Chapter 10 Language processors − a LOGO interpreter

When a computer is designed, it is organised so that it will respond to instructions in a very primitive programming language known as machine code. A machine code instruction consists of a bit pattern (Appendix 2) and, when such an instruction is obeyed, it is sent (as a pattern of electrical impulses) to a circuit in the computer which decodes the instruction and activates other circuits to carry out the the fundamental operation represented by the bit pattern. Such instructions may cause the machine to perform an arithmetic operation between two operands, or it may cause the transfer of a unit of information from one part of the machine to the other. A machine code program consists of a sequence of such bit patterns which are sent one after the other from the memory, in which the program resides, to the control unit to be decoded and obeyed.

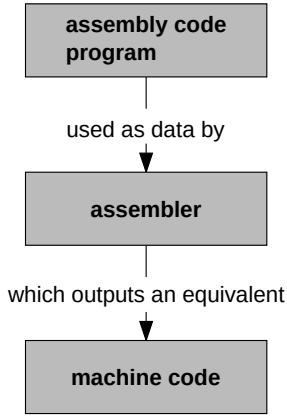
Programming in machine code is extremely tedious and error prone, and before a computer can be easily programmed it has to be provided with software that enables it to be programmed in higher level languages.

For a particular machine a variety of languages maybe available. Some languages are general purpose and some are problem oriented. The current lingua franca of microcomputers is BASIC and this is supposed to be a general purpose high level language.

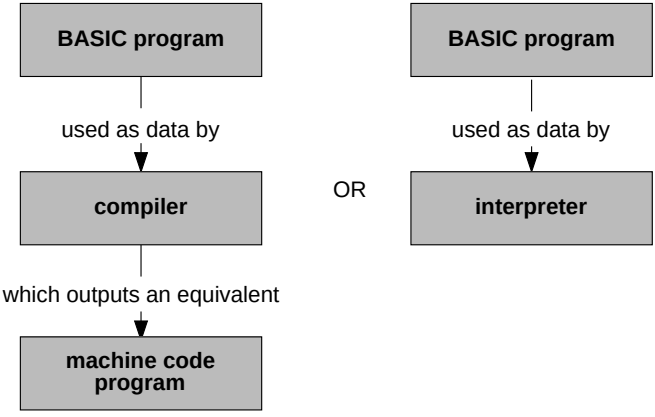
A high level language is designed with two aims in mind. First of all it should be machine independent and this is true, to a greater or lesser extent, of languages such as BASIC, PASCAL, FORTRAN and COBOL. BBC BASIC is a new language incorporating significant extensions to 'standard' BASIC and is thus machine dependent, but it could be said that the designers are attempting to set a new standard. The other aim of a high level language is that it should buffer the user from having to know anything about how the machine works, and, should supply him with the tools that enable him to implement his algorithms with ease. Whether this is achieved with standard BASIC is perhaps a matter for some discussion that we will not go into here.

Programming in low level languages is still sometimes necessary and so that we do not have to write in actual machine code, the lowest level language that is used for practical programming is assembly code. Assembly code is a mnemonic version of machine code that uses characters and

words rather than bit patterns. Progrmming may be necessary in assembly code when access is required ton machine facility that is not available through BASIC, or when high program running speed is essential. For example, many of the arcade games sold for your BBC micro are written in assembly language (although they are translated into machine code before being sold). The program that translates assembly code into machine code is called an assembler.



BASIC is either compiled or interpreted (the difference is explained fully later in the chapter).

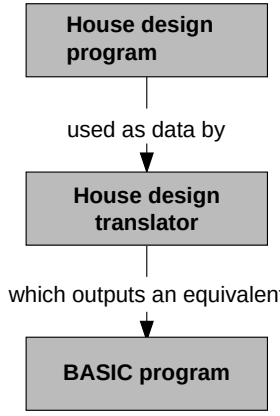


Superimposed on this basic language scheme of a single high level language and an assembly language we may have 'vertical' and 'horizontal' expansion. Vertical expansion means having a higher level language than BASIC or one that is more problem oriented. Say, for example, we wished to have a house design graphics package to be used by

architects. The architect ta only interested in house design and does not want to be concerned with all the tedious details involved in programming graphics tn BASIC. Rather than writing long programs containing BASIC constructs controlling PLOT statements, he wishes to write such things as:

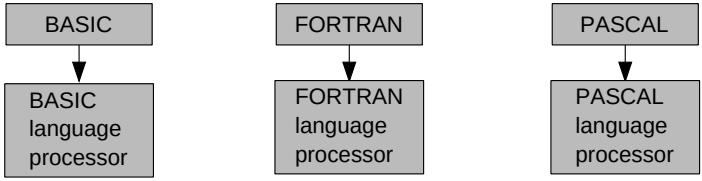
```
drawfrontelevation(wallfinish, doorstyle, windowstyle)
```

He does not need the full generality of BASIC and uses a specific problem oriented language or package. This may well be itself written in BASIC.



The house design translator will output a BASIC program that can then be handled by the BASIC interpreter just like any normal BASIC program. Now although the house design program will be easier to use (for designing houses) than BASIC, its use is restricted to a particular problem environment. The generality and the tedium of BASIC have been traded for an easier to use but more specific language.

The other cominon feature of language systems modern machines is horizontal expansion. Here a number of languages may exist at around the same level. They each have their own interpreter or compiler. A family of general purpose scientifically oriented languages is:



These different languages exist because, although they all offer a common subset of programming constructs, user preference has caused a proliferatton.

Other languages exist 'hortzontally' because they are not general purpose but are meant for particular applications. For example, LISP for list processing and artificial intelligence and COBOL for commercial data processing. Here the programming constructs are designed for the problem environment in which the language is used.

In this chapter we shall be looking at how you can design language processors in BASIC that will accept commands in another language and arrange for the computer to obey these comnands.

10.1 Language processors - an illustrative example

A language processor that enables a computer to be programmed in a new programming language can take one of two forms. A program in the new language may be stored in the computer and 'interpreted' by an 'interpreter' which scans the program being interpreted and carries out the operations indicated by the instructions being scanned. Alternatively, the language processor may take the form of a translation program that translates a program in the new language into an equivalent program, in a language that the computer can handle already. If the language into which programs are translated is machine code the translation program is called a 'compiler', otherwise it is called a translator.

