S-Pascal

on the BBC Microcomputer and Acorn Electron

PAUL FELLOWS

S-Pascal

on the BBC Microcomputer and Acorn Electron

About this book

This book describes all the features of the Acornsoft S-Pascal system and explains how to use it, It provides a complete introduction to S-Pascal and assumes no previous knowledge of the language. Listings of all the example programs included in the Acornsoft S-Pascal pack are also given in the book.

S-Pascal contains a subset of Pascal - one of the most popular programming languages available today.

Acornsoft S-Pascal supports integer, character and Boolean types, as well as multi-dimensional arrays. It is block structured and completely recursive.

Since S-Pascal compiles directly to 6502 machine code it is also eminently suitable for writing small fast utilities. The library routines provided facilitate the use of graphics and allow operating system commands to be accessed from within an S-Pascal program.

Acornsoft Limited, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, England.
 SLD14

Contents

1 Introducing the S-Pascal compiler

1.1 About the language

S-Pascal contains a subset of Pascal - one of the most popular programming languages now available. This subset was developed as a teaching language and provides an excellent introduction to structured programming. Each program split up into blocks. Variables, procedures and functions have to be declared :€-fore they can be used. In addition S-Pascal does not contain a GOTO statement which is the bane of any self-respecting programmer's life. Hence the language encourages the programmer to think about the structure of a program and what it needs to contain before he sits down to write it. The structure also makes a program easy to read and understand.

1.2 Who this pack is for

The Acornsoft S-Pascal pack was designed for people who know little or nothing about Pascal but are familiar with BASIC. It allows small programs (up to about 3.25K for the BBC Microcomputer and 1.25K for the Acorn Electron) to be written, compiled and executed. At the same time it gives comprehensive error messages to help the user debug his or her programs. The programs can be edited using the BASIC editor. For the more adventurous programmers the use of graphics is allowed.

1.3 What a compiler does

The only instructions which a computer can understand are machine code instructions. There are two ways in which a computer can be made to accept instructions written in a high-level language such as BASIC or S-Pascal. The method used for BASIC on the BBC Microcomputer and the Acorn Electron involves 'interpreting' the commands directly. A program to do this is contained in the BASIC ROM. It takes the instructions of the BASIC program, one at a time and calls the relevant machine code routines, which it provides, to perform required operations. This means that the machine needs to have the interpreter present all the time the program is running. The Acornsoft implementation of S-Pascal uses a different approach. The S-Pascal package contains a compiler which reads the whole S-Pascal program and produces an equivalent program in machine code. This machine code program can then be run directly

without the presence of the S-Pascal program or the Acornsoft S-Pascal compiler.

Compilers have certain advantages over interpreters. A compiler produces machine code which can be used to help teach the user about assembly language programming, whereas an interpreter does not. Also compiled programs run very quickly since only the machine code has to be executed. Interpreters are slower since they have to read each line of text and try to understand it every time the line is encountered. Since compilers produce a complete machine code version of the program, the compiler itself doesn't need to be in the computer's memory when the program is executed. Hence this gives more free work space for the program. If this S-Pascal compiler was instead an interpreter it would have to be present all the time and this would prevent the user from using the higher resolution graphics modes

2 Using S-Pascal

You will have purchased Acornsoft S-Pascal on either a cassette or a disc. In either case the system contains the following files:

S.PASCAL
E.FIB
E.BASES
E.RANDOM
E.HANOI
E.DIAMND1
E.DIAMND2
E.MOIRE

When using the Acornsoft S-Pascal package there are several stages you need to go through. The first one is to write a program in the S-Pascal language or alternatively use one of the example programs provided in the package. S-Pascal programs are referred to as source programs.

The source programs then have to be compiled using the S-Pascal compiler. This generates the machine code instructions which the computer will subsequently use to do its calculations etc. The machine code produced by a source program is known as the object code.

Finally, to run the program the object code is executed by the computer.

To tell the computer what you want to do next, for example edit a source program or execute some machine code, you must give it certain commands. These commands begin with an asterisk (*) and can be in upper or lower case. All the commands which it recognises and instructions on how to load the compiler are given below.

2.1 Loading the compiler from cassette

To load the compiler from cassette, place the cassette tape in the cassette recorder and make sure it is fully rewound. Type

```
CHAIN "S.PASCAL"
```

and press RETURN. The 'Searching' message should appear on the screen as you do this. Now press the PLAY button on the cassette recorder and

wait for the program to load. Loading will take about five minutes.

When loading is complete the heading

```
S-PASCAL
>
```

will be displayed on the screen.

2.2 Loading the compiler from disc

To load the compiler from disc, place the disc in the disc drive and close the hatch. Acornsoft S-Pascal is loaded by means of an 'AUTO-BOOT', and this is executed as follows:

1. Press SHIFT

2. While holding down SHIFT, press and release BREAK

3. Release SHIFT

When loading is complete the heading

```
S-Pascal
>
```

will be displayed on the screen.

Note that the disc will work in drive 0 of either 40 or 80 track disc drives.

2.3 Entering a source program

In order to enter a source program, type

```
*NEW
```

Line numbers will appear automatically and the text of the program can be typed in. pressing RETURN at the end of each line. When the program is typed in completely press ESCAPE. The source program can be listed and edited at this stage using the standard BASIC editor.

2.4 Compiling a program

To compile a source program type

```
*COMPILE
```

The program will then be compiled and the object code produced will be listed.

If there are any syntax errors in the program the compiler will stop at the relevant place and print out an error message.

2.5 Editing the source program

If you wish to list your program, eg for editing, type

```
*EDIT
```

2.6 Executing the object code

To execute the machine code generated by the compiler, type

```
*GO
```

2.7 Loading a source program

To load a source program from disc or cassette, type

```
*EDIT
```

This sets the value of PAGE to the correct value for a source program to be entered. Then type

```
LOAD "<filename>"
```

This will load a source program into the memory so that it is ready either to edit or compile.

2.8 Saving the source program

To save the source program, type

```
*EDIT
```

This will list the program on the screen. Then type

```
SAVE "<filename>"
```

Any filename can be used but it must obey the same rules that apply to BASIC program names.

2.9 Saving the object code on the BBC Microcomputer

The object code can be saved to the filing system by typing

```
*SAVE <filename> 1B00 2F00 1F00
```

This saves the block of memory from the address &1B00 to the address &2F00. The execution address of the file is set up as &1F00.

2.10 Saving the object code on the Acorn Electron

The object code can be saved to the filing system by typing

```
*SAVE <filename> 1100 1F00 1100
```

This saves the block of memory from the address &1100 to the address &1F00. The execution address of the file is set up as &1100.

2.11 Deleting a program

To delete the current source program from the computer's memory, type

```
*NEW
```

This will allow you to enter a new source program.

2.12 Pressing BREAK

Usually pressing BREAK will reset the system. The compiler will still be present as will the source code. Any object code that has been generated will also still be in the computer's memory. The heading

```
S-Pascal
```

```
>
```

should appear on the screen and the system may be used with the commands described above.

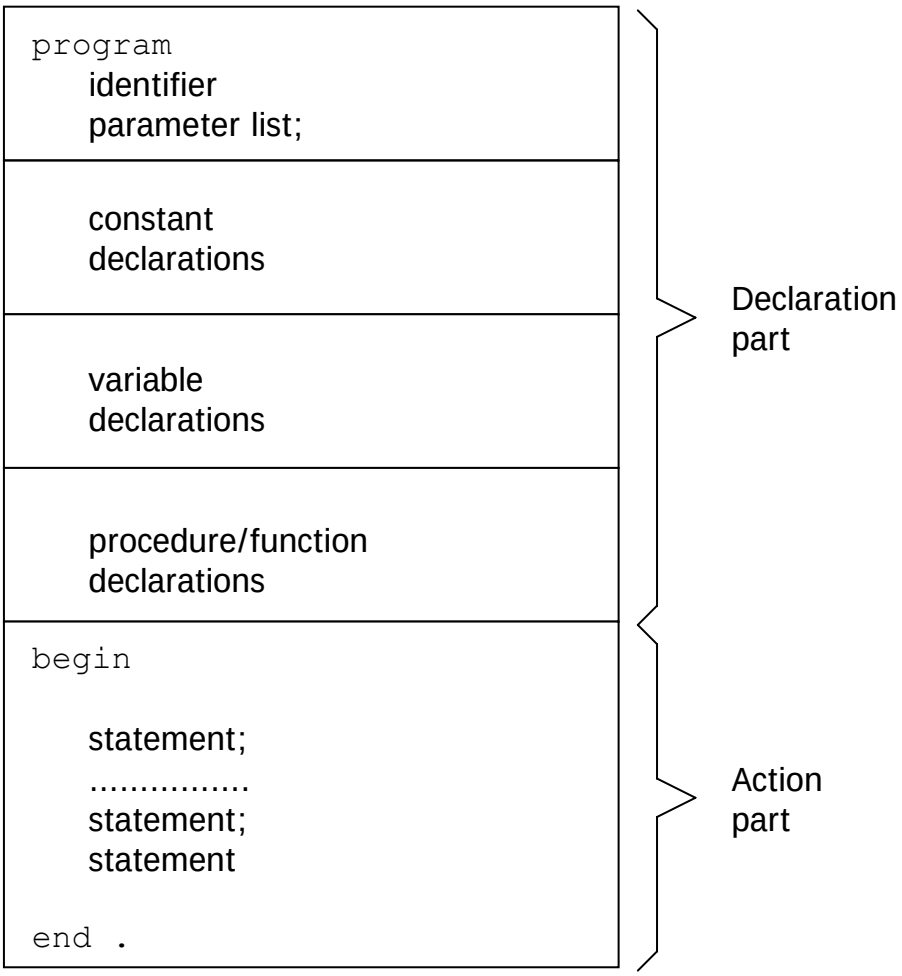
If, however, high resolution graphics modes have been used then pressing BREAK will still produce the heading

```
S-Pascal
>
```

but the source code and compiler will have been corrupted. The object code will not have been lost and can still be executed using `*GO` or saved as described above.

3 Programming in general

3.1 Program structure

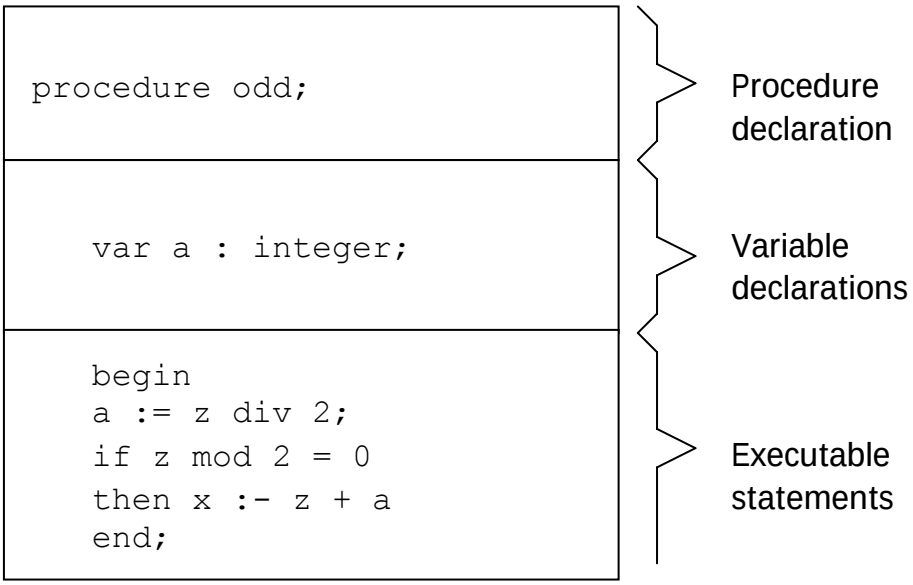


Structure of an S-Pascal program

S-Pascal contains several keywords, ie words in programs which the compiler recognises and acts upon in a special way. When writing S-Pascal programs these keywords should be in lower case. All programs must start with the word 'program' and must be terminated with a full stop. After the word 'program' the name of the program should occur, followed by a parameter list contained in curved brackets. This list must contain the word 'output' since it is assumed that every program will output results, and it should contain the word 'input' if the program is to input data while it is running. These are the only two parameters allowed. The body of the program is made up of a 'block'. This consists of a series of declarations (the declaration part) followed by the executable statements of the

program which occur in a 'compound statement' (the action part). This compound statement commences with the word 'begin' and terminates with the word 'end'. Between these words one or more statements separated by semicolons may be used.

One type of statement is the procedure call. Procedures are also blocks, ie they consist of declarations followed by executable statements, eg



4 Data types and structures

4.1 Data types

The executable statements in a program operate on variables and constants. All the variables and constants used must be declared at the beginning of the block in which they are to be used.

Variables

The declaration of variables introduces the names of 'identifiers' of the variables and fixes their type. The identifier can consist of a consecutive string of up to 15 lower case letters or digits, although it must begin with a letter and must not be an S-Pascal reserved word. Standard ISO-Pascal allows upper case letters as well, although by convention lower case is normally used. The use of lower case letters has been enforced in S-Pascal for identifiers and keywords to avoid confusion with BASIC keywords.

