExpDigits - 5;
if e = 0.0
   then begin eWritten := 0.0; ExpValue := 0 end
   else
   begin
   eWritten := abs(e);
   ExpValue := RealSize ( eWritten ) - 1;
   eWritten := eWritten / TenPower ( ExpValue );
   eWritten := eWritten + 0.5 * TenPower ( -Decplaces );
   if eWritten >= 10.0
      then
      begin
      eWritten := eWritten / 10.0;
      ExpValue := ExpValue + 1
      end;
   eWritten := Truncate ( eWritten, DecPlaces )
   end;
```

then the floating-point representation of the value of e shall consist of:

the sign character ('-' if (e < 0) and (eWritten > 0), otherwise a space),
the leading digit-character of the decimal representation of eWritten,
the character '.' ,
the next DecPlaces digit-characters of the decimal representation of eWritten,
an implementation-defined exponent character (either 'e' or 'E'),
the sign of ExpValue ('-' if ExpValue < 0, otherwise '+'),
the ExpDigits digit-characters of the decimal representation of ExpValue
(with leading zeros if the value requires them).

6.9.3.4.2 *The fixed-point representation.* Write(f,e : TotalWidth : FracDigits) shall cause a fixed-point representation of e to be written. Assume the functions *TenPower* and *Truncate* described in **6.9.3.4.1**;

let *eWritten* be the non-negative number defined by:

```
if e = 0.0
  then eWritten := 0.0
  else
  begin
  eWritten := abs(e);
  eWritten := eWritten + 0.5 * TenPower ( - FracDigits );
  eWritten := Truncate ( eWritten, FracDigits )
  end;
```

let *IntDigits* be the positive integer defined by:

```
if RealSize ( eWritten ) < 1
  then IntDigits := 1
  else IntDigits := RealSize ( eWritten );
```

and let *MinNumChars* be the positive integer defined by:

```
MinNumChars := IntDigits + FracDigits + 1;
if (e < 0.0) and (eWritten > 0)
  then MinNumChars := MinNumChars + 1; {'-' required}
```

then the fixed-point representation of the value of e shall consist of:

```
if TotalWidth ≥ MinNumChars, (TotalWidth - MinNumChars) spaces,
the character '-' if (e < 0) and (eWritten > 0),
the first IntDigits digit-characters of the decimal representation of the
  value of eWritten,
the character '.',
the next FracDigits digit-characters of the decimal representation
  of the value of eWritten.
```

NOTE. At least *MinNumChars* characters are written. If *TotalWidth* is less than this value, no initial spaces are written.

6.9.3.5 *Boolean-type*. If e is of Boolean-type, a representation of the word true or the word false (as appropriate to the value of e) shall be written on the file f. This shall be equivalent to writing the appropriate character-strings 'True' or 'False' (see **6.9.3.6**), where the case of each letter is implementation-defined, with a field-width parameter of *TotalWidth*.

6.9.3.6 *String-types*. If the type of e is a string-type with n components, the default value of *TotalWidth* shall be n. The representation shall consist of:

```
if TotalWidth > n,
  (TotalWidth - n) spaces,
  the first through nth characters of the value of e in that order.

if 1 ≤ TotalWidth ≤ n,
  the first through TotalWidthth characters in that order.
```

6.9.4 *The procedure writeln*. The syntax of the parameter list of *writeln* shall be:

```
writeln-parameter-list = [ "(" ( file-variable | write-parameter )
                           { "," write-parameter } ")" ] .
```

Writeln shall only be applied to textfiles. If the file-variable or the writeln-parameter-list is omitted, the procedure shall be applied to the required textfile *output*.

Writeln(f,p1,...,pn) shall be equivalent to

```
begin write(f,p1,...,pn); writeln(f) end
```

Writeln shall be defined by a pre-assertion and a post-assertion using the notation of **6.6.5.2**.

pre-assertion: (f0 is not undefined) and (f0.M = Generation) and (f0.R = S0).

post-assertion: (f.L = (f0.L^S(e))) and (f↑ is totally-undefined) and (f.R = S0) and (f.M = Generation), where S(e) is the sequence consisting solely of the end-of-line component defined in **6.4.3.5**.

NOTE. Writeln(f) terminates the partial line, if any, which is being generated. By the conventions of **6.6.5.2** it is an error if the pre-assertion is not true prior to writeln(f).

6.9.5 *The procedure page*. It shall be an error if the pre-assertion required for writeln(f) (see **6.9.4**) does not hold prior to the activation of page(f). If the actual-parameter-list is omitted the procedure shall be applied to the required textfile *output*. Page(f) shall cause an implementation-defined effect on the textfile f, such that subsequent text written to f will be on a new page if the textfile is printed on a suitable device, shall perform an implicit writeln(f) if f.L is not empty and if f.L.last is not the end-of-line component (see **6.4.3.5**), and shall cause the buffer-variable f↑ to become totally-undefined. The effect of inspecting a textfile to which the *page* procedure was applied during generation shall be implementation-dependent.

6.10 Programs

```
program = program-heading ";" program-block "." .
program-heading = "program" identifier [ "(" program-parameters ")" ] .
program-parameters = identifier-list .
program-block = block .
```

The identifier of the program-heading shall be the program name that shall have no significance within the program. The identifiers contained by the program-parameters shall be distinct and shall be designated program parameters. Each program parameter shall have a defining-point as a variable-identifier for the region that is the program-block. The binding of the variables denoted by the program parameters to entities external to the program shall be implementation-dependent, except if the variable possesses a file-type in which case the binding shall be implementation-defined.

NOTE. The external representation of such external entities is not defined by this standard, nor is any property of a Pascal program dependent on such representation.

The occurrence of the required identifier *input* or the required identifier *output* as a program parameter shall constitute its defining-point for the region that is the program-block as a variable-identifier of the required type denoted by the required type-identifier *text*. Such occurrence of the identifier *input* shall cause the post-assertions of *reset* to hold, and of *output*, the post-assertions of *rewrite* to hold, prior to the first access to the textfile or its associated buffer-variable. The effect of the application of the required procedure *reset* or the required procedure *rewrite* to either of these textfiles shall be implementation-defined.

Examples:

```
program copy (f, g);
var f, g : file of real;
begin reset(f); rewrite(g);
      while not eof(f) do
         begin g↑ := f↑; get(f); put(g)
         end
end.
```

```
program copytext (input, output);
{This program copies the characters and line structure of the textfile input to the
textfile output.}
var ch : char;
begin
   while not eof do
   begin
      while not eoln do
         begin read(ch); write(ch)
         end;
      readln; writeln
   end
end.
```

```
program t6p6p3p4 (output);
var globalone, globaltwo : integer;

procedure dummy;
   begin
   writeln('fail4')
   end { of dummy };

procedure p (procedure f(procedure ff; procedure gg); procedure g);
   var localtop : integer;
   procedure r;
      begin
      if globalone = 1 then
         begin
         if (globaltwo <> 2) or (localtop <> 1) then
            writeln('fail1')
         end
      else if globalone = 2 then
         begin
         if (globaltwo <> 2) or (localtop <> 2) then
            writeln('fail2')
         else
            writeln('pass')
         end
      else
         writeln('fail3');
      globalone := globalone + 1
      end { of r };
   begin { of p }
   globaltwo := globaltwo + 1;
   localtop := globaltwo;
   if globaltwo = 1 then
      p(f, r)
   else
      f(g, r)
   end { of p};

procedure q (procedure f; procedure g);
   begin
   f;
   g
   end { of q};

begin
globalone := 1;
globaltwo := 0;
p(q, dummy)
end.
```

Appendix A

Collected syntax

The non-terminal symbols *pointer-type*, *program*, *signed-number*, *simple-type*, *special-symbol* and *structured-type* are only referenced by the semantics and are not used in the right-hand-side of any production. The non-terminal symbol *program* is the start symbol of the grammar.