We shall demonstrate the difference between these two approaches by writing both an interpreter and a translator for a rather trivial 'programmning language' that we shall call SKETCH. The symbols in this language are the letters N, 8, E and W and a 'program' in this language consists of a single line containing a sequence of these symbols in any order. This defines the 'syntax' of the language. We must also define the meaning or 'semantics' of programs in our language. We shall say that a program! represents an instruction to draw a line, starting in the centre of the screen, where the letter N indicates that the line should be extended by four units 'North', E indicates that the line should be extended by four units 'East' and so on. Thus, for example, the program:

```
NNNNEEEESSSSWWWW
```

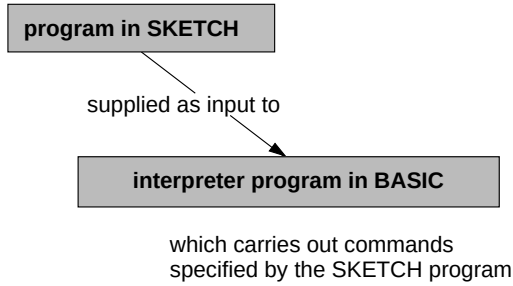
will draw a small square; and:

```
NENENENENENENENE
```

will draw a short diagonal line.

An interpreter for SKETCH

First we shall write, in BASIC, an interpreter for SKETCH programs. The interpreter wilt accept as input a SKETCH program and will carry out the operations specified by the sequence of symbols constituting the SKETCH program.



The main loop in the interpreter program will have the form:

```
REPEAT
  PROCprocesscommand
UNTIL end of program encountered
```

We shall store the prograsn to be interpreted in a string and the interpreter will maintain an integer variable 'next' indicating the point reached so far in the SKETCH program. The complete program is:

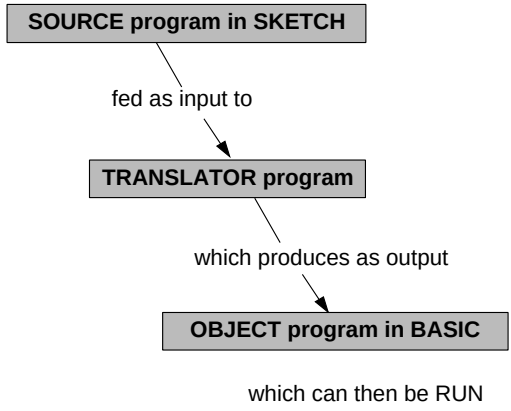
```
 10  INPUT LINE program$
 20  MODE 4
 30  MOVE 640, 512
 40  length=LEN(program$)
 50  next=1
 60  REPEAT
 70    command$=MID$(program$,next,1)
 80    next=next+1
 90    PROCprocesscomand
100  UNTIL next>length
110  END

120  DEF PROCprocesscommand
130  IF conmand$="N" THEN PLOT 1, 0 ,8
140  IF co:mand$="E" THEN PLOT 1, 8, 0
150  IF conmand$="S" THEN PLOT 1, 0,-8
160  IF commandS="W" THEN PLOT 1,-8, 0
170  ENDPROC
```

PROCprocesscommand examines the next symbol in the input 'program' and interprets immediately by drawing a short line in the direction indicated by the symbol. In order to execute a program using an interpreter, it is the interpreter that must be obeyed. The main characteristic of an interpreter is that each instruction in the program being interpreted is analysed and the operation specified by the analysed instruction is carried out immediately.

At the risk of obscuring the ideas present above, it is interesting to note that the BASIC system on the BBC computer is an interpreter. The BASIC programs typed in by the user are stored in random access memory (RAM). A user's BASIC program is not stored completely in character form - the BASIC keywords are stored as integer codes, or 'tokens' as they are sometimes called. This economises on program storage space and also speeds up recognition of the keywords while the program is being interpreted. The interpreter is a largish machine code program permanently stored in 'read only memory' (ROM) and when the user types RUN, it is this interpreter that is obeyed. Thus when the SKETCH interpreter presented above is RUN, this interpreter is itself being interpreted by the BASIC interpreter! If you find this complication confusing, you can simplify matters by ignoring the existence of the BASIC interpreter and imagine your BBC micro as a 'black box' that can obey BASIC programs.

A translator for SKETCH

In this section, we shall write a BASIC program that will translate a SKETCH program into an equivalent BASIC program. Note that there are three programs involved in this process, apart from the BASIC interpreter. The program that carries out the translation process is called the translator. The program that is being translated is usually called the 'source program' and the program being output by the translator is called the 'object program'.



We now have the problem of deciding whom to put the translated version of the SKETCH program as it is generated by the translator. Clearly this program must be stored somewhere if it is eventually to be RUN. For the moment, we shall ignore this problem and shall simply assume that we want the translated program displayed on the screen. It will be quite an easy task to divert this output to a cassette or disc file by using the *SPOOL facility.

Part of the task carried out by a translator is very similar to that carried out by an interpreter. Both programs have to carry out an analysis of the input or source program and recognise the instructions contained in the source program. The differences between an interpreter and a translator manifest themselves only after this syntax analysis has been carried out. Once a particular instruction in the source program! has been recognised, an interpreter will carry out the instruction immediately whereas a branslator outputs one or more object program instructions that are equivalent in meaning to the source program instruction just analysed. For this reason the main loop in our translator is identical to the main loop in our interpreter. Instead of switching to MODE 1 and moving to the centre of the screen, the translator must output instructions to carry out these operations. It must also arrange to number the subsequent lines in the object program that it outputs. Finally, each command in the source program causes an equivalent PLOT statement to be output. Each output PLOT statement is numbered and the variable 'lineno' is used to keep count of these line numbers.

```
 10  INPUT LINE program$
 20  PRINT "10 MODE 4"
 30  PRINT "20 MOVE 640, 512"
 40  lineno = 20
 50  length=LEN(program$) : next=1
 60  REPEAT
 70    command$=MID$(program$,next,1) : next=next+1
 90    PROCprocesscommand
100  UNTIL next>length
110  END

120  DEF PROCprocesscommand
130    IF command$="N" THEN PROCoutline("PLOT 1, 0, 8")
140    IF command$="E" THEN PROCoutline("PLOT 1, 8, 0")
150    IF command$="S" THEN PROCoutline("PLOT 1, 0,-8")
160    IF command$="W" THEN PR0Coutline("PLOT 1,-8, 0")
170  ENDPROC

200  DEFPROCoutline(line$)
210    lineno=lineno+10
220    PRINT ;lineno; " "; line$
230  ENDPROC
```

Here PROCprocesscommand examines the next symbol in the input program and 'translates' it into a PLOT statement for drawing a line in the appropriate direction. If we run the SKETCH translator and input, tor example,

```
NNNNEEEE
```

then the translator will display the object prograntt.

```
 10  MODE 1
 20  MOVE 640, 512
 30  PLOT 1, 4, 0
 40  PLOT 1, 4, 0
 50  PLOT 1, 4, 0
 60  PLOT 1, 4, 0
 70  PLOT 1, 0, 4
 80  PLOT 1, 0, 4
 90  PLOT 1, 0, 4
100  PLOT 1, 0, 4
```