The simple data types which are available are integers, characters and the Boolean truth values. A single variable can be declared as follows:

```
var x : integer;
```

This declares an integer variable with the identifier 'x'.

When declaring several variables the word 'var' is only required once. All the identifiers of the same type are separated by commas, and the different types are terminated by semicolons.

Example

```
var  x, left, right   : integer;
     initial          : char;
     answer           : boolean;
```

Note that 'character' is always shortened to 'char' and 'Boolean' is entered as 'boolean' but 'integer' remains unchanged.

Later in the program these variables can be assigned a value. An integer can be any whole number in the range -32768 to +32767 or alternatively one of the pre-defined constants: '`maxint`' (the maximum integer value allowed) or '`minint`' (the minimum integer allowed). A character may be any alpha-numeric or control character, and must be entered enclosed in

single quotes. A Boolean has one of two values, either 'true' or 'false'. Trying to assign an incorrect value, for example to set a variable to the value 27 when it was declared to be a Boolean, will result in the compiler reporting a 'type mismatch' error during the compilation.

Constants

The value of a constant is fixed by the declaration, eg

```
const height = 10;
      weight = 100;
```

Once declared, the value of a constant cannot be altered throughout the program.

In addition to decimal numbers, constants can be declared as hexadecimal values. There are base 16 numbers, made up of hexadecimal digits, ie 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. To distinguish a hexadecimal number from a decimal one it is preceded by an ampersand (&) eg

```
const age = &2A;
```

is equivalent to

```
const age = 42;
```

Note: The use of hexadecimal notation is an extension to Pascal which has been included to help people wishing to access routines in the machine operating system (see section 8.3, 'Calling external routines'). These routines are normally entered by their hexadecimal addresses.

4.2 Data structures

One-dimensional arrays

In addition to the simple data types described above it is possible to declare array structures. A one-dimensional array is essentially a list of data. Each item of data is accessed using the name of the array and an index, either an integer, character or Boolean value, to determine which entry in the list is wanted. Each item has to be of the same type, and this type has to be declared. In addition, during the declaration of the array, the range of values of the indices has to be given; the lowest and highest values are quoted, separated by two full stops and enclosed in square brackets, eg

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

In this example, the array identified by 'vec' has ten elements numbered 1 to 10. Each element is of type integer.

Later the array elements can be assigned a value, eg

```
vec [ 5 ] := 27;
vec [ 8 ] := -128
```

Note that ':=' is used to assign a value to a variable, whereas '=' is used to declare the value of a constant.

The following is also possible:

```
vec [ x ] := -53
```

where x is of type integer. However if x has a value outside the range 1 to 10 when the program is run this will cause an error due to the declaration above.

An example of an array with indices of type character is given below:

```
var v : array [ 'b'..'g' ] of integer;
```

This gives an array identified by 'v' containing six elements, each of type integer.

These array elements could be assigned a value as follows:

```
v [ 'c' ] := 23
```

If required, Boolean truth values can be used as indices to produce an array containing two elements, eg

```
var vector : array [ false..true ] of integer;
```

The indices must be in the order 'false' followed by 'true'.

Multi-dimensional arrays

It is possible to have multi-dimensional arrays in S-Pascal. These are an extension of one-dimensional arrays in that they are essentially tables of data rather than lists. A table can have several dimensions and hence each item of data is represented by the array name and several indices. The

indices need not be of the same type, but the type and range of each has to be declared. This is done in a similar manner to one-dimensional arrays but the index ranges are separated by commas, eg

```
var letter : array [ -10..10 , 'a'..'z' ] of char;
```

The array 'letter' contains 21 X 26 elements. Each could be assigned a value as follows:

```
letter [ -7, 'c' ] := '*'
```

4.3 The order of declarations

It has been shown that variables and constants have to be declared at the beginning of the block in which they are to be used. In addition, as will be seen later, procedures and functions have to be declared similarly. If several declarations have to be made in the same block then there is a particular order in which these declarations must occur. Any constants used must be declared first and then all the variables of either simple data types or arrays. Finally the procedures and functions should be declared although here the order is irrelevant and they may even be mixed.

5 Statements

Statements make up the action part of a program. When a program is run 'control' passes from one statement to the next. 'Control' is said to be at the part of the program which is currently being executed. A statement can have several effects - these include causing the control to move round a loop, as in the case of a '`repeat...until`' command, or move to another part of the program in the case of a procedure call.

There are nine types of statement which may be used in S-Pascal. Each of these is described below, and examples of their use are given. In addition, each description includes the statement's 'syntax'. This is the format which it must take, eg it might have to start with a certain keyword which must be followed by a variable etc. Where variables, statements or expressions are needed in a statement they are enclosed in angled brackets to show that the word 'variable' etc is not required but an actual variable must be used. Words not enclosed in brackets are the keywords which must be entered as shown, and similarly any punctuation shown must be used.

The syntax of an expression can also be seen in Appendix B where it is displayed as a syntax diagram.

5.1 The assignment statement

This is used to give a variable a value. The syntax of the statement is:

<variable> := <expression>

Essentially an expression is just a single constant or identifier, or a sequence of these separated by operators such as '*' and '+'. For a more complex syntax analysis of expressions see the diagram in Appendix B.

The variable and the expression must both evaluate to the same type.

Examples
Integer types

```
x := 1 ;
area := x * y
```

Character types

```
initial := 'a' ;
letter := initial
```

Boolean types

```
answer := true ;
result := (a > b) and c
```

If a variable is used in an expression without first assigning a value to it then it will take an undefined value.

5.2 The if statement

The syntax on an '`if`' statement is

`if` <expression> `then` <statement>

or

`if` <expression> `then` <statement> `else` <statement>

The expression must yield a Boolean result. If the result is true the '`then`' part is executed, otherwise the '`else`' part is executed (if it exists). Should there not be an '`else`' part then control passes to the next statement.

Example

```
if a > b then x := x + 1
         else x := x - 1
```

In the above example the statements in the '`then`' and '`else`' parts are both assignments. However it is possible to use any of the nine statement types in these places. Thus the following is allowed:

```
if a > b
then if b > 0
     then x := x + 1
     else x := 0
else x := x - 1
```

In this example the '`if...then...else`' statements have been 'nested', ie one occurs within the other.

Note: Nesting '`if...then...else`' statements has different effects in BASIC and S-Pascal. In BASIC the first '`else`' statement is executed if

either of the expressions yields the value 'false'. Hence if the equivalent statement was written in BASIC and 'a' was given the value 4, 'b' the value 5 and 'x' the value 6 this would leave 'x' with the value 0. In contrast assigning these values to the variables in the Pascal statement yields the value of 5 for 'x' since the second or outermost '`else`' is associated with the outermost '`if`' and the first or innermost '`else`' is associated only with the innermost '`if`'.

A further example of an '`if`' statement is shown below. In this case the statements are both 'compound statements'.

```
if a > b
then begin
     x := x + 1 ;
     y := y + 1 ;
     z := 0
     end
else begin
     x := x - 1 ;
     y := x * y
     end
```

5.3 The while statement

The syntax of this is as follows:

`while` <expression> `do` <statement>

The expression must evaluate to the type Boolean. If its value is 'true' then the statement following the '`do`' is executed. Then control loops back and the test is made again. Hence the statement is repeatedly executed until the expression yields the value 'false'. When this occurs control is passed beyond the '`do`' statement to the next statement in the program.

Examples

```
while n < 100 do n := n + 1
```

```
while n < 100 do
     begin
     n := n + 1 ;
     x := x + n
     end
```

5.4 The for statement

This statement has the following syntax:

```
for <variable> := <expression> to <expression>
                                  do <statement>
```

The variable following the '`for`' is known as the control variable. It is assigned the value of the first expression and is then compared with the value of the second expression. If the control variable is less than or equal to the second expression, the statement following the statement '`do`' is executed. The control variable is then incremented and the comparison is made again. Hence the body of the statement is executed once for each value of the control variable, as it is incremented in steps of one, between the values of the two expressions.

An alternative form of the '`for`' statement is as follows:

```
for <variable> := <expression> downto <expression>
                                  do <statement>
```

In this case the body of the statement is executed while the control variable is greater than or equal to the limit expression and the control variable is decremented at each stage.

The control variable can be of any simple type, ie integer, character or Boolean.

Examples

```
for initial := 'a' to 'g' do x := x + 1

for x := -10 to a * b do
    begin
    y := y + 2 ;
    z := 4 * y
    end
```

When using characters the meaning of the word 'increment' may not be immediately obvious. Each character, however, is stored in the computer's memory as a number between 0 and 255. The code which the computer uses to represent the characters is the ASCII code (American Standard Code for Information Interchange). Hence 'increment' means going from a character with a lower ASCII value to one with a higher ASCII value. A full list of the ASCII codes is given in the Appendices of the *BBC Microcomputer User Guide* and the *Acorn Electron User Guide*.

For a Boolean control variable 'increment' means going from 'false' to 'true'.

In the example given above where an expression containing variables is used as the upper limit, the limit is evaluated only once at the start of the loop. Hence changing the value of 'a' or 'b' inside the loop will not affect the number of times the loop is executed, as this is determined only by their initial values, eg

```
i := 3;
for j := 1 to i do i := i + 1
```

This loop will execute three times and will leave 'i' with the value 6.

Note: This '`for`' statement differs from the one in BASIC in that if the control variable initially has a value greater than the second expression the statement is not executed. In BASIC it is always executed at least once since the 'test', ie the comparison of the control variable and the second expression, occurs after the execution of the statement.

5.5 The repeat statement

The syntax of this statement is as follows:

```
repeat <statement>
until <expression>
```

or

```
repeat <statement> ;
       <statement> ;
          .......
       <statement> ;
       <statement>
until <expression>
```

If several statements are used it is not necessary to form them into a compound statement by enclosing them in a '`begin...end`' block, because they are already enclosed by the '`repeat...until`'.

Initially the statements are executed, and then the expression is evaluated. The expression must evaluate to a Boolean. If the result is 'false' then the body of the loop is executed again and a new test is made.

When the result of the test is 'true' control passes to the next statement. Note that the body of the loop will always be executed at least once.

Example

```
repeat
    n := n + 1 ;
    x := x + n
until n > 100
```

5.6 The case statement

The syntax of this statement is as follows:

```
case <expression> of
<case label list> : <statement> ;
<case label list> : <statement> ;
.............................
<case label list> : <statement> ;
<case label list> : <statement>
end
```

Each case label list is a list of constants. These must be of the same type as the result of the expression. The expression is evaluated and the result is compared with each constant in torn. If a 'match' is found then the statement corresponding to the 'matched' constant is executed and control then passes to the statement following the 'end'. If no matches are found an error will result.

Example

```
case k of
(1)     : x := x + 1 ;
(2,3,4) : x := x - 1 ;
(5)     : x := 0
end
```

Note: In the '`case`' statement brackets are required around the case label lists. This is not standard Pascal necessary but is necessary in this implementation because of the BASIC system. Numbers which are not preceded by other symbols on the same line are stored by the BASIC editor as hexadecimal numbers or 'tokens'; this is because it assumes that they are references to the line numbers used by `GOTO` statements. The format of a tokenised number is such that it will not be recognised by the Pascal compiler. In addition, if the program is renumbered during editing,

eg to insert extra program lines, the case labels will also be renumbered. These problems are avoided by entering the list of numbers in brackets so that they will not be tokenised.

5.7 Procedure calls

A procedure statement is the call of a procedure. The procedure can be one of the library procedures or one which has been defined by the user elsewhere in the program. To call a procedure just its name is used and any parameters it may need, eg

```
write (x)
```

This calls the library procedure '`write`' which, if 'x' is of type integer, will print its value in decimal notation.

Note: Function calls, although they are very similar to procedure calls, are not statements. Functions return a value and so are used in expressions.

The library procedures and functions are described in more detail in chapter 6 (Library routines). Details on how to define and call your own procedures and functions can be found in chapter 6 (Procedures and functions).

5.8 The compound statement

A compound statement consists of a group of statements enclosed by '`begin`' and '`end`'. Wherever any of the statements already described can be used it is also possible to use a compound statement, eg

```
begin
sum := sum + n;
if n<>0
then numb := number + 1
end
```

This compound statement could be used to form the body of a '`while`' loop, eg

```
while numb < 100 do
begin
readln (n);
sum := sum + n;
if n <> 0
```

```
then numb := numb + 1
end
```

5.9 The null statement

In S-Pascal it is possible for a statement to be 'null', ie to contain no text at all. This is used primarily in 'empty' loops, eg

```
repeat until nextchar(x) = 'y'
```

where 'nextchar' is a user-defined function.

6 Procedures and functions

6.1 Declaring a procedure

To declare a procedure the following syntax is used:

```
procedure <identifier> <parameter list> ;
          <block> ;
```

The identifier must obey the same rules as an identifier used for variables. The parameter list contains the identifiers of all the parameters and their types. This list must be enclosed in curved brackets. However the syntax of the declaration is the same as that of variables, ie identifiers of the same type are separated by commas and those of different types by semicolons. If no parameters are required this list may be omitted.

Examples

```
procedure inc (x : integer) ;
     begin
     a := a + x
     end;

procedure printn (x,n : integer ; a :char) ;
var j : integer ;
     begin
     for j := 1 to n do
            write ('a') ;
     writeln (x)
     end;
```

The second example, when called, will print out n copies of the character 'a' followed by the value of 'x'. It will then go to the next newline. '`write`' and '`writeln`' are standard library procedures which can be found in chapter 8 (Library routines).