```
actual-parameter = expression | variable-access | procedure-identifier |
                   function-identifier .

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

adding-operator = "+" | "-" | "or" .

apostrophe-image = "'" .

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

array-variable = variable-access .

assignment-statement = ( variable-access | function-identifier ) ":=" expression .

base-type = ordinal-type .

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

Boolean-expression = expression .

bound-identifier = identifier .

buffer-variable = file-variable "↑" .

case-constant = constant .

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

case-index = expression .

case-list-element = case-constant-list ":" statement .

case-statement = "case" case-index "of" case-list-element
                 { ";" case-list-element } [ ";" ] "end" .

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

component-type = type-denoter .

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

compound-statement = "begin" statement-sequence "end" .

conditional-statement = if-statement | case-statement .

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

```
conformant-array-schema = packed-conformant-array-schema |
                          unpacked-conformant-array-schema .

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

constant-definition = identifier "=" constant .

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

constant-identifier = identifier .

control-variable = entire-variable .

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

digit-sequence = digit { digit } .

directive = letter { letter | digit } .

domain-type = type-identifier .

else-part = "else" statement .

empty-statement = .

entire-variable = variable-identifier .

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

expression = simple-expression [ relational-operator simple-expression ] .

factor = variable-access | unsigned-constant | bound-identifier |
         function-designator | set-constructor | "(" expression ")" |
         "not" factor .

field-designator = record-variable "." field-specifier | field-designator-identifier .

field-designator-identifier = identifier .

field-identifier = identifier .

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

field-specifier = field-identifier .

file-type = "file" "of" component-type .

file-variable = variable-access .

final-value = expression .

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

for-statement = "for" control-variable ":=" initial-value ( "to" | "downto" )
                final-value "do" statement .

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

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

fractional-part = digit-sequence .

function-block = block .

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

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

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

function-identification = "function" function-identifier .

function-identifier = identifier .

functional-parameter-specification = function-heading .

goto-statement = "goto" label .

identified-variable = pointer-variable "↑" .

identifier = letter { letter | digit } .

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

if-statement = "if" Boolean-expression "then" statement [ else-part ] .

index-expression = expression .

index-type = ordinal-type .

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

indexed-variable = array-variable "[" index-expression { "," index-expression } "]" .

initial-value = expression .

label = digit-sequence .

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

letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
         "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" .

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

multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

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

new-pointer-type = "↑" domain-type .
```

```
new-structured-type = [ "packed" ] unpacked-structured-type .

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

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

ordinal-type-identifier = type-identifier .

packed-conformant-array-schema = "packed" "array" "[" index-type-specification "]"
                                 "of" type-identifier .

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

pointer-type-identifier = type-identifier .

pointer-variable = variable-access .

procedural-parameter-specification = procedure-heading .

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

procedure-block = block .

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

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

procedure-identification = "procedure" procedure-identifier .

procedure-identifier = identifier .

procedure-statement = procedure-identifier ( [ actual-parameter-list ] |
                      read-parameter-list | readln-parameter-list |
                      write-parameter-list | writeln-parameter-list ) .

program = program-heading ";" program-block "." .

program-block = block .

program-heading = "program" identifier [ "(" program-parameters ")" ] .

program-parameters = identifier-list .

read-parameter-list = "(" [ file-variable "," ] variable-access
                      { "," variable-access } ")" .

readln-parameter-list = [ "(" ( file-variable | variable-access )
                        { "," variable-access } ")" ] .

real-type-identifier = type-identifier .

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

record-type = "record" field-list "end" .

record-variable = variable-access .
```

```
record-variable-list = record-variable { "," record-variable } .

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

repeat-statement = "repeat" statement-sequence "until" Boolean-expression .

repetitive-statement = repeat-statement | while-statement | for-statement .

result-type = simple-type-identifier | pointer-type-identifier .

scale-factor = signed-integer .

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

set-type = "set" "of" base-type .

sign = "+" | "-" .

signed-integer = [ sign ] unsigned-integer .

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

signed-real = [ sign ] unsigned-real .

simple-expression = [ sign ] term { adding-operator term } .

simple-statement = empty-statement | assignment-statement | procedure-statement |
                   goto-statement .

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

simple-type-identifier = type-identifier .

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

statement = [ label ":" ] ( simple-statement | structured-statement ) .

statement-part = compound-statement .

statement-sequence = statement { ";" statement } .

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

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

structured-statement = compound-statement | conditional-statement |
                       repetitive-statement | with-statement .

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

structured-type-identifier = type-identifier .

subrange-type = constant ".." constant .

tag-field = identifier .

tag-type = ordinal-type-identifier .
```

```
term = factor { multiplying-operator factor } .

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

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

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

type-identifier = identifier .

unpacked-conformant-array-schema = "array" "[" index-type-specification
                                     { ";" index-type-specification } "]" "of"
                                     ( type-identifier |
                                     conformant-array-schema ) .

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

unsigned-constant = unsigned-number | character-string | constant-identifier |
                    "nil" .

unsigned-integer = digit-sequence .

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

unsigned-real = unsigned-integer "." fractional-part [ "e" scale-factor ] |
                unsigned-integer "e" scale-factor .

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

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

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

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

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

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

variable-identifier = identifier .

variable-parameter-specification = "var" identifier-list ":" type-identifier .

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

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

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

while-statement = "while" Boolean-expression "do" statement .

with-statement = "with" record-variable-list "do" statement .
```

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

write-parameter = expression [ ":" expression [ ":" expression ] ] .

write-parameter-list = "(" [ file-variable "," ] write-parameter
                       { "," write-parameter } ")" .