In order to run the program, it must somehow be RUN by the BASIC system. A translator would normally build up the object program in a file or in a separate area of the computer store. A simple way of outputting our object program to a file is to leave our translator exactly as it is and issue a *SPOOL command before running the translator. This diverts a copy of every character that subsequently appears on the screen to a named file, and, the BASIC program in that file can subsequently be loaded using the *EXEC command the complete sequence would be:

```
> *SPOOL "Program"
> RUN
? NNNNEEEE
  .
  .
object program appears on screen
(and in file)
  .
.
> *SPOOL
> NEW
> *EXEC "Program"
> RUN
```

The second *SPOOL command closes the file. This sequence of actions is extremely cumbersome and you may well wonder if there is ever any point in using a translator (or compiler) instead of an interpreter. The answer lies in the fact that once we have gone through the above process, we can run the object program as often as we like. In the case of SKETCH

this is not much of an advantage but tor a more realistie and complex language, a translator or compiler could have considerable advantages.

The process of syntax analysis carried out by a language processor (an interpreter or a compiler) can be very time consuming if the source language is of any complexity and the source program is of any length. For a source program that is to be obeyed many times, it is advantageous to analyse it once and for all and then run the translated version whenever required. If a compiler is available for the source language, then the object program is in machine code and would be handled directly by the computer hardware. When using an interpreter, every time a program is run it is re-analysed.

A second advantage of the use of a translator or compiler concerns the the memory requirement. Using an interpreter, both the source program and the interpreter have to be in memory at the same time. In the case of a translator, it can usually be arranged for the source program to be supplied from a file, a small fragment at a time and during the translation phase only the translator program needs to occupy space. The object program is output to a file as it is generated. During the execution phase of the object program, the translator is no longer required and only the object program needs to occupy space in memory. These considerations are very important when large programs are handled and no doubt you have already encountered space problems in your experience with the BBC micro.

10.2 A simple logo interpreter

Most of the remainder of this chapter will be devoted to a case study - developing a translator for a real language - LOGO. This will be used to illustrate the techniques involved in writing a language processor.

Writing a translator or a compiler requires similar techniques. This is because all types of language processors share the need to analyse the structure of the source program.

An introduction to LOGO

The progratnning language LOGO is widely used for introducing young children to the ideas involved in computer progrannting. LOGO was originally conceived as a language for controlling a 'turtle graphics system'. A turtle is a small wheeled vehicle containing a pen that can be raised or lowered. A LOGO program contains commands that control the actions of the turtle and make it draw a picture. The turtle system is usually replaced by a graphics screen on which the pictures are drawn, but the behaviour of a LOGO program is still explained in terms of an imaginary turtle moving about the screen drawing a picture.

The basic LOGO interpreter usually operates on a line by

line basis. A line of instruction is typed by the user and that line is interpreted immediately by the system, the current picture being extended if appropriate. The turtle always starts in the centre of a blank screen facing the top of the screen.

We begin by introducing a number of simple commands. The following commands each consist of just a single word.

comamnd	effect
PENDOWN	press pen down onto paper (i.e. switch on the plot)
PENUP	lift pen off the paper (i.e. switch off the plot)
CLEARSCREEN	clear the screen and move the turtle to its start position

The following commands each need to be followed by a single numeric parameter.

command	effect
FORWARD	move the turtle forward a specified distance
BACK	move the turtle back a specified distance
LEFT	Turn the turtle anti-clockwise through a specified number of degrees.
RIGHT	Turn the turtle clockwise through a specified number of degrees

Thus the following sequence of LOGO commands will draw a square.

```
PENDOWN
FORWARD 100
LEFT 90
FORWARD 100
LEFT 90
FORWARD 100
LEFT 90
FORWARD 100
```

This LOGO program could be typed one statement to a line, as above, each line being interpreted as it is typed. Alternatively, several statements can be included on a single line, for example:

```
PENDOWN   FORWARD 100
LEFT 90   FORWARD 100
LEFT 90   FORWARD 100
LEFT 90   FORWARD 100
```

or

```
PENDOWN   FORWARD 100  LEFT 90  FORWARD 100
LEFT 90   FORWARD 100  LEFT 90  FORWARD 100
```

Note that there is no statement separator. This small subset of LOGO stateanents will suffice for introductory purposes and we shall write a simple interpreter to handle these statemants. In later sections, we shall introduce LOGO loops and procedures and show how the interpreter can te extended to handle such constructs. For the time being, we also assume that the numeric parameters for our simple statements can only be numeric constants.

Lexical analysis

Most computer programming languages have a more elaborate syntax than the simple language SKETCH that was used in the preceding sections. Before we mve on to the problem of 'syntax analysis', we first look at another problem: that must be considered - that of 'lexical analysis'.

A computer program is usually presented to the computer as a string of characters. The syntax (or grammar), on the other hand, is usually defined in terms of 'symbols' where a symbol may consist of more than one character. A language processor must analyse the sequence of characters with which it is presented and must break it up into the separate-symbols to which the syntax analysis will be applied. The purpose of lexical analysis is to do this, generally ignoring spaces (these are used arbitrarily by a programmer for layout). In the case of the BBC BASIC interpreter, reserved words (FOR, PRINT, etc.) are compressed by lexical analysis into single numeric codes, but we will not do this in the LOGO interpreter. In our LOGO interpreter, lexical analysis will be carried out by PROCgetnextsymbol which will be called whenever the interpreter is ready to examine the next symbol in the LOGO program being interpreted. It will scan through the characters from the point reached so far until a complete symbol has been found and will extract that symbol in the form of a string. A symbol may be a word like PENUP or FORWARD, it may be a number such as 100 or it may be a single character symbol such as '[' or ']' or ':'. (We shall see what square brackets and colons are used for in later sections.) The largest unit handled by our simple interpreter will be a line, each line being interpreted when it has been typed.

The line of LOGO program currently being interpreted will

always be held in a variable 'line$'. A variable 'linep' (short for line pointer) will indicate the position in the line at which the search for the next symbol should start

We shall always ensure that a line is terminated by a recognisable symbol by adding such a symbol to each line before it is processed by the interpreter. The symbol we shall use for this purpose is the single character '!'. Here is the definition of PROCgetnextsymbol.

```
100  DEFPROCgetnextsymbol
110    LOCAL j
120    IF MID$(line$,linep,1)=" " THEN
         REPEAT : linep=linep+1 :
         UNTIL MID$(line$,linep,1)<>" "
130    IF INSTR("[!]:",MID$(line$,linep,1)) THEN
         symbol$=MID$(line$,linep,1) :
         linep=linep+1 : ENDPROC,
140    j=linep
150    REPEAT
160      j=j+1
170    UNTIL INSTR("[!]:",MID$(line$,j,1))
180    symbol$=MID$(line$,linep,j-linep)
190    linep=j
200  ENDPROC
```

Note that the rules concerning separation of symbols are built into this procedure, for example, at line 170.