These procedure declarations occur in the declaration part of the program. When control reaches them their identifier and any parameters are noted and control then passes to the next declaration. Control does not pass to the body of the procedure until the procedure is called from the action part of the program.

6.2 Local variables and parameters

In the last example the variable 'j' was declared within the procedure and 'j' is referred to as a local variable. It is only possible to reference 'j' from within the block that forms the body of the procedure, any attempt to use 'j' outside this block will cause an error. The 'scope' of 'j' is therefore said to be the block in which it is declared. Local procedures may be declared in a similar manner.

The scope of a parameter is the same as the scope of a local variable, and thus it cannot be referenced outside the block in which it is declared.

6.3 Calling a procedure

To call a procedure such as '`printn`', declared in section 6.1, the following kind of statement should be used:

```
printn (5,6,'*')
```

The procedure should be called with the same number, and type, of parameters as in its definition.

6.4 Passing parameters

The parameters are said to be passed by value. This means that the values of the parameters given in the call of a procedure are passed over to the local parameters of the procedure definition. Hence in the above example the value '5' is passed over to the parameter 'x', '6' to 'n' and '`*`' to 'a'. However, any subsequent alteration in the values of these local parameters does not affect the value of the parameter outside this block.

Example

```
procedure inc (x : integer) ;
     begin
     x := x + 1
     end ;
........
j := 1 ;
inc (j) ;
write (j)
```

This will write out the value 1 since the procedure 'inc' will only affect the value of the local variable 'x' and will have no effect on the external variable 'j'.

6.5 Declaring a function

Functions are similar to procedures except that they return a value whereas procedures do not. Hence it is necessary to declare which type a function returns, eg

```
function inc (x : integer) : integer ;
     begin
     inc := x + 1
     end;
```

This declares a function called 'inc' which returns an integer value.

6.6 Calling a function

A possible call of the function is:

```
j := inc (j)
```

where 'j' is an integer.

Control returns from a call of a function when the '`end`' statement is reached. The function should include at least one assignment to the function identifier. The value returned is that obtained by evaluating the expression on the right hand side of the most recent assignment, eg

```
function smaller (x,y : integer) : integer;
     being
     smaller := x;
         if y < x then smaller := y
     end;
```

If this function was created as follows:

```
s := smaller (5,3)
```

then the most recent assignment to 'smaller' gave it the value 3, hence the value returned by the function call is 3.

If an assignment does not occur then control will return from the function call but the value obtained will be undefined, eg

```
function odd (x : integer) : integer ;
     begin
     if x mod 2 = 0
     then odd := 0
     end;
```

If 'odd(1)' is called then the body of the '`if`' statement will not be executed and control will reach the '`end`' of the block which forms the body of the function without an assignment to 'odd' being made. Control then returns but the value of 'odd' is undefined.

6.7 Recursive calls

Procedures and functions can call themselves recursively. Under certain circumstances procedures can call other procedures. See chapter 7 (Scope) for further details.

7 Scope

7.1 Definition of scope

The scope of a variable is the region of the program text in which its name may be used. For all names declared at the head of a block, their scope is from their declaration to the end of that block. If the block contains nested declarations of functions and procedures then the blocks which make up their bodies are also included in the scope of the names.

Example

```
program scope (output);

var x,y : integer;
procedure off(z : integer);
    var a : integer;
    begin
    a := 3;
    if z mod 2 = 0
    then x := a + 3
    end;                    (2)
begin
x := 3;
y := 2 * x;
odd(y)
end.                                (1)
```

In this example the scopes are as follows:

- The scope of x is block 1 and block 2.
- The scope of y is block 1 and block 2.
- The scope of z is block 2.
- The scope of a is block 2.
- The scope of odd is block 1 and block 2.

If a variable declared in an inner block has the same identifier as a variable declared further out, then all references to that identifier made in the inner block are to the most recently declared variable of the name.

Thus the scope of this variable declared on the outer level does not include the inner block. As an example consider the program above. If the variable 'a' was replaced by 'x' so that there were two separate variables both with the identifier 'x', the scope of one would be block 1 and the scope of the other block 2.

A similar situation applies to procedure identifiers. The scope of a procedure is from its declaration to the end of the block in which it was declared.

7.2 Forward reference

Since the scope of an identifier starts at its declaration it is not possible to declare a pair of mutually recursive procedures in the following way:

```
100 procedure a (.....);
110   begin
120   ............
130   b (.....);
140   ............
150   end

200 procedure b (.....);
210   begin
220   ............
230   a (.....);
240   ............
250   end;
```

The scope of 'procedure b' is from its declaration on line 200 until the end of the block. Hence the above example is not allowed because the forward reference on line 130 to 'procedure b' from the body of 'procedure a' does not lie within the scope of the declaration of 'procedure b'. However, mutually recursive routines can be declared as shown below:

```
100 procedure a (.....);
110 ............
120   procedure b (.....);
130     begin
140     ............
150     a (.....);
160     ............
170     end;
180   begin
190   ............
200   b (.....);
```

```
210    ............
220    end;
```

In this case the scope of 'procedure a' is from its declaration on line 100 onwards and that of 'procedure b' is from its declaration on line 200 until the end of that local block, ie the end of 'procedure a' at line 220.

7.3 Recursion and local variables

If a recursive procedure contains local variables, each call of the procedure brings into existence a new set of local variables. Each new set is distinct from those of any preceding call of the procedure. Thus the most recent manifestation of the local variables is in scope and none of the previous one are. However the values of the previous sets of local variables are preserved, and when a return is made from the recursive call these previous values become accessible again.

The best known example of a recursive function is the factorial function. This program calculates the factorial of a number which is entered from the keyboard:

```
 10 program factorial (input, output);
 20 var a:integer
 30 function fac(x:integer):integer;
 40   begin
 50   if x<=1
 60   then fac:=1
 70   else fac:=x*fac(x-1)
 80   end;
 90
100 begin
110
120   while true do
130   begin
140   readln(a);
150   writeln(fac(a))
160   end
170
180 end
190.
```

The function 'fac' defines the factorial of x to be:

```
x*(x-1)*...*2*1
```

Note that fac(0) = 1

and fac(1) = 1

All others are obtained using the recursion.

The '`while`' statement keeps on reading numbers as they are entered in from the keyboard, calculating factorials and printing the results until an error occurs, ie until ESCAPE is pressed or an arithmetic overflow is caused.

The factorial function could of course be calculated without using any recursion. However the function to do this is not so easy to read.

```
 30 function fact(x : integer ) :integer;
 40     var n,product : integer;
 50     begin
 60     product:=1;
 70     for n:=1 to x do
 80     product:=product*n;
 90     fact:=product
100     end;
```

Also this second example needs to declare two local variables, 'n' and 'product'.

8 Library routines

Library routines are a series of routines which are included to act as an interface between the user and the computer's operating system. These usually take the form of procedures and functions which may be used for several purposes, including inputting and outputting data while a program is being executed.

8.1 Input and output routines

read( ) - read number/character

This takes an arbitrary number of parameters of either integer or character type. When the routine is called the program stops and waits for the user to enter a line/lines of text. When entering a line of text DELETE deletes the last character on that line and CTRL U can be used to delete the entire input line. The line is terminated by pressing RETURN.

When a line of text has been entered the program computes a value for each of the parameters in turn, according to their respective types. If a parameter is of type character this simply means that the next character from the input line is returned. If the parameter is of integer type then a number, in decimal notation, is read. This may be preceded by leading spaces, which will be ignored, and by '+' or '-' signs. Once the spaces and signs have been dealt with the system expects the digits 0-9 and continues to incorporate these into the number until a non-digit is found. This non-digit should be a space or a RETURN character; if it is not then an error message will be printed. If the number of values entered on one line is less than the number of parameters then the program will wait for more values to be entered on the next line. Once values have been obtained for all the parameters the rest of the input line will be 'preserved'.

The maximum length of an input line is 127 characters. Note that although a character can be assigned the value corresponding to a space, it cannot be assigned that of RETURN. Also note that numbers must be in the range -32767 to +32767, the value of '`minint`' cannot be read in.

readln( ) - read number/character

This is the same as 'read' but once values have been obtained for all the parameters the rest of the input line will be 'thrown away'.

write( ) - write number/character/Boolean

This routine can take an arbitrary number of parameters of any type. Integer parameters will be printed as decimal numbers, characters will be printed directly and Booleans will be printed as the strings 'true' or 'false', In addition strings enclosed in single quotes may be used.

writeln( ) - write number/character/Boolean

This is identical to 'write' except that a carriage return/line feed will be produced after all the parameters have been output.

Note that if only a carriage return/line feed is required then 'writein' may be used without a parameter list.

8.2 Type conversion routines

ord( )

This takes one parameter of type character and returns the ASCII code of the parameter.

chr( )

This takes one parameter of type integer and returns the corresponding ASCII character.

8.3 Calling external routines

call( )

This statement needs four parameters, the first being a constant whose value is the address of the location which is jumped to, and the other three being expressions whose values are sent to the A, X and Y registers respectively. It is useful because it allows operating system routines to be accessed, eg

```
10 program mos (output);
20 const osbyte = &FFF4;
30 begin
40   write('THIS IS');
50   call (osbyte,0,0,0)
60 end.
```

When this program is compiled and executed the 'call' instruction will cause a jump to the location &FFF4 with the A, X and Y registers all containing the value 0. This causes the version name of the operating system to be printed, eg

```
THIS IS
OS 1.20
```

More advanced users may like to use this statement to call their own assembler routines, hence combining S-Pascal and machine code programs.

Note: This is not a standard Pascal library routine but has been added as an extension to allow access to the machine operating system.

9 Operators and expressions

9.1 Expressions

An expression is a sequence of constants and identifiers separated by operators, which has a value of type integer, character or Boolean.

The syntax of an expression can be found in Appendix B.

9.2 Operators

The full list of operators is printed below along with the type or types they act on and the type they return. Dyadic operators are those which take two operands, ie they need to operate on or compare two integers, characters or Booleans to give a result. Monadic operators act on just one operand.

Dyadic operators

Operator	*Types acted on*	*Type returned*
+	integer	integer
–	integer	integer
*	integer	integer
div	integer	integer
mod	integer	integer
and	Boolean	Boolean
or	Boolean	Boolean

Relational dyadic operators

Operator	*Types acted on*	*Type returned*
=	integer, character, Boolean	Boolean
<>	integer, character, Boolean	Boolean
>	integer	Boolean
<	integer	Boolean
>=	integer	Boolean
<=	integer	Boolean

Monadic operators

Operator	*Types acted on*	*Type returned*
not	Boolean	Boolean
-	integer	integer

A brief description of these operators now follows:

Dyadic operators

`+` Adds two integers together.

`-` Subtracts one integer from another.

`*` Multiplies two integers together.

`div` Gives the number of times that one integer will divide into another.

`mod` Gives the remainder left when one integer is divided by another.

`and` Compares two Boolean expressions, returning the value 'true' if both are 'true', and 'false' if both are 'false' or they are different

`or` Compares two Boolean expressions, returning the value 'false' if both are 'false', and the value 'true' if both are 'true' or they are different.

Relational dyadic operators

`=` Compares two integers, characters or Booleans, returning the value 'true' if they are the same, and 'false' if not.

`<>` Compares two integers, characters or Booleans, returning the value 'true' if they are different, and 'false' if they are both the same.

`>` Compares two integers, returning the value 'true' if the first is greater than the second, and 'false' if it is less or they are equal.

`<` Compares two integers, returning the value 'true' if the first is smaller than the second, and 'false' if it is greater or they are equal.

`>=` Compares two integers, returning the value 'true' if the first is greater than or equal to the second, and 'false' if it is smaller.

`<=` Compares two integers, returning the value 'true' if the first is less than or equal to the second, and 'false' if it is greater.

Monadic operators

`not` Inverts a Boolean expression, changing the value 'true' into 'false' and the value 'false' into 'true'.

`-` Negates the value of an integer.

9.3 Operator precedence

To determine the order in which the constants and identifiers in an expression are evaluated, operators are given a precedence to show how binding they are. An operator is said to be more binding than another if it is always performed first when the two are present in an expression. The value of the precedence is between 0 and 3; a lower number indicates that the operator is more binding.

Operators	*Precedence*
All relational operators	3
'+', '-', 'or'	2
'*', 'div', 'mod', 'and'	1
All monadic operators	0

For example in the case of the expression:

```
7 + 3 * 2
```

precedence of '+' = 2
precedence of '*' = 1

Therefore '*' is more binding so the expression is essentially:

```
7 + (3 * 2)
```

If the precedence of two operators is found to be equal evaluation then occurs from left to right, eg

```
3 > 5 = false
```

precedence of '>' = 3
precedence of '=' = 3

Therefore the expression is evaluated from left to right so the expression is essentially:

```
(3 > 5) = false
```

This will return the value 'true'.

9.4 Function calls

Function calls are evaluated immediately they are required, eg

```
function f (x : integer) : integer;
      begin
      a := a + 1;
      f := x + 2
      end;
```

If this is called as follows:

```
a := 4;
y := f (1) + a * 2
```

then 'y' will become 13.