writeln-parameter-list = [ "(" ( file-variable | write-parameter )
                         { "," write-parameter } ")" ] .
```

Appendix B

Index

Appendix C

Required identifiers

Identifier	Reference(s)
abs	6.6.6.2
arctan	6.6.6.2
Boolean	6.4.2.2
char	6.4.2.2
chr	6.6.6.4
cos	6.6.6.2
dispose	6.6.5.3
eof	6.6.6.5
eoln	6.6.6.5
exp	6.6.6.2
false	6.4.2.2
get	6.6.5.2
input	6.10
integer	6.4.2.2
ln	6.6.6.2
maxint	6.7.2.2
new	6.6.5.3
odd	6.6.6.5
ord	6.6.6.4
output	6.10
pack	6.6.5.4
page	6.9.5
pred	6.6.6.4
put	6.6.5.2
read	6.6.5.2, 6.9.1
readln	6.9.2
real	6.4.2.2
reset	6.6.5.2
rewrite	6.6.5.2
round	6.6.6.3
sin	6.6.6.2
sqr	6.6.6.2
sqrt	6.6.6.2
succ	6.6.6.4
text	6.4.3.5
true	6.4.2.2
trunc	6.6.6.3
unpack	6.6.5.4
write	6.6.5.2, 6.9.3
writeln	6.9.4

Appendix D

Errors

D.0 A complying processor is required to provide documentation concerning its treatment of errors. To facilitate the production of such documentation, all the errors specified in clause 6 are described again in this appendix.

D.1 For an indexed-variable closest-containing a single index-expression, the value of the index-expression is assignment-compatible with the index-type of the array-type.

D.2 It is an error unless a variant is active for the entirety of each reference and access to each component of the variant.

D.3 It is an error if the pointer-variable of an identified-variable denotes a nil-value.

D.4 It is an error if the pointer-variable of an identified-variable is undefined.

D.5 It is an error to remove from its pointer-type the identifying-value of an identified variable when a reference to the identified variable exists.

D.6 It is an error to alter the value of a file-variable f when a reference to the buffer-variable f↑ exists.

D.7 For a value parameter, the actual-parameter is an expression of an ordinal-type whose value is assignment-compatible with the type possessed by the formal parameter.

D.8 For a value parameter, the actual-parameter is an expression of a set-type whose value is assignment-compatible with the type possessed by the formal parameter.

D.9 It is an error if the file mode is not Generation immediately prior to any use of *put*, *write*, *writeln* or *page*.

D.10 It is an error if the the file is undefined immediately prior to any use of *put*, *write*, *writeln* or *page*.

D.11 It is an error if end of file is not true immediately prior to any use of *put*, *write* (or *writeln* or *page*).

D.12 It is an error if the buffer-variable is undefined immediately prior to any use of *put*.

D.13 It is an error if the file is undefined immediately prior to any use of *reset*.

D.14 It is an error if the file mode is not Inspection immediately prior to any use of *get* or *read*.

D.15 It is an error if the file is undefined immediately prior to any use of *get* or *read*.

D.16 It is an error if end of file is true immediately prior to any use of *get* or *read*.

D.17 For *read*, the value possessed by the buffer-variable is assignment-compatible with the variable-access.

D.18 For *write*, the value possessed by the expression is assignment-compatible with

the buffer-variable.

D.19 For new(p,c1,...,cn), it is an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

D.20 For dispose(p), it is an error if the identifying-value had been created using the form new(p,c1,...,cn).

D.21 For dispose(p,k1,...,km), it is an error if the variable had been created using the form new(p,c1,...,cn) and m is not equal to n.

D.22 For dispose(p,k1,...,km), it is an error if the variants in the variable identified by the pointer value of p are different from those specified by the case-constants k1,...,km.

D.23 For *dispose*, it is an error if the parameter of a pointer-type has a nil-value.

D.24 For *dispose*, it is an error if the parameter of a pointer-type is undefined.

D.25 It is an error if a variable created using the second form of *new* is accessed by the identified-variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

D.26 For *pack*, the parameter of ordinal-type is assignment-compatible with the index-type of the unpacked array parameter.

D.27 For *pack*, it is an error if any of the components of the unpacked array are both undefined and accessed.

D.28 For *pack*, it is an error if the index-type of the unpacked array is exceeded.

D.29 For *unpack*, the parameter of ordinal-type is assignment-compatible with the index-type of the unpacked array parameter.

D.30 For *unpack*, it is an error if any of the components of the packed array are undefined.

D.31 For *unpack*, it is an error if the index-type of the unpacked array is exceeded.

D.32 Sqr(x) computes the square of x. It is an error if such a value does not exist.

D.33 For ln(x), it is an error if x is not greater than zero.

D.34 For sqrt(x), it is an error if x is negative.

D.35 For trunc(x), the value of trunc(x) is such that if x is positive or zero then $0 \leq x-\text{trunc}(x) < 1$; otherwise $-1 < x-\text{trunc}(x) \leq 0$. It is an error if such a value does not exist.

D.36 For round(x), if x is positive or zero then round(x) is equivalent to trunc(x+0.5), otherwise round(x) is equivalent to trunc(x-0.5). It is an error if such a value does not exist.

D.37 For chr(x), the function returns a result of char-type which is the value whose ordinal number is equal to the value of the expression x if such a character value exists. It is an error if such a character value does not exist.

D.38 For succ(x), the function yields a value whose ordinal number is one greater than that of x, if such a value exists. It is an error if such a value does not exist.

D.39 For pred(x), the function yields a value whose ordinal number is one less than that of x, if such a value exists. It is an error if such a value does not exist.

D.40 When eof(f) is activated, it is an error if f is undefined.

D.41 When eoln(f) is activated, it is an error if f is undefined.

D.42 When eoln(f) is activated, it is an error if eof(f) is true.

D.43 An expression denotes a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use is an error.

D.44 A term of the form x/y is an error if y is zero.

D.45 A term of the form i div j is an error if j is zero.

D.46 A term of the form i mod j is an error if j is zero or negative.

D.47 It is an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

D.48 It is an error if the result of an activation of a function is undefined upon completion of the algorithm of the activation.

D.49 For an assignment-statement, the expression is of an ordinal-type whose value is assignment-compatible with the type possessed by the variable or function-identifier.

D.50 For an assignment-statement, the expression is of a set-type whose value is assignment-compatible with the type possessed by the variable.

D.51 For a case-statement, it is an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

D.52 For a for-statement, the value of the initial-value is assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

D.53 For a for-statement, the value of the final-value is assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

D.54 On reading an integer from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-integer.

D.55 On reading an integer from a textfile, the value of the signed-integer read is assignment-compatible with the type possessed by variable-access.

D.56 On reading a number from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-number.

D.57 It is an error if the buffer-variable is undefined immediately prior to any use of *read*.

D.58 On writing to a textfile, the values of *TotalWidth* and *FracDigits* are greater than or equal to one; it is an error if either value is less than one.

D.59 For a conformant array, it is an error if the smallest or largest value specified by the index-type of the actual parameter lies outside the closed interval specified by the index-type-specification of the conformant-array-schema.

Standards publications referred to

See preface.

This British Standard, having been prepared under the direction of the Office and Information Standards Committee, was published under the authority of the Board of BSI and comes into effect on 31 March 1982

ISBN 0 580 12531 9

British Standards Institution

Incorporated by Royal Charter, BSI is the independent national body for the preparation of British Standards. It is the UK member of the International Organization for Standardization and UK sponsor of the British National Committee of the International Electrotechnical Commission.

Copyright

Contract requirements

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Revision of British Standards

British Standards are revised, when necessary, by the issue either of amendments or of revised editions. **It is important that users of British Standards should ascertain that they are in possession of the latest amendments or editions.** Information on all BSI publications is in the *BSI Catalogue*, supplemented each month by *BSI News* which is available to subscribing members of the Institution and gives details of new publications, revisions, amendments and withdrawn standards. Any person who, when making use of a British Standard, encounters an inaccuracy or ambiguity, is requested to notify BSI without delay in order that the matter may be investigated and appropriate action taken.