Syntax analysis

We shall not adopt a very formal or mathematical approach to syntax analysis. There are whole text books on syntax definition and syntax analysis. In Chapter 5, a brief presentation of a notation for the precise definition of the syntax of music rhythms was given. Similar notation can be used for the precise definition of the syntax of a programming language, but we shall not do this here. Instead, we move straight to syntax analysis and adopt the pragmatic (and effective) approach of writing a procedure to handle each construction in the language.

An introductory LOGO interpreter

An interpreter that handles the set of simple LOGO statements that have been described is now presented. A text window of four lines at the botttom of the screen is used for displaying the input lines of the program and any error messages. The remainder of the screen constitutes the graphics area for the turtle.

```
 10  MODE 4
 20  PROCinitialise
 30  REPEAT
 40    PROCprocessline
 50  UNTIL FALSE
 60  END
     .
     .
     .
300  DEF PROCinitialise
310    VDU 24,0;128;1279;1023;
320    VDU 28,0,31,39,28
330    PROCclearscreen
340  ENDPROC

360  DEFPROCclearscreen
370    CLG
380    x=642 : y=578 : MOVE x,y
400    xdir=0 : ydir=1 : angle=90 : penup=TRUE
430  ENDPROC

500  DEF PROCprocessline
510    PROCgetline("Line: ")
520    linep=1 : PROCgetnextsymbol
530    PROCprocessgroup("!")
540  ENDPROC

550  DEF PROCgetline(prompt$)
560    REPEAT : PRINT prompt$; : INPUT LINE ""line$
570    UNTIL line$<>""
580    line$=line$+"!"
590  ENDPROC

600  DEF PROCprocessgroup(terminator$)
610    failed=FALSE
620    REPEAT
630      PROCprocesscommand
640      PROCgetnextsymbol
650    UNTIL symbol$=terminator$ OR failed
560  ENDPROC

700  DEF PROCprocessconmand
710    IF symbol$="PENDOWN" THEN penup=FALSE :ENDPROC
720    IF symbol$="PENUP" THEN penup=TRUE :ENDPROC
730    IF symbol$="CLEARSCREEN" THEN
         PROCclearscreen:ENDPROC
740    IF symbol$="FORWARD" THEN PROCforward:ENDPROC
750    IF symbol$="BACK"    THEN PROCback   :ENDPROC
760    IF symbol$="LEFT"    THEN PROCleft   :ENDPROC
770    IF symbol$="RIGHT"   THEN PROCright  :ENDPROC
780    PROCfail(1)
790  ENDPROC
```

```
  900  DEF PROCforward
  910  LOCAL d
  920    d=FNgetvalue
  930    IF failed THEN ENDPROC
  940    x=x+d*xdir
  950    y=y+d*ydir
  960    IF penup THEN MOVE x,y ELSE DRAW x,y
  970  ENDPROC

 1000  DEF PROCback
 1010  LOCAL d
 1020    d=FNgetvalue
 1030    IF failed THEN ENDPROC
 1040    x=x-d*xdir
 1050    y=y-d*ydir
 1060    IF penup THEN MOVE x,y ELSE DRAW x,y
 1070  ENDPROC

 1100  DEF PROCleft
 1110  LOCAL a
 1120    a=FNgetvalue
 1130    IF failed THEN ENDPROC
 1140    angle=(angle+a) MOD 360
 1150    xdir=COS(RAD(angle))
 1160    ydir=SIN(RAD(angle))
 1170  ENDPROC

 1200  DEF PROCright
 1210  LOCAL a
 1220    a=FNgetvalue
 1230    IF failed THEN ENDPROC
 1240    angle=(angle-a)MOD 360
 1250    xdir=COS(RAD(angle))
 1260    ydir=SIN(RAD(angle))
 1270  ENDPROC

 1300  DEF FNgetvalue
 1310    PROCgetnextsymbol
 1320    value=VAL(symbol$)
 1330    IF value=0 THEN PROCfail(2)
 1340  =value

 1400  DEF PROCfail(errorno)
 1410    PRINT"Error ";errorno
 1420    PRINT LEFT$(line$,LEN(line$)-1)'TAB(linep-2);""
 1430    failed=TRUE
 1440  ENDPROC
```

Although the main unit handled by our interpreter is a line (PROCprocessline), we have defined PROCprocessline in terms of a separate procedure for processing a 'group' of statements, PROCprocessgroup. This procedure takes a

parameter indicating the symbol that is expected to terminate the group. For the time being, a group of statements is just a line of statements, but it will be convenient to have a separate procedure PROCprocessgroup when we come to interpret LOGO REPEAT loops, where it will be necessary to repeatedly process a group of statements up to a loop terminator.

Error handling
It is appropriate at this point to make a few remarks about the problem of 'error handling'. What should the interpreter do it' it encounters a symbol that it does not recognise or does not expect? Clearly an error message should be displayed and our procedure PROCerror displays an error number indicating the type of error encountered.

Error 1 : Unrecognised symbol at the start of a statement

Error 2 : Illegal parameter

The line containing the error is displayed with a marker in the next line pointing to the symbol that caused the problem.

Once an error message has been displayed, our interpreter simply sets a logical variable 'errorfound' to TRUE and this causes interpretation of the current line to be abandoned. It would be rather risky for the interpreter to attempt to continue execution of the line after an error has been detected, as there is no way of knowing which of many possible events caused the error. For example, a symbol may have been misspelt, a symbol may have been omitted, or an extra symbol may have been typed.

A translator or compiler will often attempt to continue its syntax analysis after an error has been encountered with a view to finding all the errors in the program in one go. Whether or not this is useful will depend on how good the translator's 'error recovery' system is. The translator must make some assumption about the likely cause of the error and continue the syntax analysis process on the basis of that assumption. An interpreter cannot easily do this.

10.3 Interpreting loops

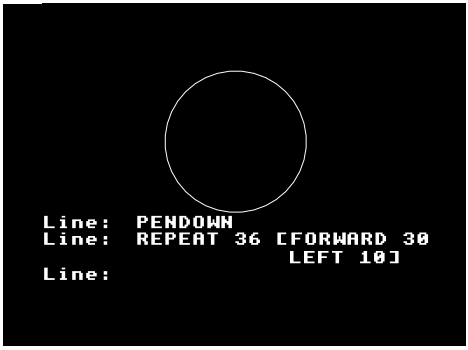
We now extend our subset of LOGO to include simple loops. A LOGO loop takes the form

REPEAT number_of_times [...any nuinber of statements...]