If, however, it is called in the reverse order:

```
a := 4;
y := a * 2 + f (1)
```

then 'y' will become 11.

The example above has been included to illustrate the effects of using functions which alter variables other than those declared in the block of the function. As can be seen great care is needed when this is done since the order in which the function call and other expressions are evaluated affects the final result. In general it is bad programming style to do this because of the side effects which can occur.

When a function is called its parameters are evaluated from left to right before the body of the function is executed.

Note that division by zero or numeric overflow will give errors at run-time.

10 Error handling

10.1 Syntax errors

All syntax errors are handled during the first pass of the compilation. A single error will cause the compilation to fail. The line on which the error is detected will be printed, together with an arrow pointing to the approximate position of the error. This arrow will either point directly to the position or to a point beyond the error, since it corresponds to the part of the program the compiler was analysing when it decided that it was not able to proceed.

In the version for the BBC Microcomputer the actual error will be stated after this, eg

```
Error found near 'write' in:-
100  if x = 0 write ('zero')
                ↑
then expected
```

In the version for the Acorn Electron an error number will be printed instead of the error message. A reference card is included in the pack which relates this number to the actual error message, eg

```
Error 5 found near 'write' in:-
100 if x = 0 write ('zero')
               ^
```

10.2 Error messages

The error messages which can occur are listed below, along with the most likely causes for them being obtained. Some of the messages are given in upper case, this indicates that the error is not due to a syntax mistake by the programmer but occurs because of the space limitations imposed by the compiler.

Error number 1 - := expected
This message is obtained when an assignment statement was expected. The most common cause is using '=' instead of ':=' when assigning values to a variable.

Error number 2 - end expected
This occurs when the word '`end`' has been omitted from the end of a compound statement or '`case`' statement. Another common cause is when a semicolon is missed out between statements; since when the compiler has dealt with the first statement it expects there to be either a semicolon followed by another statement or the word '`end`'.

Error number 3 - to expected
This applies to errors in a '`for`' statement. It occurs when the word '`to`' has been missed out or sometimes when the expression preceding it is malformed.

Error number 4 - do expected
This applies to errors in a '`for`' statement or '`while`' statement. It occurs when the word '`do`' has been missed out completely or sometimes when the expression preceding it is malformed.

Error number 5 - then expected
This applies to errors in an '`if`' statement. It occurs when the word '`then`' has been missed out or sometimes when the expression preceding it is malformed.

Error number 6 - wrong no. args
This occurs when a call of a procedure or function is made and it is given e different number of arguments from the number with which it is defined.

Error number 7- file not allowed
This applies to errors in the initial parameter list. The only two files allowed are '`input`' and '`output`'.

Error number 8 - file used twice
This applies to errors in the initial parameter list. The files '`input`' and '`output`' should each appear at most once.

Error number 9 - Assignment to const
This occurs if a constant identifier has been used on the left-hand side on an assignment statement since the value of a constant cannot be altered in a program.

Error number 10- output file not declared
This applies to errors in the initial parameter list. All programs must declare '`output`'.

Error number 11 - <punctuation> expected
The actual item of punctuation which is missing is printed when this message is given. This applies to both the BBC Microcomputer and Acorn Electron versions. The error occurs when the compiler is expecting some punctuation, ie a colon, semicolon, comma, full stop, bracket or single quote.

Error number 12 - until expected
This applies to errors in a '`repeat`' statement. It occurs when the word '`until`' has been missed out or when a semicolon has been missed out between the statements inside the '`repeat...until`' loop.

Error number 13 - program expected
This occurs when the first word of the program being compiled is not '`program`'.

Error number 14 - Bad type
This occurs when an attempt has been made to declare a variable to be of a type other than 'integer', 'char', 'boolean' or 'array'. Normally this occurs because the word has been misspelt.

Error number 15 - identifier expected
This occurs when the compiler is trying to analyse an assignment statement and expected an identifier but didn't find one. Alternatively it will occur when the program hasn't been assigned a name.

Error number 16 - PROC/ARRAY SPACE
This occurs when the compiler has run out of work space in which to store information about user-defined procedures or arrays. The number of procedures which may be defined depends upon the number of parameters associated with each one, and the number of arrays depends upon the number of dimensions each one has.

Error number 17 - operator expected
This occurs when the compiler is trying to analyse an expression and expected an operator but didn't find one.

Error number 18 - Type mismatch
This occurs when a variable identifier has been assigned a value with a type other than the one with which it was defined.

Error number 19 - Bracket expected
This occurs when an unequal number of left-hand and right-hand brackets has been used, or when brackets are required but not used.

Error number 20 - TOO MANY VARS
This occurs when the compiler has run out of room in which to store the names of variables. Up to 30 are allowed to be in scope at any time.

Error number 21- STACK FULL
This occurs when the compiler has run out of stack space. This is normally due to the use of deeply nested brackets since the program uses a recursive method to analyse the syntax of expressions etc.

Error number 22 - structure error
This occurs if a full stop is used in the middle of a program instead of after the final '`end`'; alternatively it can occur if the compiler becomes corrupted.

Error number 23 - wrong no. dims
This occurs when an array variable is used and it is given a different number of indices from the number with which it was declared.

Error number 24 - TOO MANY LABELS
This occurs when the compiler has run out of space in which to store all the labels in the object code. This means that it cannot cope with the source program because it is too large and complicated - '`case`' statements in particular produce a large number of labels when compiled.

Error number 25 - name too long
This occurs when an identifier containing more than 15 characters has beer used.

Error number 26 - No such func/proc
This occurs when a function or procedure is called but hasn't been declared as such. Usually this is because the identifier was declared as a variable rather than a procedure or function.

Error number 27 - No such var
This occurs when the compiler encounters an identifier which has not been declared. Usually this is because the identifier has been misspelt.

Error number 28 - . . expected
This occurs during an array declaration when the range of the indices has not been declared properly. The lowest and highest indices must be given separated by two consecutive full stops.

Error number 29 - No such array
This occurs when an array element is used but the array hasn't been declared. Usually this is because the identifier was declared as a simple data type rather than an array.

Error number 30 - of expected
This applies to '`case`' statements or the declaration of arrays. It occurs when the word '`of`' has been missed out or is preceded by a malformed identifier.

Error number 31 - Bad array size
This applies to the declaration of arrays. It occurs when the range of the indices is given the wrong way round, ie the highest value rather than the lowest one is stated first.

Error number 32 - input file not declared
This applies to errors in the initial parameter list. Any program which attempts to use the library routine '`readln`' must declare '`input`' in the program heading.

Error number 33 - reserved word used
This occurs during the declaration part of the program if an attempt is made to use a reserved word as an identifier.

Error number 34 - name already used
This occurs during the declaration part of the program if the same identifier is used for two different entities when they would both have the same scope.

10.3 Other compile-time errors

Compiler corrupted
This message means that the amount of machine code generated by the source program is so great that it has filled the space allocated for it and started writing over the space which was occupied by the compiler, hence corrupting it. In this case the compiler will need to be reloaded and the source program will have to be made simpler so that it generates less object code.

10.4 Run-time errors

Numeric overflow

This occurs when a calculation has produced a result which is outside the range -32768 to +32767.

Div by zero
This occurs when a calculation has tried to divide a number by zero.

Index too high
This occurs when the value of an array index exceeds the upper limit specified in the array declaration.

Index too low
This occurs when the value of an array index is less than the lower limit specified in the array declaration.

Escape
This occurs when ESCAPE is pressed.

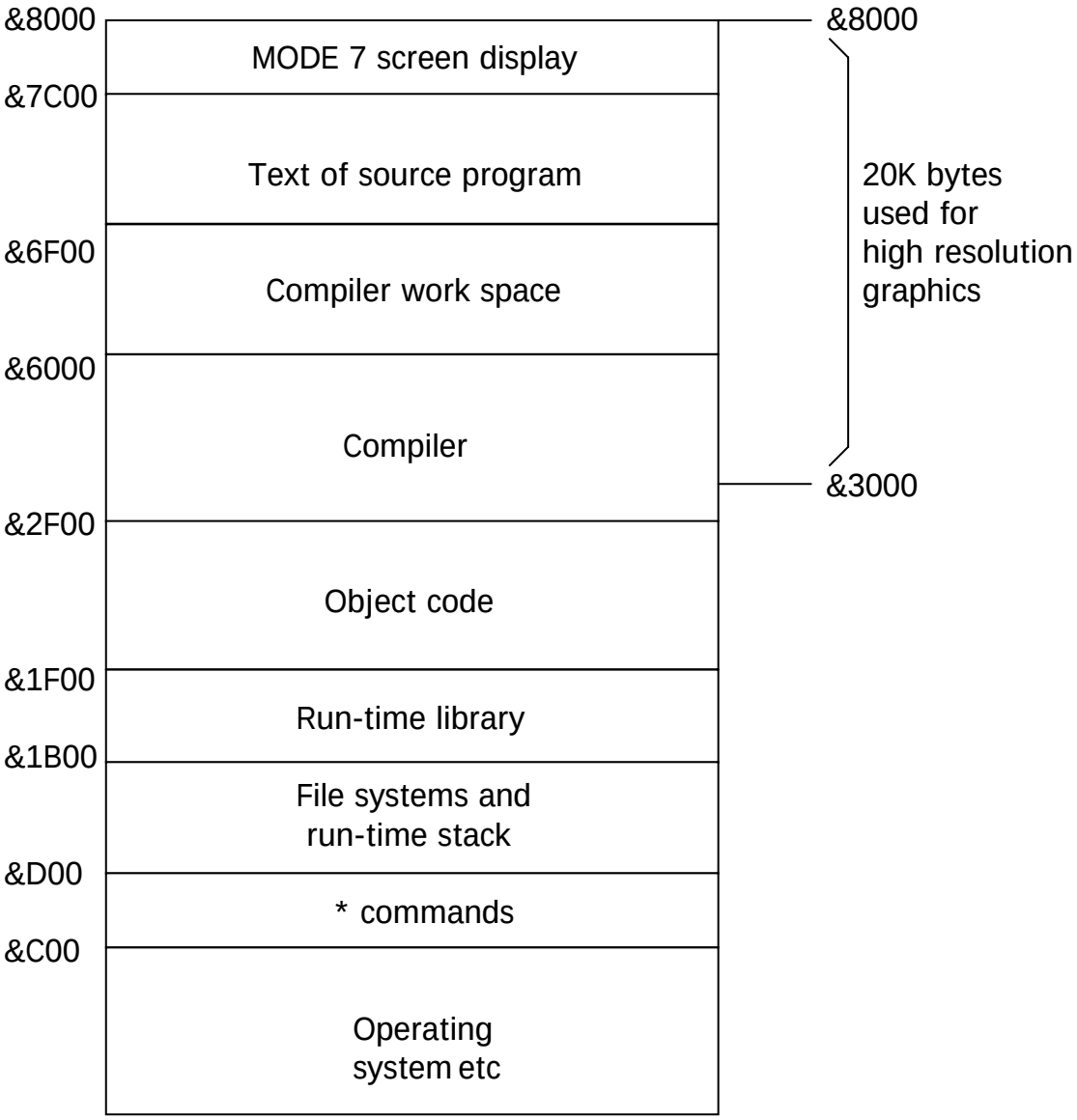
Bad digit
This occurs when '`read`' or '`readln`' is used with an integer parameter and a non-valid character is found in the input line where a digit or a space is expected

Bad case index
This occurs when the value of a case subject expression is not matched by any of the case labels.