The following BSI references relate to the work on this standard:
Committee reference OIS/5 Draft for comment 79/60528 DC

Cooperating organizations

The Office and Information Standards Committee, under whose direction this British Standard was prepared, consists of representatives from the following:

*British Computer Society Ltd.
British Paper and Board Industry Federation (PIF)
*Business Equipment Trade Association
*Central Computer and Telecommunications Agency
Committee of London Clearing Bankers on behalf of the Committee of Scottish Clearing Bankers, Co-operative Bank, Central Trustee Savings Bank and Yorkshire Bank
*Computing Services Association
*Department of Industry (Computers, Systems and Electronics)
*Department of Industry (National Physical Laboratory)
Drawing Office Material Manufacturers' and Dealers' Association
*Electricity Supply Industry in England and Wales
HM Customs and Excise
Her Majesty's Stationery Office
Institute of Administrative Management
Institute of Chartered Secretaries and Administrators
Institute of Cost and Management Accountants
Institute of Data Processing Management
Institute of Purchasing and Supply
Institution of Electrical Engineers
Inter-university Committee on Computing
Local Authorities Management Services and Computer Committee
London Transport Executive
*Ministry of Defence
National Computer Users' Forum
*National Computing Centre Ltd.
National Research Development Corporation
Office Machines and Equipment Federation
*Post Office
Simplification of International Trade Procedures Board

The organizations marked with an asterisk in the above list, together with the following, were directly represented on the Technical Committee entrusted with the preparation of this British Standard:

Association for Literary and Linguistic Computing
Association of Computer Units in Colleges of Higher Education (ACUCHE)
British Gas Corporation
Control and Automation Manufacturers' Association (BEAMA)
Engineering Equipment Users' Association
Hatfield Polytechnic
UK APL Users' Group
University of Edinburgh
University of London

Amendments issued since publication

Amd. No.	Date of issue	Text affected

British Standards Institution · 2 Park Street London W1A 2BS · Telephone 01-629 9000 · Telex 266933

8404—8—0.5k—JWD

OIS/5

Appendix B
Compliance statements

B.1 ROM-based ISO-Pascal system for the BBC Microcomputer Model B

The above processor complies with the requirements of level zero of ISO 7185 with the following exceptions:

1. No implementation of pack/unpack or functions and procedures as parameters. This causes nine conformance and six pretest failures. These programs either test the non-implemented features or utilise them in testing other features.

2. Some checks are omitted due to code size limitations and implementation of the compiler in a ROM. These result in the following deviance test failures:

6.2.2–8 6.2.2–12	Tests that the defining-point of an identifier precedes all applied occurrences of that identifier, with the exception of pointer-type declarations.
6.6.1–3 6.6.1–4	Tests that scoping rules are not violated by incorrectly binding a procedure call to an 'ambiguous' defining occurrence.

The processor contains extensions to ISO 7185 as described in *ISO-Pascal on the BBC Microcomputer and Acorn Electron* (chapter 5). These extensions may be switched on/off by means of a compiler option.

The implementation defined features are as follows:

E.1 The value of each char-type corresponding to each allowed string-character (all but chr(4) and chr(13)) is as defined in E.3 below.

E.2 The subset of real numbers denoted by signed real is the set of values representable by 40-bit floating point, as used by BBC BASIC. This is about ten decimal places.

E.3 The values of char-type are the ISO 646 (ASCII) character set, with the following exceptions:

i. The grave accent (chr(96)) is replaced by the pound sign ('£').
ii. The pound sign (chr(35)) is replaced by the hash sign ('#').
iii. The overscore (chr(126)) is replaced by the tilde (' ~').
iv. The visible representation of any character is dependent on the current screen mode and soft character definitions in force. The

visible representation of control characters differs when in the Pascal editor, and when otherwise displayed on the console (see the *BBC Microcomputer System User Guide* and *ISO-Pascal on the BBC Microcomputer and Acorn Electron).*

E.4 See E.3.

E.5 The point at which file operations rewrite, put, reset and get are performed differs dependent on whether the file is resident on backing storage, or is connected to the console (text files only):

i. For files resident on backing storage, this point is determined by the selected filing system.
ii. For files connected to the console, a form of lazy-IO is used to ease interactive use. This can be characterised by the fact that get(f) is never performed until the point where the result of the operation is actually used.

E.6 The value of MAXINT is 2147483647.

E.7 The accuracy of the approximations of the real operations and functions is determined by the representation (see E.2), and by the rounding of the last bit of intermediate results. This gives approximately ten decimal digits of precision.

E.8 The default value of TotalWidth for integer-type is 12.

E.9 The default value of TotalWidth for real-type is 12.

E.10 The default value of TotalWidth for Boolean-type is 5.

E.11 The value of ExpDigits is 2.

E.12 The exponent character is 'e'.

E.13 The case used for output of Boolean-type is upper case.

E.14 The procedure page performs writeln if and only if required (to force EOLN as required by the Standard), and then outputs the form-feed character. The effect on any particular device depends upon that service.

E.15 File-type program parameters (ie all program parameters) can be bound to physical files (including the console, if text) by means of Acornsoft ISO-Pascal language extensions as described in *ISO-Pascal on the BBC Microcomputer and Acorn Electron* (section 5.6). Note that (file-type) program parameters may also be used as local files.

E.16 Alternative representations of symbols permitted by the Standard are supported by the implementation.

The following errors are not, in general, reported:

D.1, D.2, D.4, D.5, D.6, D.19, D.20, D.21, D.22, D.25, D.26, D.27, D.28, D.29, D.30, D.31, D.43, D.59 – level one or non-implemented feature deviance.

The following errors are detected prior to, or during, execution of a program:

D.3, D.7, D.8, D.9, D.10, D.11, D.12, D.13, D.14, D.15, D.16, D.17, D.18, D.23, D.24, D.32, D.33, D.34, D.35, D.36, D.37, D.38, D.39, D.40, D.41, D.42, D.44, D.45, D.46, D.47, D.48, D.49, D.50, D.51, D.52, D.53, D.54, D.55, D.56, D.57, D.58.

Implementation dependent features F.1–F.8, F.10 and F.11 of Pascal are treated as undetected errors. The binding of variables denoted by program parameters which are not of file-type is treated as a detected error (F.9).

B.2 Disc-based ISO-Pascal system for the BBC Microcomputer with 6502 Second Processor

The above processor complies with the requirements of level one of ISO 7185 with no exceptions.

The processor contains extensions to ISO 7185 as described in *ISO-Pascal on the BBC Microcomputer and Acorn Electron* (chapter 5). These extensions may be switched on/off by means of a compiler option.

The implementation defined features are as follows:

E.1 The value of each char-type corresponding to each allowed string-character (all but chr(4) and chr(13)) is as defined in E.3 below.

E.2 The subset of real numbers denoted by signed real is the set of values representable by 40-bit floating point, as used by BBC BASIC. This is about ten decimal places.

E.3 The values of char-type are the ISO 646 (ASCII) character set, with the following exceptions:

i. The grave accent (chr(96)) is replaced by the pound sign ('£').
ii. The pound sign (chr(35)) is replaced by the hash sign ('#').
iii. The overscore (chr(126)) is replaced by the tilde (' ~').
iv. The visible representation of any character is dependent on the current screen mode and soft character definitions in force. The visible representation of control characters differs when in the Pascal editor, and when otherwise displayed on the console (see the *BBC Microcomputer System User Guide* and *ISO-Pascal on the BBC Microcomputer and Acorn Electron*).

E.4 See E.3.

E.5 The point at which file operations rewrite, put, reset and get are performed differs dependent on whether the file is resident on backing storage, or is connected to the console (text files only):

i. For files resident on backing storage, this point is determined by the selected filing system.

ii. For files connected to the console, a form of lazy-IO is used to ease interactive use. This can be characterised by the fact that get(f) is never performed until the point where the result of the operation is actually used.

E.6 The value of MAXINT is 2147483647.