For example, we can abbreviate the instructions for drawing a square to

```
PENDOWN REPEAT 4 [ FORWARD 100 LEFT 90]
```

Here we two more examples:



(These photographs, and the others displayed in this chapter, were obtained using LOGO interpreter that is developed in this chapter.) To deal with such a construction, we must extend PROCprocesscommand to recognise the word REPEAT and take appropriate action.

```
   772   IF sytnbol$="REPEAT" THEN PROCrepeat :ENDPROC
```

To process the loop, the interpreter must evaluate the repeat count, check for the symbol '[', record the point reached in the current line and then repeatedly obey the following group of statements up to the symbol ']' setting 'linep' to point to the start of this group each time round. If an error occurs, execution of the loop should be abandoned.

```
  1500   DEF PROCrepeat
  1510     LOCAL start,no,loop
  1520     no=FNgetvalue
  1530     IF failed THEN ENDPROC
  1540     PROCgetnextsymbol
  1550     IF symbol$<>"[" THEN PROCfail(3)
  1560     start=linep:loop=0
  1570     REPEAT
  1580       loop=loop+1
  1590       linep=start
  1600       PROCgetnextsymbol
  1610       PROCprocessgroup("]")
  1620     UNTIL loop=no OR failed
  1630   ENDPROC
```

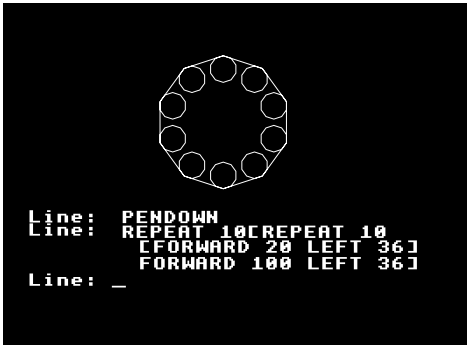
Note that there is hidden recursion here: PROCprocessline calls PROCprocessgroup which may call PROCrepeat which calls

PROCprocessgroup.

The above alterations will also handle loops within loops: such as

```
REPEAT 4 [ REPEAT 4 [DRAW 100 LEFT 90] RIGHT 90]
```

which draws a pattern of four squares (a 'window pane' pattern). Here are two more examples:



Such constructs will cause further recursion in the interpreter. PROCprocessgroup calls PROCrepeat which calls PROCprocessgroup which calls PROCrepeat which calls PROCrocessgroup. Note that the use of LOCAL in these procedures is essential (see Chapter 7). The activation of PROCrepeat for the inner loop must have its own local variable for recording information about the loop, so as not to destroy the corresponding information for the outer loop.

10.4 Defining and interpreting simple LOGO procedures

A LOGO procedure definition is started by a line containing the word TO followed by the name of the procedure. For example

```
TO DRAWSQUARE
  PENDOWN
  REPEAT 4 [FORWARD 100 LEFT 90]
END
```

This definition will be stored by the LOGO system and the LOGO prograrmner can subsequently type lines such as

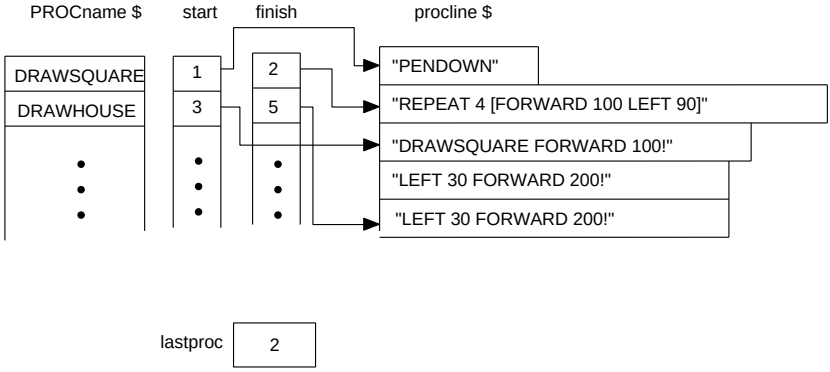
```
DRAWSQUARE
DRAWSQUARE   RIGHT 90   DRAWSQUARE
REPEAT 4 [DRAWSQUARE   RIGHT 90]
```

which draws the 'window pane' pattern. Here is another call

of DRAWSQUARE and the pattern produced:

In order to implement such a facility, we require to use some form of look-up table in which a procedure name can be stored together with its definition. Whenever the system is processing a command that starts with a symbol that is not one of the LOGO primitive symbols, then that symbol must be looked up in the table. If it is found there, the procedure definition associated with the symbol must be returned and obeyed. If it is not found then the system must report an unrecognised symbol as before.

For the purposes of illustration, we shall use simple linear search for finding procedure names in our table. Associated with each procedure name will be two 'index entries' that point to the start and finish of the procedure definition in an array of strings that contains the lines of all the procedures stored. For example, the next diagram shows the state of the table when it contains two simple procedure definitions.



Because DRAWHOUSE appears in slot 2 of 'procname$, this means that 'start(2)' and 'finish(2)' indicate the position in 'procline$' of the first and last lines of the definition of DRAWHOUSE. These arrays are declared and initialised by:

```
304  DIM procname$(10),start(10),finish(10),
         procline$(100)
         .
         .
         .
306  lastproc=0: lastline=0
```

We now need to arrange for a procedure definition to be added to the above table when a procedure heading is encountered. We insist that the TO heading for the definition of a procedure appears by itself on a line and it is therefore appropriate to alter PROCprocessline so that it can recognise the start of a LOGO procedure definition and act accordingly:

```
530  IF symbol$="TO" THEN PROCdefineproc
       ELSE PROCprocessgroup("!")
```

PROCdefineproc will read the rest of the procedure definition a line at a time and add it to the above data structure :

```
1700  DEF PROCdefineproc
1710    PROCgetnextsymbol
1720    lastproc=lastproc+1
1730    procname$(lastproc)=symbol$
1740    start(lastproc)=lastline+1
1750    PROCget1ine("TO line: ")
1760    REPEAT
1770      lastline=1astline+1
1780      procline$(lastline) = line$
1790      PROCgetline("TO line: ")
1800    UNTIL line$="END!"
1810    finish(lastproc)=lastline
1820  ENDPROC
```

PROCprocesscommand now needs to be extended to cover the possibility that the symbol at the start of a command is in the procedure table:

```
705  LOCAL procfound, proc
       .
       .
       .
774    PROClookup
776    IF procfound THEN PROCcall(proc):ENDPROC
```