11 The memory map

11.1 Memory allocation for the BBC Microcomputer

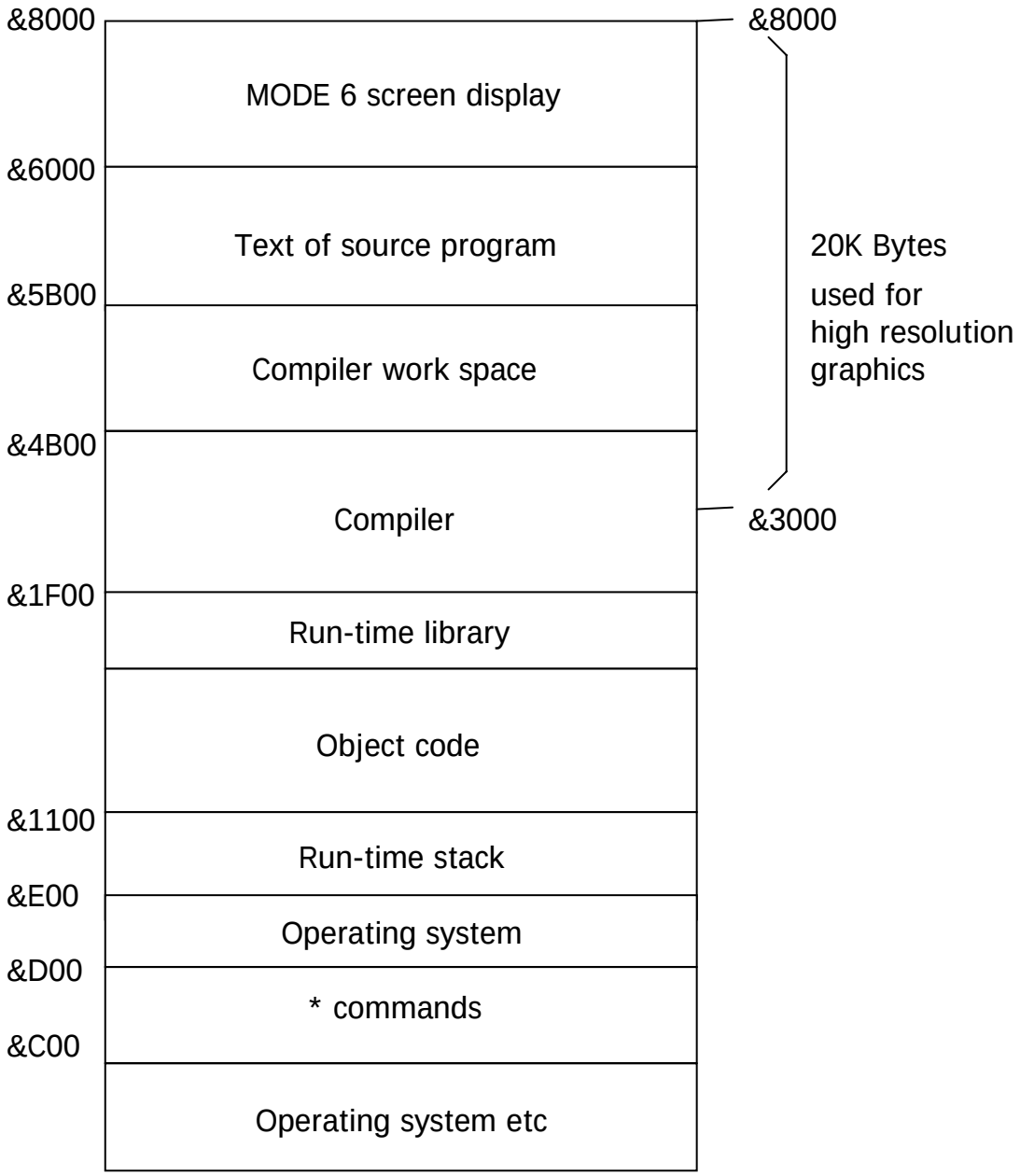
The memory of the machine is organised as follows:



Note that the use of high resolution graphics will corrupt the compiler and the source program. Graphics can be used but the compiler and source program will have to be reloaded at every stage.

11.2 Memory allocation for the Acorn Electron

The memory of the machine is organised as follows:



Note that the use of high resolution graphics will corrupt the compiler and the source program. Graphics can be used but the compiler and source program will have to be reloaded at every stage.

Appendix A

Example programs

This section gives the listings of the example programs contained in the Acornsoft S-Pascal pack. In addition a brief description of each program is given; the later ones demonstrate the use of graphics with the S-Pascal compiler. Note that when these programs are executed, changing into the graphics mode corrupts the compiler which is normally situated in this part of memory. Hence after they have been run the compiler will need to be reloaded. Alternatively the machine code generated by these programs can be saved so that they can be executed by means of a `*RUN` command without the compiler being present. For details on how to do this see section 2.9 (Saving the object code on the BBC Microcomputer) or 2.10 (Saving the object code on the Acorn Electron).

Note the layout of the programs. The indenting of the text and the splitting of the different blocks etc is not necessary, but it makes the programs much easier to read and understand. To put comments in the programs they should be enclosed in curly brackets and then these pieces of text will be ignored when the program is compiled.

Example 1- E.FIB

This first example calculates the first 21 Fibonacci numbers. Each number in this sequence is the sum of the two numbers preceding it. In addition the values of Fib(0) and Fib(1) are both set to be 1. Hence the sequence is:

1 1 2 3 5 8 ...

```
 10 program fibonacci (output);
 20
 30 var n,m : integer;
 40
 50 function fib(x : integer) : integer;
 60   begin
 70   if x <= 1
 80   then fib := 1
 90   else fib := fib(x-2) + fib(x-1)
100   end;
110
```

```
120 begin
130   for m := 0 to 20 do
140     begin
150     n := fib(m);
160     write('Fib(',m,') is ');
170     writeln(n)
180     end
190
200 end.
```

The main work of the program is performed by the function 'fib'. This sets fib(x) to be 1 if 'x' is less than or equal to 1, and to equal the sum of the previous two numbers in the sequence if 'x' is greater than 1.

The 'action' part of the program starts on line 120. Here the function 'fib' is called with each number in turn between 0 and 20, and the value returned is then printed out.

Example 2- E.BASES

This program allows numbers to be entered in any base and converted into any other base. For example to convert a number from decimal (base 10) to binary (base 2) type `10` when asked for the input base and press RETURN. Then the program will ask you for the number to be converted - enter any number, eg 20 followed by a space and press RETURN. Finally, enter the base in which you want the number displayed (in this example it is 2) and again press RETURN. The decimal number 20 will then be printed out in binary notation, ie 10100.

Note that the program will not work if you enter bases less than 2. Bases higher than 10 are allowed - for example with hexadecimal the extra numerals are represented by the letters A, B, C .

```
 10 program bases (input,output);
 20
 30 var khar : char;
 40     number,numeral,radix : integer;
 50
 60 function nextch : char;
 70   var ch :char;
 80   begin
 90   read(ch);
100   if ord(ch) > ord('9')
110   then ch:=chr(ord(ch)-7);
120   nextch:=ch
130   end;
```

```
140
150 procedure wrbas(numb,base:integer);
160   begin
170   if numb >= base
180   then wrbas(numb div base,base);
190   numb:=numb mod base;
200   if numb > 9
210   then write(chr(numb+ord('@')-9))
220   else write(chr(numb+ord('0')))
230   end;
240
250 begin
260 repeat
270   write('Choose input base --->');
280   readln(radix);
290   write('Enter number >');
300   khar:=nextch;
310   number:=0;
320
330   repeat
340     number:number*radix;
350     numeral:=ord(khar)-ord('0');
360     if numeral>=radix
370     then write('error')
380     else number:=number+numeral;
390     khar:=nextch
400   until ord(khar)=32;
410
420   writeln;
430   write('Choose output base -->');
440   readln(radix);
450   write('Value is : ');
460   wrbas(number, radix);
470   writeln;
480
490 until false
500
510 end.
```

The program contains one function, 'nextch', and one procedure, 'wrbas'. The function is used to read in the next character or number from the keyboard and convert it from the character representation '0' . . . '9' and 'A' . . .'Z' etc into a numerical value. Thus 'A' becomes the decimal value 10. The procedure prints out a number in any given base (except 'silly' bases less than 2). It uses a standard trick to do this. It calls itself re cursively, passing on the high part of the number each time until only a single 'digit' is left. This is then printed and the sequence of calls 'unwinds',

printing the appropriate 'digit' at each stage. Thus the highest 'digit' is printed first followed by the less significant 'digits'.

The main body of the program is in three parts. These respectively select the input base, read in numbers and print them in the chosen output base.

Example 3- E.RANDOM

This third example is a short program for generating 'random' numbers. Although the numbers are generated according to a sequence they are suitable for most applications which need a random element.

```
 10 program random (output);
 20
 30   var seed : integer;
 40
 50   function rnd(n :  integer) : integer;
 60     begin
 70     seed := (seed + 49) mod 256;
 80     rnd := seed mod n
 90     end;
100
110 begin
120   seed := 10;
130   repeat
140     writeln(rnd(100))
150   until false
160 end.
```

The only function contained in this program returns a 'random' value. The action part of the program causes random numbers to be printed out, each on a new line. These numbers will be printed out indefinitely since the 'until' expression will never return the value 'true'.

The range of numbers generated is determined by the parameter given to the function 'rnd'. This is the value of 'n' and the numbers lie in the range 0 to n-i. In this example 'n' was given the value 100.

Example 4- E.HANOI

This program illustrates how graphics may be used with S-Pascal programs. In this example the computer solves the classic 'Tower of Hanoi' problem in which there is a board with three pegs in it. A pile of discs is stacked up on one peg in order, smallest at the top and largest at the

bottom. The whole pile has to be moved onto one of the other pegs so that the discs are stacked in the same way. This is to be done by moving one disc at a time, placing it either on an empty peg or on top of a disc which is larger than itself.

```
 10 program hanoi (input,output);
 20
 30   var n,j  : integer
 40   one,two,three : integer;
 50
 60   procedure up(x:integer);
 70     begin
 80     case x of
 90       (1): one:=one+40;
100       (2): two:=two+40;
110       (3): three:=three+40
120     end
130   end;
140
150   procedure down(x:integer);
160     begin
170     case x of
180       (1): one:=one-40;
190       (2): two:=two-40;
200       (3): three:=three-40
210     end
220   end;
230
240   function height(x:integer):integer;
250     begin
260     case x of
270      (1): height:=one;
280      (2): height:=two;
290      (3): height:=three
300      end
310    end;
320
330 procedure bytes(x:integer);
340   begin
350   write(chr(x mod 256));
360   write(chr(x div 256))
370   end;
380
390 procedure box(col,x,y,size:integer);
400   begin
410   write(chr(18),chr(0),chr(col+128));
420   write(chr(24));
430   bytes(x-size);
```

```
440   bytes(y);
450   bytes(x+size);
460   bytes(y+20);
470   write(chr(16))
480   end;
490
500   procedure move(n,s,e:integer);
510   begin
520   box(0,s*400-200,height(s),n*16);
530   down(s);up(e);
540   box(n mod 7 +1,e*400-200,height(e),n*16)
550   end;
560
570   procedure hanoi (a,b,c,d:integer);
580   begin
590   if a<>0
600   then begin
610        hanoi(a-1,b,d,c);
620        move(a,b,c);
630        hanoi(a-1,d,c,b)
640        end
650   end;
660
670 begin {Main program}
680  readln(n);
690  write(chr(22),chr(2));
700  j:=n;
710
720  while j>0 do
730    begin
740    box(j mod 7 +1,200,(n-j+1)*40,j*16);
750    j:=j-1
760    end;
770
780   one:=n*40;two:=0;three:=0;
790
800   hanoi(n,1,2,3)
810
820 end. {Main program}
```

The procedures and functions up to line 550 are used to obtain the graphic displays.

The procedure 'up' increments the count of the number of discs on pile 'x' and the procedure 'down' decrements the count. The function 'height' returns the height of column 'x'.

The procedure 'bytes' sends the bytes of 'x' to the VDU drivers, with the low byte sent first and the high byte second.

The procedure 'box' draws a disc at the specified *x*- and *y*- coordinates. The size of the box is specified by 'size' and the colour it is to be drawn in by 'col'. The procedure is also used to delete discs that have been moved elsewhere by giving the 'col' parameter the value 0 so that the disc is drawn in black.

The procedure 'move' arranges for a disc to be moved from pile 's' to pile 'e' by calling the relevant procedures described above with the necessary parameters.

The other procedure 'hanoi' determines which disc should be moved and where it should be moved to.

The action part of the program inputs a value for the number of discs the user wants to start with. It then initialises everything by putting the screen into MODE 2, drawing a pile of that number of discs on the left-hand peg and initialising the heights of the piles. It then calls the procedure 'hanoi' to solve the problem and display the solution.

Example 5 - E.DIAMND1

This second graphics example program draws out a series of diamonds, each inside the others, by dividing each diamond into four others at every stage. Hence the diamonds become smaller and smaller until they eventually merge together.

```
 10 program diamond (input,output);
 20
 30   var m : integer;
 40
 50   procedure plot(p,x,y : integer);
 60     begin
 70     write(chr(25),chr(p));
 80     write(chr(x mod 256),chr(x div 256));
 90     write(chr(y mod 256),chr(y div 256))
100     end;
110
120   procedure square(x,y,s : integer);
130     begin
140     plot(4,x+s,y);
150     plot(5,x,y+s);
160     plot(5,x-s,y);
170     plot(5,x,y-s);
```

```
180     plot(5,x+s,y)
190     end;
200
210   procedure diamond(x,y,s : integer);
220     begin
230     if s >=m
240     then begin
250          s:=s div 2;
260          diamond(x+s,y,s);
270          diamond(x-s,y,s);
280          diamond(x,y-s,s);
290          diamond(x,y+s,s);
300          square(x,y,s)
310     end
320 end;
330
340 begin
350   write(chr(22),chr(0));
360   m:=400;
370
380   repeat
390     diamond(640,512,512);
400     write(chr(18),chr(0),chr(1));
410     m:=m div 2
420   until m<4;
450 end.
```

The procedure 'square' draws a square centred at (x,y) by moving to the position of one corner and drawing the four lines which make up the square.

The procedure 'diamond' then draws squares of half the size centred on its corners. Similarly each of these squares has four squares of half the size again on its corners. This continues until the square of minimum size is reached.

The action part of the program puts the screen into MODE 0, sets the initial size of the diamond and where it is to be drawn. Then it repeatedly causes more diamonds to be drawn within the others until the size reaches the minimum value of four.