E.7 The accuracy of the approximations of the real operations and functions is determined by the representation (see E.2), and by the rounding of the last bit of intermediate results. This gives approximately ten decimal digits of precision.

E.8 The default value of TotalWidth for integer-type is 12.

E.9 The default value of TotalWidth for real-type is 12.

E.10 The default value of TotalWidth for Boolean-type is 5.

E.11 The value of ExpDigits is 2.

E.12 The exponent character is 'e'.

E.13 The case used for output of Boolean-type is upper case.

E.14 The procedure page performs writeln if and only if required (to force EOLN as required by the Standard), and then outputs the form-feed character. The effect on any particular device depends upon that service.

E.15 File-type program parameters (ie all program parameters) can be bound to physical files (including the console, if text) by means of Acornsoft ISO-Pascal language extensions as described in *ISO-Pascal on the BBC Microcomputer and Acorn Electron* (section 5.6). Note that (file-type) program parameters may also be used as local files.

E.16 Alternative representations of symbols permitted by the Standard are supported by the implementation.

The following errors are not, in general, reported:

D.2, D.4, D.5, D.6, D.19, D.20, D.21, D.22, D.25, D.27, D.28, D.30, D.31, D.43, D.59.

The following errors are detected prior to, or during, execution of a program:

D.3, D.7, D.8, D.9, D.10, D.11, D.12, D.13, D.14, D.15, D.16, D.17, D.18, D.23, D.24, D.32, D.33, D.34, D.35, D.36, D.37, D.38, D.39, D.40, D.41, D.42, D.44, D.45, D.46, D.47, D.48, D.49, D.50, D.51, D.52, D.53, D.54, D.55, D.56, D.57, D.58.

Implementation dependent features F.1–F.8, F.10 and F.11 of Pascal are treated as undetected errors. The binding of variables denoted by program parameters which are not of file-type is treated as a detected error (F.9).

Appendix C
Command summary

Here is a list of the commands that are available in Acornsoft ISO-Pascal:

CLOSE
Close all open files on the selected filing system.

COMPILE
Compile to and from memory.

COMPILE source file
Compile from source file memory.

COMPILE >object-file
Compile from memory to object file.

COMPILE source-file object-file
Compile from source file to object file.

EDIT [source-file]
Call the editor, optionally loading a source file.

GO [arguments]
RUN the object file in memory, passing the optional arguments if the T option was used at compilation.

LOAD object-file
Load the specified object file.

MODE number
Change the display mode to the one specified.

RUN object-file [arguments]
Load and run the specified code file.

SAVE object-file
Save the memory code file under the name given.

TRACE [0, 1 or 2]
Set the current TRACE level.

Appendix D

Editor command summary

Below is a summary of the cursor movement and function key commands available in the editor.

BBC editor	Function	Electron editor
Up arrow	Move up to a line	Up arrow
Down arrow	Move down a line	Down arrow
Left arrow	Move left a character	Left arrow
Right arrow	Move right a character	Right arrow
SHIFT up	Move up a page	FUNC N
SHIFT down	Move down a page	FUNC M
SHIFT left	Move to start of line	FUNC <
SHIFT right	Move to end of line	FUNC >
CTRL up	Move to top of text	FUNC Z
CTRL down	Move to end of text	FUNC X
DELETE	Delete left of the cursor	DELETE
COPY	Delete at the cursor	COPY
SHIFT COPY	Initiate cursor-edit mode	FUNC :
TAB	Move cursor to non-space	FUNC A
f0	Find a line number	FUNC Q
f1	Issue MOS command	FUNC W
f2	Load the text in a file	FUNC E
f3	Save the text to a file	FUNC R
f4	Find and replace a string	FUNC T
f5	Global count/replace string	FUNC Y
f6	Set marker	FUNC U
f7	Copy a block of text	FUNC I
f8	Send text to printer	FUNC O
f9	Restore old text	FUNC P
SHIFT f0	Toggle <CR> display	FUNC 1
SHIFT f1	Toggle insert/overtype	FUNC 2
SHIFT f2	Insert text from a file	FUNC 3
SHIFT f3	*** NOT USED ***	FUNC 4
SHIFT f4	Quit from the editor	FUNC 5
SHIFT f5	*** NOT USED ***	FUNC 6
SHIFT f6	Clear marker(s)	FUNC 7

SHIFT f7	Move a block of text	FUNC 8
SHIFT f8	Delete a block of text	FUNC 9
SHIFT f9	Delete the text	FUNC 0

Appendix E

Error numbers/messages produced by the compiler

The table below lists all of the error numbers that the compiler produces, and the messages that are associated with them. These messages are printed automatically when {$F+} compiler option is specified when using discs. Cassette users may not use this option, so the list below (which is duplicated on the reference card) should be consulted during compilation. Additional information is printed by specifying the {$<CTRL @>+} option in the first line of the source file.

1 Variable identifier expected.
2 Comma expected / missing parameter.
3 '.' expected.
4 ':' expected.
5 ';' expected.
6 Type mismatch.
7 '(' expected.
8 ')' expected.
9 '[' expected.
10 ']' expected.
11 Can't assign a real to an integer.
12 RHS not compatible with LHS type mismatch.
13 Bad statement start.
14 Not ISO-Pascal (use compiler option X+ to allow extensions).
15 Equals expected.
16 If INPUT or OUTPUT is used then it must be declared in program header.
17 Missing parameter(s).
18 Parameter can't be a packed var.
19 Missing semicolon.
20 For loop control variable must be declared in the variable declaration part of this procedure / function.
21 Assignment operator ':=' expected.
22 '..' expected.
23 Actual and formal parameters should both be either packed or unpacked.
24 A label was declared in this block but was not defined.
25 Hex number too large.
26 Variable too big for memory.

27 Too much code for code buffer, claim larger area using compiler option 'C'.
28 Set base type must be max 0 . . 255.
29 BEGIN expected.
30 Too many procedures (max 127).
31 Missing body of FORWARD proc/func.
32 DO expected.
33 Label not declared.
34 This label does not prefix a statement which is in the same statement sequence that contains the GOTO statement.
35 END expected / missing semicolon.
36 This label should prefix a statement at the outermost level of statement nesting in a block.
37 Label not declared in this block.
38 Label already defined.
39 Label already declared.
40 Label must be a sequence of digits 0 to 9999.
41 Array element selector is not the same type as the array's index type.
42 Unpacked array variable expected.
43 Component types of both arrays must be the same.
44 OF expected.
45 Packed array variable expected.
46 Can't pass a conformant array as a value parameter.
47 PROGRAM expected.
48 Can't pass a bound identifier as a var parameter.
49 Function result type mismatch.
50 Formal parameter is a procedure and actual parameter is a function or vice versa.
51 THEN expected.
52 TO expected.
53 Procedural/functional parameter expected.
54 UNTIL expected.
55 Can't alter the value of this variable because it is the control variable of an active FOR loop.
56 Control variable must be an entire variable ie not an array element or field of a record.
57 Too many digits.
58 Premature end of file.
59 Can only output integers in hex.
60 Too many parameters.
61 String parameter expected.
62 Undeclared identifier expected.
63 For loop initial & final values must be same type as control variable.

64 For loop control variable must be ordinal type.
65 Record's field identifier expected.
66 Can only assign value to current function identifier.
67 Current function identifier is only allowed on LHS of assignment.
68 Ordinal parameter expected.
69 Parameter must be a file variable.
70 Parameter must be a textfile.
71 Constant already specifies a variant part in this record.
72 Constant does not specify a variant.
73 Variant constant / tag-type mismatch.
74 Too many variant constants.
75 Pointer's base type must be record in order to have variant constants.
76 Formal parameters have the same conformant array type but the actual parameters are not of the same type.
77 Can only have variant constants if type pointed to is a record.
78 Set base type and IN operand are not the same type.
79 Real parameter expected.
80 Real / integer parameter expected.
81 Integer parameter expected.
82 Text file variable expected.
83 Filename string expected.
84 Temp files do not have file-names.
85 Can't have a file as a parameter to READ/WRITE.
86 File and parameter type mismatch.
87 Can't read / write this type.
88 Only reals can have a decimal place.
89 File must be of type TEXT to do WRITELN/READLN.
90 Type mismatch between actual and formal parameter.
91 Procedure/function has no arguments.
92 File variable expected.
93 Bad filename.
94 Control variable threatened by nested procedure / function.
95 Procedural parameter list mismatch.
96 Function id is unassigned.
97 Structured types containing a file component cannot be assigned to each other.
98 File type must be TEXT to allow use of field widths.
99 Can't assign value to function parameter identifier.
100 Set of all tag-constants does not equal the set of all values specified by the tag-type.
101 Can't pass tag-field as var param.
102 A variable appeared in the program header but was not defined.
103 Too many stmt sequences (max 255).

104 Can't redefine identifier because it has been used earlier in this block.
105 No hex reals allowed.
106 Can only pack conformant arrays.
107 Case value must be ordinal type.