Finally, PROClookup and PROCcallproc are defined:

```
1900  DEF PROClookup
1910    IF lastproc=0 THEN procfound=FALSE : ENDPROC
1920    proc=0
1930    REPEAT
1940      proc=proc+1
1950      procfound = procname$(proc)=symbol$
1960    UNTIL procfound OR proc=lastproc
1970  ENDPROC

2000  DEF PROCcall(proc)
2010  LOCAL line$,linep,count
2020    count=start(proc)
2030    REPEAT
2040      line$=procline$(count)
2050      linep=1 : PROCgetnextsymbol
2060      PROCprocessgroup("!")
2070      count = count+1
2080    UNTIL count>finish(proc) OR failed
2090  ENDPROC
```

The declaration of LOCAL variables 'line$' and 'nextp' in PROCcall is absolutely essential. The new variables with these names are used for holding successive lines of the LOGO procedure as they are being interpreted. The previous variables with these names are restored when PROCcall terminates and interpretation can then continue on the line that contained the LOGO procedure call. You will also find that, for similar reasons, the LOCAL declaration at line 705 is essential if one LOGO procedure is to be allowed to call another.

10.5 Parameters and variables

In this section we shall demonstrate how our interpreter can be extended to handle procedure parameters. A parameter is simply a local variable that is given a value when a procedure is called and similar techniques to those described here could be used to extend our LOGO interpreter to handle variables of all kinds.

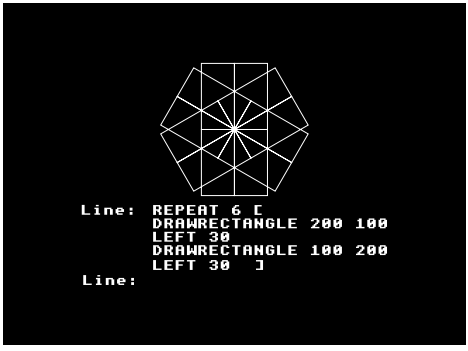
Here is an example of a LOGO procedure to draw a rectangle whose length and width are specified as parameters. The turtle ends up in the same position and pointing the same way as it was when it started.

```
TO DRAWRECTANGLE  :LENGTH  :WIDTH
  PENDOWN
  FORWARD :LENGTH  LEFT 90
  FORWARD :WIDTH   LEFT 90
  FORWARD :LENGTH  LEFT 90
  FORWARD :WIDTH   LEFT 90
END
```

Note that a LOGO variable or parameter name is always preceded by a colon. This procedure could be called by, for example,

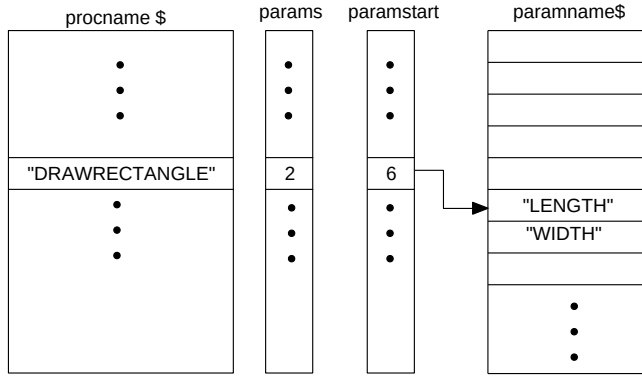
```
DRAWRECTANGLE 200  100
RIGHT 90   DRAWRECTANGLE 100 75
RIGHT 90   PENUP   FORWARD 100
DRAWRECTANGLE 200 50
```

or :



There are two aspects to the problem of handling parameters. Firstly, additional information has to be stored in the procedure table described earlier, so that when a procedure is called, the interpreter knows how many parameters to expect after the procedure name, each time it is used. Secondly, when a procedure is called, the value supplied for each parameter must be associated with the name of the pararneter so that when the name is encountered in a statement being obeyed, its current value can obtained. To handle the first problem, we extend our procedure table to include the number of parameters each procedure takes together with an index pointer to an array that contains the parameter names. The arrays 'start', 'finish' and 'procline$' are set up as before. We need the additional statements:

```
305  DIM pnrams(10), paramstart(10), paramname$(30)
307  lastpn = 0
```



When a procedure definition is processed, the heading must be analysed in more detail, in case it includes parameters. The alterations to PROCdefineproc to add parameter information to the procedure table are as follows.

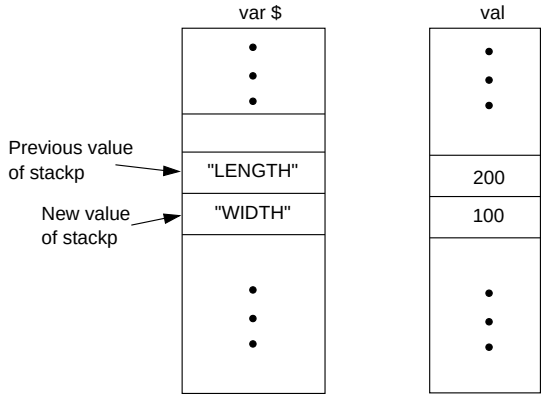
```
1735  PROCgetparamnames
        .
        .
2200  DEF PROCgetparamnames
2210    params(lastproc)=0
2220    PROCgetnextsymbol
2230    IF symbol$<>":" THEN ENDPROC
2240    paramstart(lastproc)=lastpn+1
2250    REPEAT
2260      params(lastproc)=params(lastproc)+1
2270      PROCgetnextsymbol
2280      lastpn=lastpn+1
2290      paramname$(lastpn)=symbol$
2300      PROCgetnextsymbol
2310    UNTIL symbol$<>":"
2320  ENDPROC
```

When a procedure call is processed and the procedure name is found in the table, the extra information is used to determine how many parameter values to search for. These values must then be paired with the corresponding parameter names. For this purpose, we will use a parameter or variable value table, or 'stack' as it is more usually called. This stack will consist of two parallel arrays containing

parameter (or variable) names and corresponding values. A variable 'stackp' points to the last entry in the table. When a procedure is called, for example, by:

```
   DRAWRECTANGLE 200 100
```

two new pairs of values will be added to the stack as follows:



The stack is initialised by:

```
  303   DIM var$(100), val(100)
          :
  308   stackp = 0
```