Example 6 - E.DIAMIND2

This program is similar to the one in example 5, however it works in a different way.

```
 10 program diamond2 (input,output);
 20
 30   var m : integer;
 40      ch : char;
 50   procedure plot(p,x,y : integer);
 60     begin
 70     write(chr(25),chr(p));
 80     write(chr(x mod 256),chr(x div 256));
 90     write(chr(y mod 256),chr(y div 256))
100     end;
110
120   procedure du(d,x,y,s :integer);
130     begin
140     if s >= m
150     then begin
160          plot(4,x,y-s);
170          plot(5,x,y+s);
180          du(0,x,y+s,s div 2);
190          du(0,x,y-s,s div 2);
200          plot(4,x-s,y);
210          plot(5,x+s,y);
220          du(1,x+s,y,s div 2);
230          du(1,x-s,y,s div 2)
240          end
250     end;
260
270 begin
280   repeat
290     write(chr(22),chr(0));
300     write('size ');
310     readln(m);
320     if m<2 then write('silly')
330     else du(0,640,512,256);
340     readln(ch);
350   until ch <> 'y'
360 end.
```

Example 7- E.MOIRE

This final example program draws out a high resolution Moire pattern.

```
 10 program moire (input,output);
 20
 30   var n,m,x,y : integer;
 40
 50   procedure mode(x:integer);
 60     begin
 70     write(chr(22));
```

```
 80     write(chr(x))
 90     end;
100
110   procedure plot(p,x,y:integer);
120     begin
130     write(chr(25),chr(p));
140     write(chr(x mod 256));
150     write(chr(x div 256));
160     write(chr(y mod 256));
170     write(chr(y div 256))
180     end;
190
200 begin
210
220   mode(0);
230
240   n:=0;
250   m:=0;
260   x:=1280;
270   y:=1024;
280
290   while n<1280 do
300     begin
310     plot(4,n,m);
320     plot(5,x,y);
330     n:=n+12;
340     x:=x-12
350     end;
360
370   n:=0;
380   x:=1280;
390
400   while m<1024 do
410     begin
420     plot(4,n,m);
430     plot(5,x,y);
440     m:=m+12;
450     y:=y-12
460     end
470 end.
```

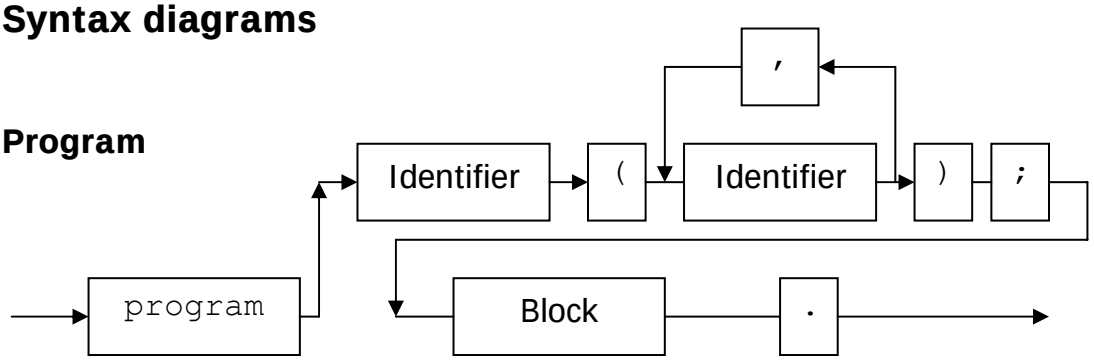
The action part of the program starts with the coordinates of the bottom left and top right-hand points of the screen and draws a line joining them. Then the *x*-coordinates are altered and a new line is drawn. This continues until the coordinates reach the other edge. This is repeated with the *y*-coordinates being altered instead of the *x*-coordinates so that the whole screen is covered.

The pattern obtained is due to the overlapping of the lines.

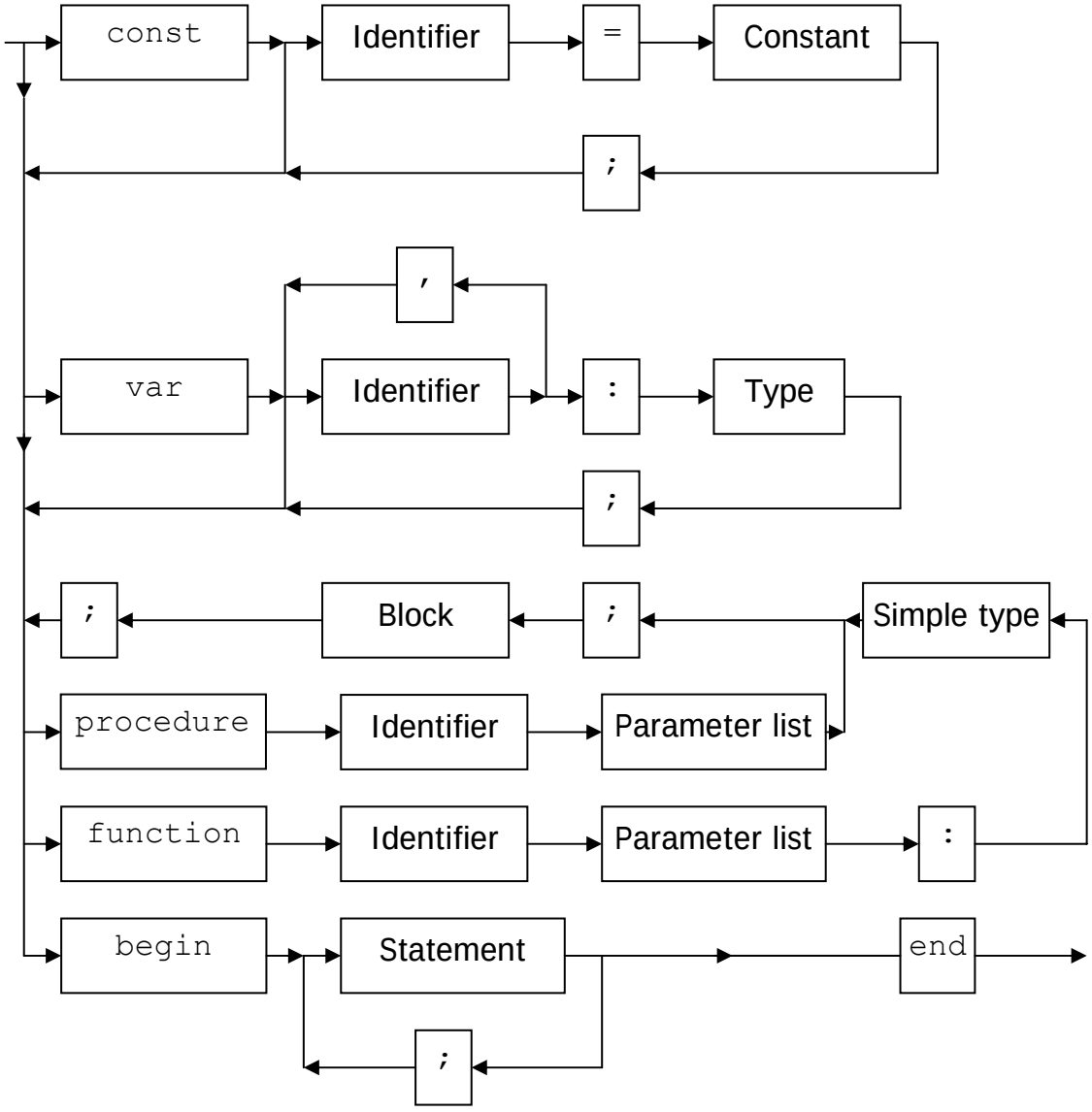
Appendix B

Syntax diagrams

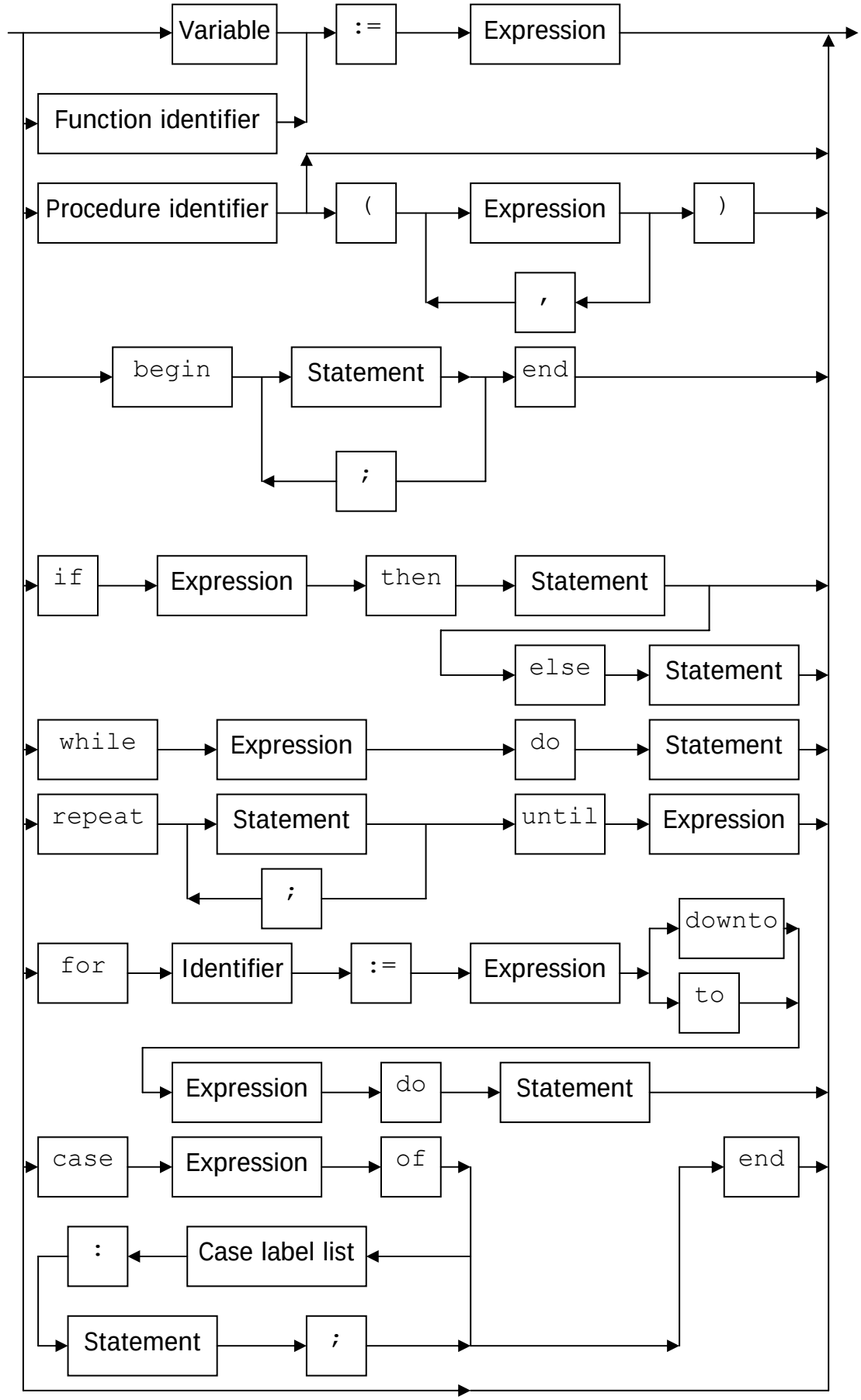
Program



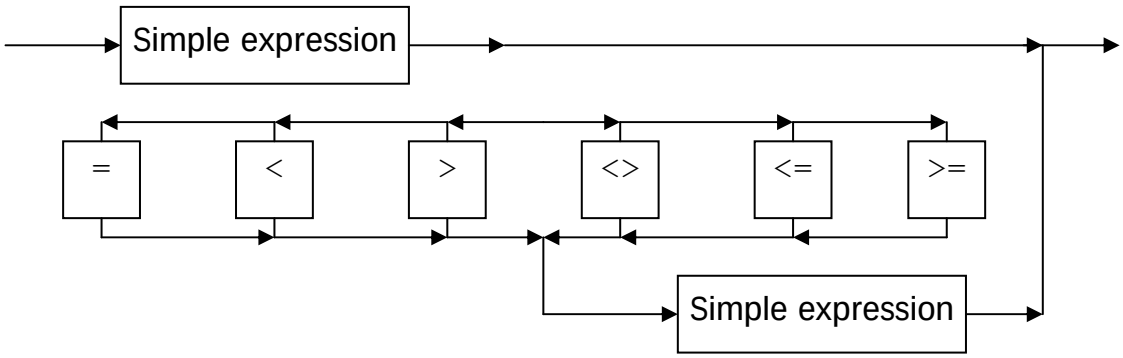
Block



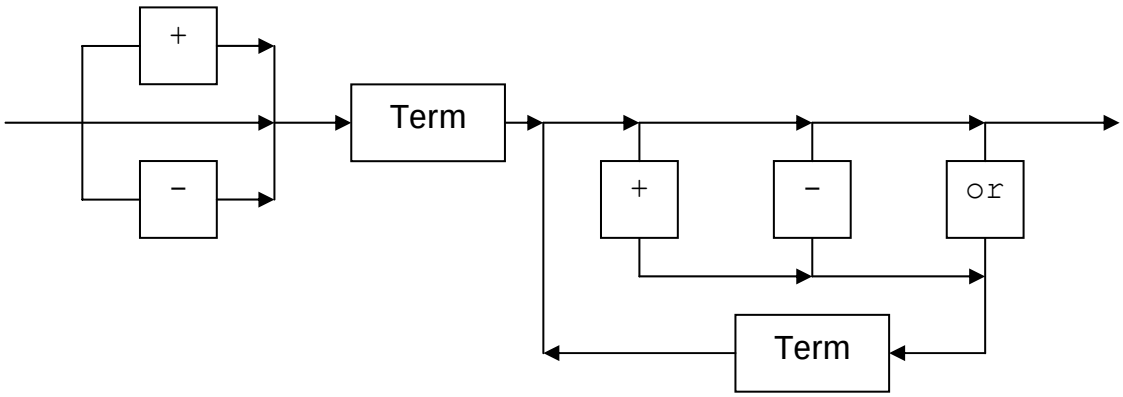
Statement


Variable
:=
Expression
Function identifier
Procedure identifier
(
Expression
)
,
begin
Statement
end
;
if
Expression
then
Statement
else
Statement
while
Expression
do
Statement
repeat
Statement
until
Expression
;
for
Identifier
:=
Expression
downto
to
Expression
do
Statement
case
Expression
of
end
:
Case label list
Statement
;