108 Index limits out of range.
109 Standard file already declared.
110 File variable expected.
111 Constant expected.
112 Can't sign non-numeric expressions.
113 Type mismatch between case constant and case expression.
114 Bad pointer type.
115 Type identifier expected.
116 Duplicate case constant.
117 Subrange limits must be scalar.
118 Upper and lower limits must be same type.
119 Low bound exceeds high bound.
120 Ordinal type expected.
121 Too many dimensions for interpreter.
122 Set member must have ordinal type.
123 Can't have file of file(s).
124 Set member must have an ordinal value of 0 to 255.
125 Unresolved pointer type.
126 Function type expected.
127 Digit expected.
128 Function type must be ordinal, real or pointer.
129 Illegal character detected.
130 Unexpected EOF in a comment or a string constant.
131 File already declared permanent.
132 Unresolved pointer base type.
133 Pointer base type identifier is not a type identifier.
134 Structured type expected.
135 Tag type expected.
136 Ordinal constant expected.
137 Field does not belong to this record.
138 Procedure or function id expected.
139 Sets are not of the same base type.
140 Procedure/function already declared.
141 Variant selector type does not match variant constant type.
142 Pointer type expected.
143 Permanent files must be declared in global variable section.
144 Packed conformant arrays must be single dimension.
145 Can't change this compiler option once it is set.
146 Component type mismatch.

147 Set members must have the same type.
148 Variable is not a file or pntr type.
149 Missing index / spurious comma.
150 Variable is not a record.
151 Variable is not an array.
152 Numbers must be terminated by a non alphabetic character.
153 Permanent file not declared in global variable section.
154 Decimal places field-width must be an integer expression.
155 Field-width must be integer value.
156 Can't assign a value to a conformant array bound identifier.
157 Can't have EOLN in string constants.
158 Can't have a file variable contained in a value parameter.
159 Illegal operation on these operands.
160 Index type mismatch.
161 Boolean type expected.
162 Can't use function id in this way.
163 Integer operands needed for this operation.
164 Procedure identifier has been used before its defining occurrence.

{ These are fatal errors and cause termination of the compilation }

165 Id table overflow (increase table size using compiler option 'I').
166 Too many nested records / procedures.
167 To compile using disc Pascal, use DCOMP <source> <object>.
168 Code and source filenames the same.

Appendix F
Interpreter error messages

Below are the error messages which are generated by the run-time system, ie the command line interpreter, the editor and the Pascal interpreter. The error numbers are given for information only; there is no way in which they may be accessed from Pascal.

F.1 Command line errors

The errors below are produced in response to commands typed when Pascal displays its '%' prompt.

Bad command **(Error 1)**

This is produced when an unknown Pascal command is typed and the cassette filing system is selected. Typing an unknown command when the disc or Econet is selected causes Pascal to treat the command as a filename to be RUN.

Bad mode **(Error 2)**

This is caused by typing a MODE command when there isn't enough memory left to support the mode specified without overwriting the memory file(s). It is also given when an attempt is made to call the editor in MODE 2 or 5.

No code **(Error 3)**

This is given in response to SAVE or GO when there is no memory code file present. See chapter 3 for a list of the commands that affect the memory code file.

No room **(Error 10)**

This is given when there is not enough room to LOAD a code file. Note that this is only given on 'fast' filing systems. When using tape, the error is not detected until the file has been loaded, so Pascal workspace may be overwritten.

No text **(Error 4)**

This is caused by an attempt to compile from memory when there is no memory text file present. See chapter 3 for a list of the commands that affect the memory text file.

F.2 Editor errors

The errors described below occur when the editor is in use.

Bad marking **(Error 9)**

An attempt was made to peform a 'block' operation with the wrong number of marks set. For example, an attempt to delete a block with zero or two marks set will cause this. Also, an attempt to insert text with marks set causes the error.

Field number **(Error 8)**

An attempt to access a field outside the range %0 to %9 in a replacement string causes this error. See chapter 2 for details of the find and global replace commands.

No room **(Error 10)**

When the editor does not have enough space to perform an operation this may be given, for example trying to load or insert a file that is too long, trying to copy a block when there isn't enough space, trying to replace strings with much larger ones.

Not found **(Error 5)**

This occurs when a line number given in response to a 'goto line' command or a pattern given in response to a 'find and replace' command cannot be found.

Syntax error **(Error 6)**

A pattern or replacement not conforming to the definitions given in chapter 2 was specified. Remember that all special characters have to be escaped with '\' whether they have a special meaning in a particular context or not.

Too complex **(Error 7)**

This occurs when a search pattern contains too many repeated patterns prefixed by a '*'. The maximum number of such sub-patterns is seven. Very long patterns also cause this error.

F.3 Errors caused by Pascal programs

The errors presented below occur during the execution of Pascal programs. If the program was compiled with debugging information turned on, the line number and procedure name in which the error occurred is also printed.

Accuracy lost **(Error 26)**

This error is given when attempting to take the SIN or COS of an angle whose magnitude is greater than 2 to the power of 23 radians.

Bad MOD **(Error 23)**

This is caused by the second operand of MOD being negative. The Standard requires j in the term i MOD j to be greater than zero.

Bad number **(Error 27)**

This is caused by an illegal character appearing in a number read from a text file. The only legal characters in integers are '0' .. '9', '+' and '–'; reals may also include '.', 'E' and 'e'.

Case index **(Error 11)**

This error occurs when the expression of a CASE statement does not evaluate to any of the case labels (and there is no OTHERWISE clause).

CHR range **(Error 22)**

The argument to CHR must be an integer in the range 0 .. 255 (the ordinal range for characters).

Division by zero **(Error 21)**

If the righthand operand of /, DIV or MOD is zero, this error will be generated.

EOF **(Error 33)**

An attempt to read past the last element of a 'reset'-type file will cause this error to be generated.

Escape **(Error 17)**

This error is caused by the user pressing the ESCAPE key.

Field width **(Error 34)**

The print field width for numbers and strings must be between 1 and 255 characters. The number of decimal places for reals must be between 1 and 48. Values outside of this range cause a 'Field width' error.

File not found **(Error 28)**

An attempt to RESET an external file which does not exist using the extended form of the statement will cause this error.

Integer overflow **(Error 15)**

An integer expression which evaluates outside the range ±2 to the power of 31 will cause this error.

LN range **(Error 25)**

The argument of the LN function should lie in the range 0 to approximately 2E38, but the lower limit must not be zero.

-ve SQRT **(Error 24)**

An attempt to take the square root of a negative number will result in this error.

No room **(Error 10)**

This error is caused by more stack or heap being required than is available. For example, too many recursive procedure calls or dynamically-allocated variables would cause this, as would changing mode when there is insufficient memory to support the new mode.

Pointer **(Error 12)**

This is caused by referencing through a pointer whose value is nil or undefined.

Read only **(Error 31)**

This is caused by an attempt to WRITE or PUT to a file that was opened for input using RESET.

Real overflow **(Error 16)**

This is caused by a real expression evaluating to outside the range ±1E38 approximately.

Set value **(Error 14)**

This is caused by an attempt to use a value in a set expression outside of the range of the set's elements. The error is only given when range checking is enabled.

Subrange **(Error 13)**

This is caused by trying to assign a value to a variable declared as a subrange type when the value is outside the subrange. The error is only given when range checking is enabled.

Too many open **(Error 30)**

This error is caused by trying to open (using RESET or REWRITE) more than eight files. The number of open files may be reduced by making some files temporary, local ones.

Undefined buffer variable **(Error 20)**

This occurs when an attempt is made to PUT a buffer variable before it has been assigned to, or to read it after it has been put. For example:

```
rewrite(f);
put(f);
```

would cause the error: a f^:=<expression> should be inserted between the two statements.

Undefined file **(Error 19)**

This error occurs when an attempt is made to reference a file before it has been opened using RESET or REWRITE.

Undefined variable **(Error 18)**

This error is given when an attempt is made to access a variable before it has been assigned. The error will only be given if full or partial unassigned checking is enabled.

Write only **(Error 32)**

This occurs when an attempt is made to read from a file (using READ or GET) that was opened for output using REWRITE.

Appendix G
Notes on 'Pascal from BASIC'

This appendix details the differences between the Pascal variant described in *Pascal from BASIC* by P J Brown and the two versions of Acornsoft ISO-Pascal supplied in the package. Extensions allowed by the {$X+} compilation option are not discussed below; they are described in detail in chapter 5 of this manual.

G.1 Acornsoft level zero (ROM) Pascal

There are two differences between the Acornsoft ROM Pascal (ISO level zero) and the one described in Brown:

G.1.1 Halt

The procedure 'halt' mentioned in Brown is not standard and is not present in either of the Acornsoft ISO-Pascals.

G.1.2 Pack and unpack

The standard procedures 'pack' and 'unpack' are not recognised by the ROM compiler. This restriction was necessary in order to fit the compiler in 16K. The disc compiler fully implements 'pack' and 'unpack'. The ROM interpreter recognises all of the code that the disc compiler produces, so programs compiled with the disc-based system may be run using the ROM interpreter, even if level one features were used.

G.2 Acornsoft level one (disc) Pascal

The differences between the Acornsoft disc compiler (ISO level one) and the Brown Pascal are more major. The features present in the compiler which are not mentioned in Brown are:

G.2.1 Conformant array parameters

These allow programmers to pass arrays whose exact size is not known to procedures and functions. The precise specification of conformant array parameters is given in Appendix A, but the general technique is described with examples below.

Consider a procedure that converts a string variable to upper case:

```
procedure toUpper(var astring: packed array
                  [low .. high:integer] of char);
const
   factor = 32; { Difference between 'a' and 'A'}
var
   i : integer;
begin
   for i:=low to high do
     if (astring[i] >= 'a') and (astring[i] <='z')
      then
        astring[i] := chr(ord(astring[i])-factor)
end;
```

The format of the parameter astring is a typical conformant array declaration. It corresponds in a general way to the declaration of the array used as the actual parameter, but the bounds (low and high) are determined at each invocation of the procedure. The type of the index is also given explicitly in the parameter specification. 'low' and 'high' are neither variables nor constants, but 'bound identifiers'. They may be used in expressions within the procedure, but may not be assigned to. The program below uses toUpper:

```
program  test(input,output);
type
   str1 = packed array[1..10] of char;
   str2 = packed array[1..5]  of char;

var
   s1 : str1;
   s2 : str2;

procedure toUpper(var astring: packed array
                  [low .. high:integer] of char);
const
   factor = 32; { Difference between 'a' and 'A'}
var
   i : integer;
begin
   for i:=low to high do
     if (astring[i] >='a') and (astring[i] <=
                                        'z') then
        astring[i] := chr(ord(astring[i])-factor)
end;
```

```
begin
   s1 := 'a string o';
   s2 := 'short';

   write(s1);
   toUpper(s1);
   writeln(s1:20);

   write(s2);
   toUpper(s2);
   writeln(s2:20)
end.
```

In the first call to toUpper, low is 1 and high is 10. In the second call, low is 1 and high is 5.

The array parameter in toUpper was made a var one as the actual parameter had to be changed. In addition, the usual rule of using var parameters for large arrays (to obviate the copying of large amounts of data) also applies to conformant array parameters. One situation where value conformant array parameters are necessary is when the actual parameter is a constant. The only constant arrays allowed in Pascal are strings.

Consider this procedure:

```
function inint(prompt : packed
          array[low..high:integer] of char):integer;
:integer;
var
   i : integer;
begin
   for i := low to high do
      write(prompt[i]);
   readln(inint)
end;
```

A typical use of the call would be:

```
numEggs:=inint('How many eggs: ');
```

Notice that the prompt string is printed a character at a time, rather than in a single write. This is because although the actual parameter may be a string, the formal conformant array parameter can never be.

Multi-dimensional conformant array parameters are declared in much the same way as single dimensional ones. The difference is that the shorthand form of expressing arrays of arrays uses ; instead of ,. For example, the two declarations below are equivalent:

```
parm : array[l1..h1:char] of array[l2..h2:integer]
                                              of integer;
parm : array[l1..h1:char ; l2..h2:integer] of
                                                integer;
```

Not all the dimensions of an array parameter have to be conformant. For example:

```
list : array[first..last:char] of string;
```

The parameter called list is a conformant array of the non-conformant type string, which is assumed to have been declared earlier as a packed array of char with a lower bound of 1.

When an actual parameter is passed to a procedure using a conformant array formal parameter, the actual parameter must obey the rules of conformability. These are:

(a) The index type of the real parameter must be compatible with the type given in the conformant array declaration.

(b) The index bounds of the real parameter must lie in the range allowed by the type given in the conformant array declaration.

(c) Arrays must be packed, or both must be unpacked.

(d) The component types of the arrays must be the same.

For multi-dimensional arrays, the rules must be obeyed for each conformant subscript.

G.2.2 Procedural and functional parameters

It is possible to have parameters that are functions or procedures, in addition to normal scalar and structured types. Suppose a function must be written which calculates the derivative (slope) of a given function. The function whose derivative is required may be passed as a parameter:

```
function dyByDx(function f(x:real):real;
                              x:real):real;
const
  epsilon = 0.0001;
begin
  dyByDx := (f(x+epsilon) - f(x))/epsilon
end;
```

A typical call to dyByDx would be:

```
writeln(dxByDx(func1,23);
```

func1 is the name of a function which is being passed as a parameter to dyByDx. The second parameter 23 is passed to x, the second parameter of dyByDx, as usual.

It can be seen from the example that a procedural or functional parameter is declared in exactly the same way as a procedure or function heading. The parameters of the procedural or functional parameter are present only as place holders so that the compiler can check that it is called properly in the body of the main procedure. The identifiers used have a very small scope and may be used elsewhere in the main procedure definition.

Note that it is not possible to use built-in functions or procedures as actual functional/procedural parameters. If, for example, the built-in function sqrt must be passed as a functional parameter, a user-function must be defined thus:

```
function mySqrt(x : real) : real;
begin
   mySqrt=sqrt(x)
end;
```

Then mySqrt may be passed as a functional parameter to a function such as dyByDx above.

G.2.3 Variant forms of new and dispose

Consider the declarations below:

```
type
    smallInt = 1..3;
    myRec =
    record
       next : ^myRec;
       tipe : smallInt of
          1 : (x,y,z : real);
          2 : (a,b,c : integer);
          3 : (case select : boolean of
                   false : (q : char);
                   true  : (r : set of 1..10)
               )
    end;
var
    pRec : ^myRec;
```

A new record of type myRec may be created with the statement:

```
new(pRec);
```

This will allocate space for the maximum possible size that the record could occupy (which is 21 bytes when pRec^.tipe is equal to 1). However, in some circumstances the program knows that this variant will never occur, and only space for the smaller fields is required. For example, if a record was created in which pRec^.tipe=3 and prec^.select=false, then only 8 bytes would be needed (two for next, two for tipe, one for select and one for q). It would be wasteful for the compiler to allocate 21 bytes when only a maximum of 8 was required.

In cases such as the one described above it is possible to force the compiler to allocate space for particular values of variant fields. For example, the statement:

```
new(pRec, 3, false);
```

would force the compiler to allocate space for the minimum-sized variant record. The form of the extended new statement is:

```
new(pointer, C1, C2, ... Cn)
```

where C1, C2, . . . Cn are the case constants of successive case parts of the record-type declaration. They must appear in the order of the case parts in the type declaration and if any are omitted from the end the compiler will choose the largest value. For example:

```
new(pRec, 3);
```

will allocate enough space for the field r (ie 32 bytes).

If a record is created using the extended form of new, it must be destroyed using the extended form of dispose with the same parameters, for example the last example would be de-allocated with:

```
dispose(pRec, 3);
```

Appendix H
Further reading on Pascal

Those wishing to find out more about Pascal may find the following books of interest:

Standard Pascal by D Cooper, published by WW Norton Co, New York

Pascal User Manual and Report by H Jensen and N Wirth, published by Springer-Verlag, New York

Software Tools in Pascal by B W Kernighan and P J Pauger, published by Addison-Wesley, Reading, Mass

Algorithms + Data Structures = Programs by N Wirth, published by Prentice-Hall, Englewood Cliffs, NJ

Index

ISO-Pascal

on the BBC Microcomputer and Acorn Electron

About this book

This book is the reference manual for Acornsoft ISO-Pascal. It explains the steps necessary to create and compile Pascal programs on the BBC Microcomputer or the Acorn Electron, and describes the Acorn extensions to the language.

The manual *Pascal from BASIC*, which is also contained in the Acornsoft ISO-Pascal pack, is a comprehensive tutorial course in Pascal for those familiar with BASIC.

ISO-Pascal is a very suitable language for education as well as for general purpose use because of its structured format which encourages good programming practice. It is compiled to a compact intermediate code which runs faster than interpreted languages such as BASIC.

About the author

Pete Cockerell graduated in computer science at the University of Kent at Canterbury and joined Acornsoft in 1983 as a technical author.

Acornsoft Limited, Betjeman House, 104 Hills Road,
Cambridge CB2 1LQ, England. Telephone (0223) 316039

Note: British Broadcasting Corporation has been abbreviated to BBC in this publication.

ISBN 0 907876 97 8

SBD18