PROCcallproc must in extended to add this infornnation to the variable stack.

```
 2005   PROCgetparamvals:IF failed THEN ENDPROC
        :
 2400   DEF PROCgetparamvals
 2410   LOCAL pa,nextpn,v
 2420   IF params(proc)=0 THEN ENDPROC
 2430   pn=paramstart(proc)
 2440   nextpn=pn+params(proc)
 2450   REPEAT
 2460   v=FNgetvalue
 2470   stackp=stackp+1
 2480   var$(stackp)=paramname$(pn) : val(stackp)=v
 2490   pn=pn+1
 2500   UNTIL pn=nextpn OR failed
 2510   ENDPROC
```

Note the position of line 2005 within PROCcall. The LOGO parameter values must bv obtained from the line of LOGO that contains the procedure call being obeyed. Only when this has been done can the LOCAL versions of 'line$' and 'nextp' be declared. When a procedure call terminates, the parameter information can be removed from the stack as follows:

```
2085  stackp=stackp-params(proc)
        :REM clears parameters from stack
```

When the procedure is being obeyed, the above table can now be used to look up a named value encountered while a line of the procedure is being interpreted. When a name is being searched for in the variable stack, we must search through the stack in reverse order. We want to find the most recent occurrence of the variable whose name has been encountered. There may be two currently active procedure calls, each with a parameter of the same name and, when that name is encountered, it is the value supplied with the most recent procedure activation that must be used. FNgetvalue needs to be extended.

```
1315  IF symbol$=":" THEN =FNvarvalue
        .
        .
        .
2600  DEF FNvarvalue
2610  LOCAL varfound,sp
2620    PROCgetnextsymbol
2630    IF stackp=0 THEN PROCfail(5): =0
2640    sp=stackp+1 : varfound=FALSE
2650    REPEAT
2660      sp=sp-1
2670      varfound = var$(sp)=symbol$
2680    UNTIL varfound OR sp=1
2690    IF varfoundTHEN =val(sp)
2700    PROCfail(5)
2710  =0
```

Exercises

1 Arrange for our LOGO interpreter to output meaningful error messages instead of error numbers.

2 In our LOGO interpreter, the only expressions that can be used for representing parameter values are single integer constants or single variable names. Extend the interpreter so that it will accept expressions involving + and -. The easiest way to do this is to insist that an expression is always enclosed in round brackets. If this restriction is not imposed, you will have to restructure

the way in which the interpreter calls PROCgetnextsymbol.

3 A LOGO IF statement has the form

IF condition THEN [...list of statements...]

where a 'condition' is an expression involving =, <, >, etc. Extend the interpreter to handle IF statements. You may again prefer to insist that expressions (conditions) are always enclosed in round brackets and you could restrict the operators permitted in a condition.

4 The LOGO STOP statement has the same effect as an ENDPROC statement in BASIC. Implement this. You should now find that your interpreter can handle recursive procedures such as:

```
TO SPIRALRECTANGLES :LENGTH :WIDTH
    IF (:WIDTH<4) THEN [STOP]
    DRAWRECTANGLE :LENGTH :WIDTH
    LEFT 10
    SPIRALRECTANGLES (:LENGTH-4 ) (:WIDTH-4)
END
```

10.6 A program compacter

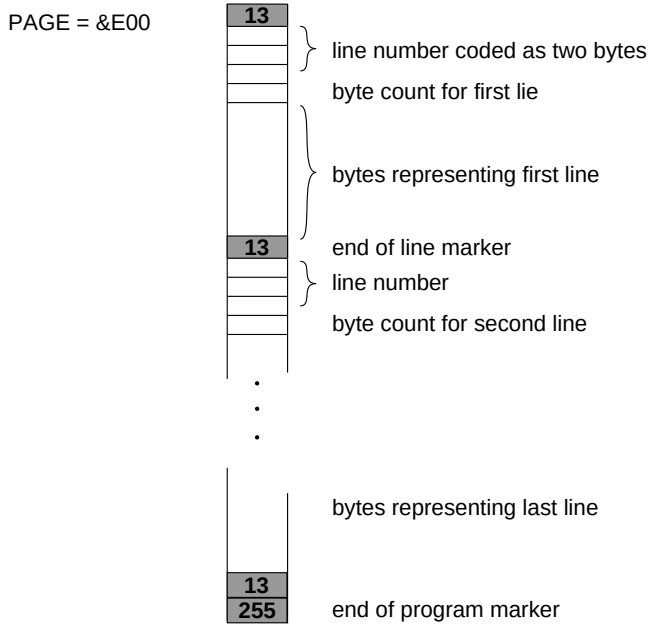
In this final section on language processors, we present an extremely useful utility program that can be used to process the BASIC programs that you write. The program compacts BASIC programs and this is useful for two reasons. Firstly throughout this book we have been fairly extravagant with our use of long variable names, extra spaces, remarks and the insertion of redundant progranl features such. as the control variable after NEXT. All of these deivices contribute towards the readability of a program and this is extremely important when developing a program of any complexity or indeed when writing a book on programming techniques. However, because BASIC is an interpreted language, all these features occupy extra storage space. This may mean that there is not enough user memory space (or RAM) available for the program, its variables and arrays and the screen memory needed to run the program in a high resolution graphics mode such as MODE 0.

We can, however, adopt the strategy of developing the program in a low resolution mode such as MODE 4 and eventually run the program in a high resolution mode by compacting it. Once the program development is complete and the program is ready for regular use, the readable version has served its purpose and we can remove all the redundant spaces shorten our variable names and so on. Of course this strategy is not guaranteed to work - a program may still be

too long after compaction.

Depending on the style of programming that you adopt a compacted program will occupy some fraction of the storage space occupied by the uncompacted version. Programs in this book can be compacted by around 50 per cent so you can see that the savings are worthwhile. For example the f(x,y) hidden line removal algorithm in Chapter 3 (Section 3.5) must be compacted before it can be run in MODE 0. Another incidental advantage in using compacted programs is that the loading tine is also reduced in the same proportion. A disadvantage with compacted programs is that they are by definition not amenable to further development and you should always keep a copy of the uncompacted version in case further development is necessary.

The second reason for presenting a program compacter in this chapter is that it gives further insight into the way in which BASIC programs are represented in store. The next diagram illustrates the line by line organisation of a BASIC program which is normally stored by the computer in memory locations &E00 onwards (in a cassette based system).



Within a line, each BASIC keyword is represented as a single one-byte code or 'token'. For example REM is coded as 244 or &F4. NEXT is coded as &ED. Apart from keywords, (and a special coded representation for labels) each character in a user's program is stored as the corresponding ASCII code.