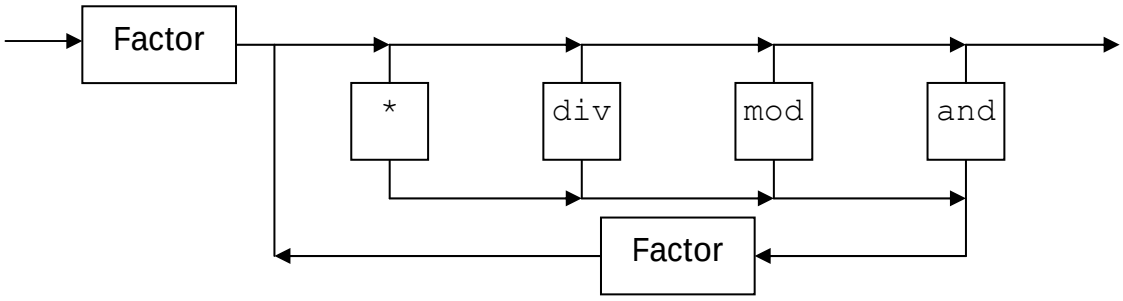
Expression



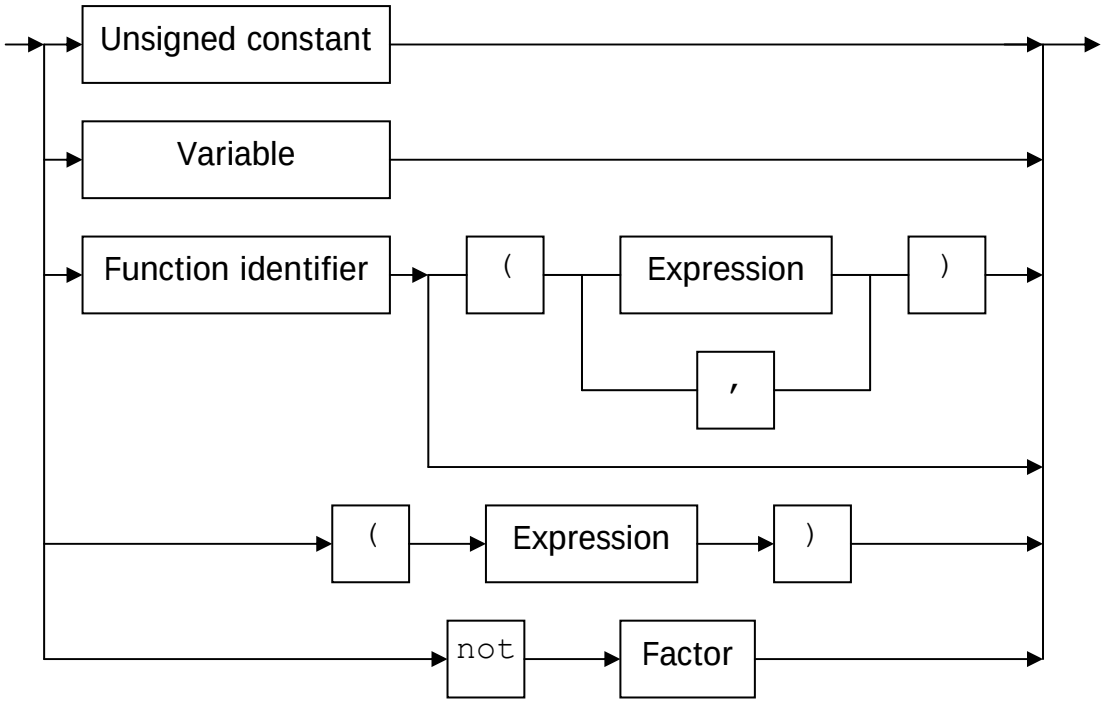
Simple expression



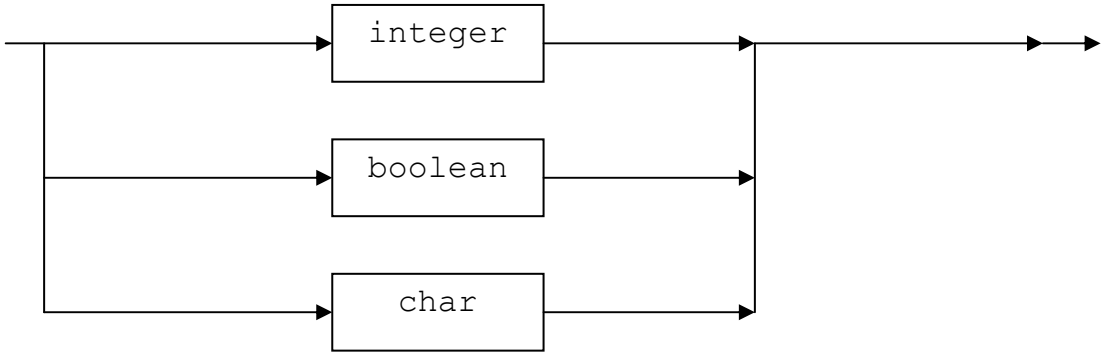
Term



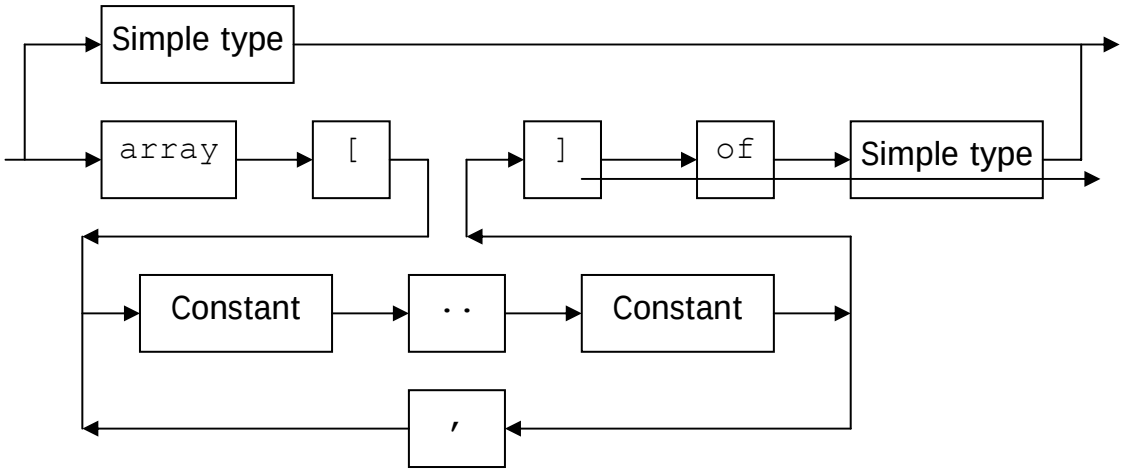
Factor



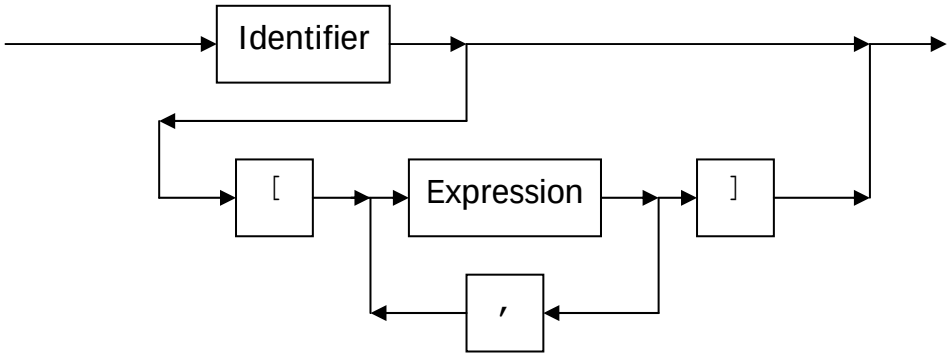
Simple type



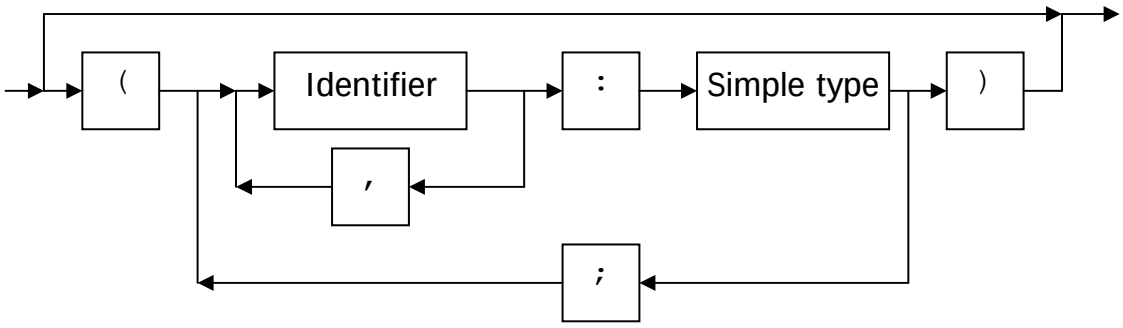
Type



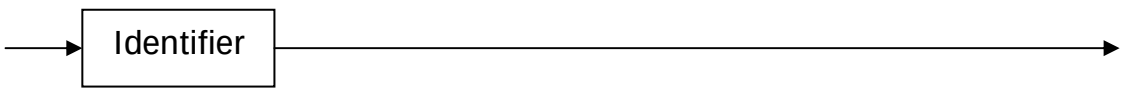
Variable



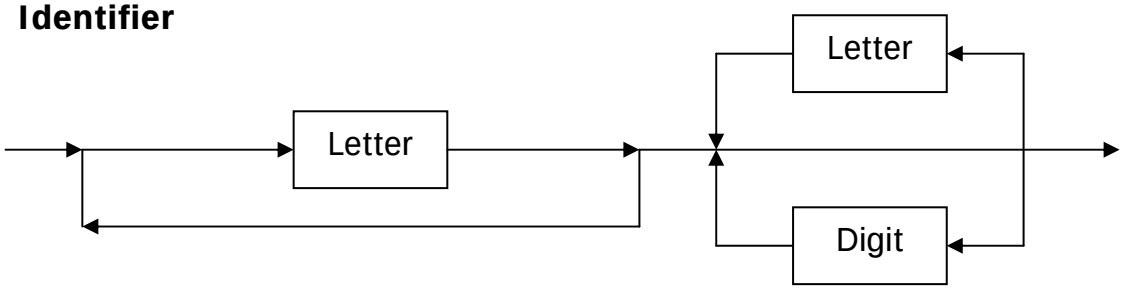
Parameter list



Function identifier

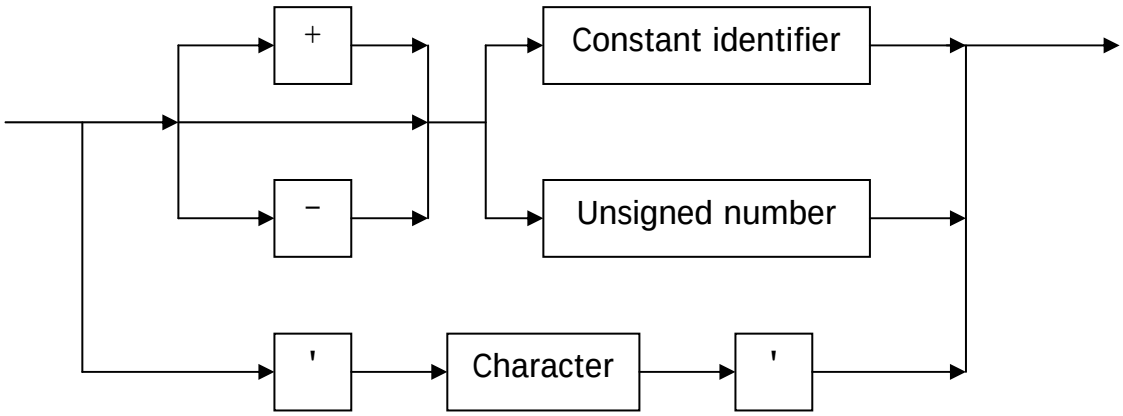


Identifier



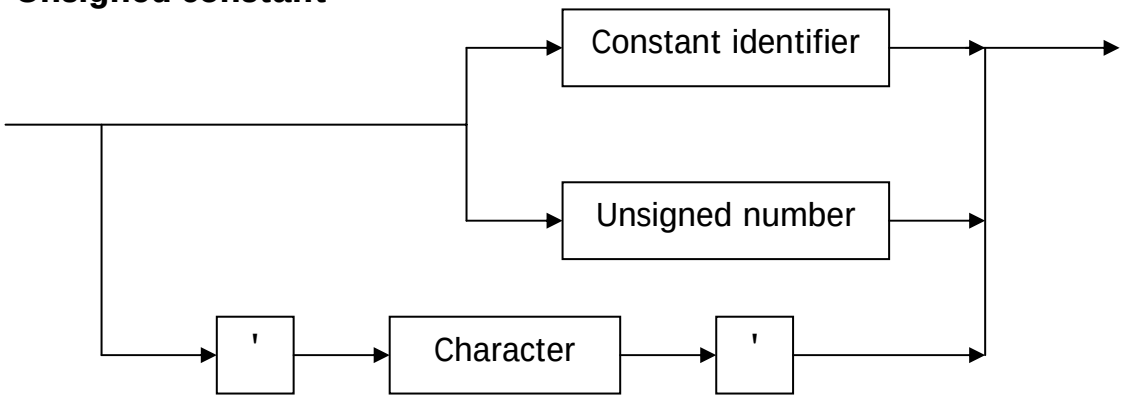
Letter
Any lower case letter.

Constant

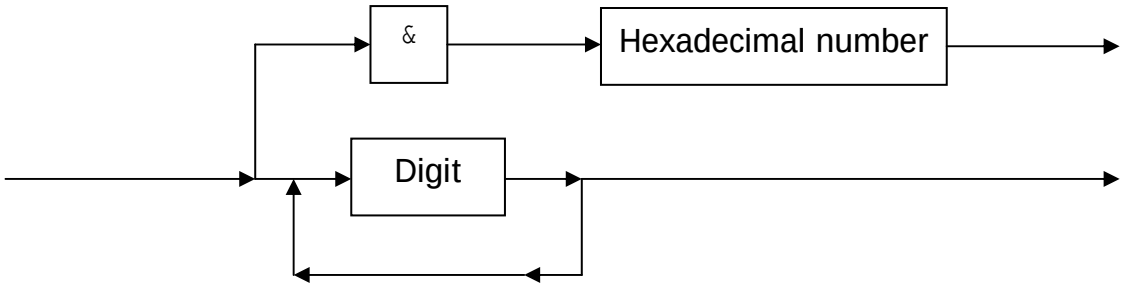


Character
Any printable character.

Unsigned constant

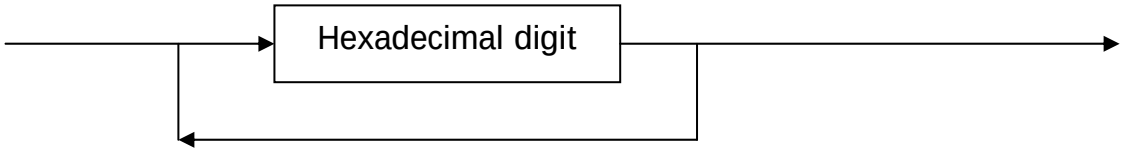


Unsigned number



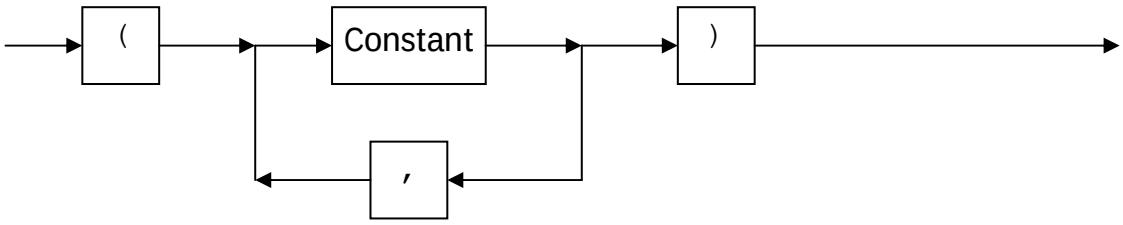
Digit
Any decimal digit 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Hexadecimal number



Hexadecimal digit
Any hexadecimal digit 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Case label list



Index

Review (Electron User)

PASCAL is the latest in a series of programming languages from Acornsoft. It arose from investigations into possible developments resulting from the inclusion of data structuring facilities in an ALGOL-60 like language.

It was designed around 1970 mainly by Professor Niklaus Wirth working at the Institute for Informatics in Zurich, but also benefited by the inclusion of some of the ideas of C. A. R. Hoare who was also working on data structuring facilities in programming languages.

He published his language in 1971 and named it after the great seventeenth century French philosopher Blaise Pascal, who invented one of the earliest known calculators.

Two years later, in 1973, Hoare and Wirth attempted a formal definition of the language in response to user experience to shed light on areas of uncertainty. This led to a revision and extension of the original language.

As with all computer languages, Pascal was designed for a specific purpose. Niklaus Wirth's main objective was a language better suited to teaching programming than any existing language at the time. He was successful in his aims and it soon became popular as a teaching language.

Very quickly, user groups sprang up in several countries to exchange information and ideas on Pascal and the language was adopted by the University of California, San Diego in 1973/4 as their main teaching language. UCSD were responsible for a implementing Pascal for a wide range of computers.

One of the main reasons for Pascal catching on so quickly is that it is concise - the rules of grammar can be written down on just four or five pages.

Pascal is fairly simple to learn although complete beginners may have trouble initially as the knowledge required to write your first program is greater than for Basic.

Pascal is a highly structured language with a rigid format that the programmer is required to adhere to. Everything is laid out so neatly and logically that it is difficult to go wrong. It encourages a style of programming in which programs are built up step by step from small well defined procedures.

All programs start with the word 'program' followed by the name of the program. All the constants and variables used must be declared after the title, plus their type - for example, integer. Any procedures used are defined following the variables and constants and the action part of the program commences with 'begin' and finishes with 'end'.

Pascal programs are very readable, being almost self documenting and needing very few comments. The program flow is easy to follow and the structure clear; making alterations, improvements and debugging very simple.

Lisp is quite interesting, Forth is fast and powerful, Basic just a Mickey Mouse toy for kids - but Pascal is a real programmer's language and a delight to use. Pascal is a complied language, not an interpreted one like Basic which means that Pascal programs run many times faster than their Basic equivalents.

There are two popular ways of implementing Pascal, each with its own

advantages. Either the text of the source code can be decompiled to pure machine code - which makes it very fast but specific to that machine - or it can be compiled to P-Code which is then interpreted when run, not unlike Forth.

This is slower but more easily transferred to other machines. Acornsoft's S-Pascal is not a full blown version but contains a subset of Pascal to teach the language and provide an introduction to structured programming. It is designed for people who know little or nothing about Pascal but are familiar with Basic. It allows short programs of up to 1.25k to be written, compiled and executed.

There are several important differences between this latest language from Acornsoft and the previous ones. The first is noticed immediately on opening the box - which is slightly larger than normal. Inside is the cassette and manual whereas with the other languages, the manual had to be purchased separately on top of the cassette. This makes S-PASCAL some seven pounds cheaper than the others.

The second difference is noticed when S-PASCAL is loaded and totally confused me at first - it wouldn't have if I'd read the instructions, but who does? When loading is complete, after about five minutes, the Electron is still in Basic. The loader can be listed and Basic programs typed in and run. I thought that it hadn't loaded and wondered where the Pascal program was.

S-PASCAL is a compiler only - not an interpreter - so commands cannot be entered in direct mode. What you get are several new * commands to enable you to write, compile and run Pascal programs.

To type in a Pascal program *NEW is entered. Programs can be typed in, edited and listed as with Basic, but using lower case characters so as not to confuse the compiler when it is run with Basic keywords which are stored as tokens.

*COMPILE will activate the compiler producing code which is stored in a reserved area of memory. It can then be executed with *GO.

Pascal programmers will be disappointed with Acornsoft's S-PASCAL as there are so many omissions compared to a full implementation and they will feel very restricted with the subset. However, this is only designed to be a simple, limited version to give people an insight into how Pascal works.

Most Pascal reserved words are present with procedures, functions and arrays being possible, and all the mathematical operators are available. However, hardly any of the predefined functions or procedures have been included such as SIN, COS and ABS.

Variables can be character, Boolean or integer, but not real, which explains why many of the functions are not available.

CALL has been added - not a standard Pascal word - to allow machine code routines and the operating system to be accessed from within Pascal.

Acornsoft have chosen to compile the source text directly to machine code instead of P-Code as with many implementations.

The code is placed at &1100 and there is enough room for about 2.5k. The source text can be saved in the same way as basic and the object code produced, saved with *SAVE.

Compiling the source text directly to machine code has several advantages over compiling to P-Code. After compiling, the compiler -

actually a Basic program 11k long plus 4k workspace, residing at &1F00 - is no longer needed.

This means the object code can be *RUN on its own, or the compiler space used for a Basic program which calls the machine code, or high resolution graphics - for example Mode 0.

Instead of using a Basic compiler program, why not write in Pascal, a far superior language and compile that? A Pascal compiler is far more powerful than a Basic equivalent, with far fewer restrictions. Can a Basic compiler cope with multi-dimensional arrays, procedures and functions to which parameters are passed and that have local variables? Acornsoft's S-PASCAL can.

The compiler uses a two pass assembly, printing the mnemonics and object code each time, and if the printer is enabled, it can be listed. Errors are spotted on the second pass and the appropriate line listed with an arrow pointing to the mistake, and a message is printed saying what the error number is and where it occurred in the line. The error can then be looked up in the manual or on the reference card supplied.

I was curious to find out just how fast Pascal was. How efficient is the machine code? So I wrote equivalent - or near enough - programs in Basic, Forth, Lisp, Pascal and assembly language. It simply involved setting a variable to zero, then going round a loop 30,000 times, incrementing the variable by one each time. The speed test results are shown below:

Assembler	1.4 seconds
Pascal	11.3 seconds
Forth	12.5 seconds
Basic	34.9 seconds
Lisp	285.0 seconds

The test showed Pascal to be up to three times as fast as Basic and marginally faster than Forth, which is generally reckoned to be a fast language itself. The test also highlighted the incredible inefficiency of the code produced - Pascal taking some eight times longer than the specifically written machine code routine.

This is not a criticism of S-PASCAL but is just a fact of life. Compilers cannot hope to be as efficient as a purpose written machine code program.

Acornsoft has achieved their main objective of producing a simple subset of Pascal for teaching the language and structured programming. The compiler is straightforward to use and the manual is short - 67 pages - but clear, and covers every aspect in detail.

The tape, and manual, contain seven demonstration programs showing what the system is capable of, which is quite a lot.

S-PASCAL has a further function as a tool for writing short machine code routines which can be *RUN or called from within a Basic program. This is probably more useful to the experienced programmer.

Programmers are strongly recommended to look at Pascal - especially those writing so called 'spaghetti' programs full of GOTOs. It will improve their structure no end. If you already write structured programs, then learning Pascal will be a doddle.

S-PASCAL is a welcome addition to the list of programming languages for the Electron, and if they ever bring out a full blown version on a ROM Cartridge you can bet that I will be one of the first to get it.

Roland Waddilove, ELECTRON USER 2. 6

acorn electron

S-Pascal

```
 10 program fibonacci (output);
 20
 30  var n,m : integer;
 40
 50  function fib(x:integer):integer;
 60    begin
 70    if x <= 1
 80    then fib := 1
 90    else fib := fib(x-2)+fib(x-1)
100    end;
110
120  begin
130  for m:=0 to 20 do
140    begin
150    n:=fib(m);
160    write('Fib(',m,') is ');
170    writeln(n)
180    end
190
200  end.
```

REFERENCE CARD

Error number 1 – :=expected
Error number 2 – end expected
Error number 3 – to expected
Error number 4 – do expected
Error number 5 – then expected
Error number 6 – wrong no. args
Error number 7 – file not allowed
Error number 8 – file used twice
Error number 9 – Assignment to const
Error number 10 – output file not declared
Error number 11 – ‹punctuation› expected
Error number 12 – until expected
Error number 13 – program expected
Error number 14 – Bad type
Error number 15 – identifier expected
Error number 16 – PROC/ARRAY space
Error number 17 – operator expected
Error number 18 – Type mismatch
Error number 19 – Bracket expected
Error number 20 – TOO MANY VARS
Error number 21 – STACK FULL
Error number 22 – structure error
Error number 23 – wrong no. dims
Error number 24 – TOO MANY LABELS
Error number 25 – name too long
Error number 26 – No such func/proc
Error number 27 – No such var
Error number 28 – . . expected
Error number 29 – No such array
Error number 30 – of expected
Error number 31 – Bad array size
Error number 32 – input file not declared
Error number 33 – reserved word used
Error number 34 – name already used