The above inforntation should enable you to understand our explanation (below) of how the compacter program works. The program to be compacted should be loaded with PAGE set to its normal value (PAGE = &E00 for a cassette system). The compacter itself must be loaded with

```
PAGE = &5000
```

and run with PAGE set to this value. When the compacter has been run, change PAGE back to &E00 to use the compacted program.

```
 10  putp=&E00:lookp=&E00
 20  DIM var$(100)
 30  lastvar=-1
 40  PROCcopycode
 50  REPEAT
 60    PROCline
 70  UNTIL ?lookp=255
 80  PROCcopycode
 90  END

100  DEF PROCline
110  LOCAL chars,countp
120    chars=0
130    PROCcopycode
140    PROCcopycode
150    countp=putp
160    PROCcopycode
170    PROCskipspaces
180    IF ?lookp=42 OR ?lookp=&DC THEN
         PROCcopyline
       ELSE IF ?lookp<>13 THEN
           REPEAT:PROCcheckcode:UNTIL ?lookp=13
190    PROCcopycode
200    ?countp=chars
210    IF chars=4 THEN putp=putp-4
220  ENDPROC

230  DEF PROCcopycode
240    ?putp=?lookp:putp=putp+1
250    lookp=lookp+1:chars=chars+1
260  ENDPROC

270  DEF PROCcheckcode
280    IF ?lookp=32 THEN lookp=lookp+1:ENDPROC
290    IF ?lookp=34 THEN PROCcopystring:ENDPROC
300    IF ?lookp=&F4 THEN PROCrem:ENDPROC
310    IF (?lookp>64 AND ?lookp<91) OR
         (?lookp>96 AND ?lookp<123) OR ?lookp=95
       THEN PROCvariable:ENDPROC
```

```
320     IF ?lookp>47 AND ?lookp<58 THEN
          PROCnumber:ENDPROC
330     IF ?lookp=38 THEN PROChex : ENDPROC
340     IF ?lookp=&ED THEN PROCnext :ENDPROC
350     IF ?lookp=141 THEN
          PROCcopycode:PROCcopycode :PROCcopycode:
          ENDPROC
          :REM 141 is code that precedes a label.
360     IF ?lookp=42 AND (?(putp-1)=58 OR
          ?(putp-1)=&BE OR ?(putp-1)=&8C)
        THEN PROCcopyline:ENDPROC
370     PROCcopycode
380   ENDPROC

390   DEF PROCcopystring
400     REPEAT
410       PROCcopycode
420     UNTIL ?lookp=34
430     PROCcopycode
440   ENDPROC

450   DEF PROCrem
460     REPEAT
470       lookp=lookp+1
480     UNTIL ?lookp=13
490   ENDPROC

500   DEF PROCvariable
510   LOCAL name$,position
520     name$=""
530     REPEAT
540       IF FNvarchar THEN
            name$=name$+CHR$(?lookp):lookp=lookp+1
550     UNTIL NOT FNvarchar
560     IF LEN(name$)=1 AND
          ASC(name$)>64 AND ASC(name$)<91
        THEN PROCputvar(name$):ENDPROC
570     position=FNlookup
580     newname$=FNmakename(position)
590     PROCputvar(newname$)
600     PRINT name$;TAB(20);newname$
610   ENDPROC

620   DEF PROCputvar(n$)
630   LOCAL i
640     FOR i=1 TO LEN(n$)
650       ?putp=ASC(MID$(n$,i,1))
660       putp=putp+1:chars=chars+1
670     NEXT i
680     PROCskipspaces
690     IF FNvarchar THEN
          ?putp=32:putp=putp+1:chars=chars+1
700   ENDPROC
```

```
  710  DEF FNlookup
  720  LOCAL i
  730    i=-1
  740    REPEAT
  750      i=i+1
  760      found=(var$(i)=name$)
  770    UNTIL found OR i>lastvar
  780    IF found THEN =i
  790    lastvar=lastvar+1
  800    var$(lastvar)=name$
  810  =lastvar

  820  DEF FNmakename(no)
  830    letter$=CHR$(no MOD 26 + 97)
  840    IF no<26 THEN =letter$
  850  =letter$+CHR$(no DIV 26 + 96)

  860  DEF PROChex
  870  LOCAL hexchar,ch
  880    PROCcopycode
  890    REPEAT
  900      ch=?lookp
  910      hexchar=(ch>47 AND ch<58) OR (ch>64 AND ch<71)
  920      IF hexchar THEN PROCcopycode
  930    UNTIL NOT hexchar
  940  ENDPROC

  950  DEF PROCnext
  960    PROCcopycode
  970    PROCskipspaces
  980    REPEAT
  990      IF ?lookp=44 THEN
             ?putp=58:putp=putp+1:?putp=237:putp=putp+1:
             lookp=lookp+1:chars=chars+2 :PROCskipspaces
 1000      REPEAT
 1010        IF FNvarchar THEN lookp=lookp+1
 1020      UNTIL NOT FNvarchar
 1030      PROCskipspaces
 1040    UNTIL ?lookp<>44
 1050  ENDPROC

 1060  DEF FNvarchar
 1070  LOCAL ch
 1080  ch=?lookp
 1090  =(ch>64 AND ch<91) OR (ch>96 AND ch<123) OR
         (ch>47 AND ch<58) OR ch=95

 1100  DEF PROCskipspaces
 1110    REPEAT
 1120      IF ?lookp=32 THEN lookp=lookp+1
 1130    UNTIL ?lookp<>32
 1140  ENDPROC
```

```
1150  DEF PROCcopyline
1160    REPEAT
1170      PROCcopycode
1180    UNTIL ?lookp=13
1190  ENDPROC

1200  DEF PROCnumber
1210    PROCcopynumber
1220    PROCskipspaces
1230    IF FNvarchar THEN
          ?putp=32:putp=putp+1 :chars=chars+1
1240  ENDPROC

1250  DEF PROCcopynumber
1260    REPEAT
1270      PROCcopycode
1280    UNTIL ?lookp<48 OR ?lookp>57
1290  ENDPROC
```

The compacter operates with two addresses 'lookp' and 'putp'. 'lookp'contains the address of the next byte in the uncompacted program and 'putp' contains the address for the next byte in the compacted program. These pointers both start at &E00, but they will soon get out of step as spaces, REMs and blank lines are eliminated, and variable names are shortened.

Note the special cases that have to be checked for by PROCcheckcode. We will deal with the handling of variables (line 310) in a moment. A space in the source program is simply ignored. There are a number of other special cases that also have to be dealt with.

Characters between quotation marks must be copied exactly as they stand (including spaces) or the behaviour of the compacted program could change. If the character code for a quotation mark (34) is encountered, then PROCcopystring is called. This copies characters from the source program to the object program until the next question mark is encountered.

If the word REM (token &F4) is encountered, the rest of the line in the source program is skipped by PROCrem. Decimal numbers and hex numbers are copied character for character (PROCnumber and PROChex). Any variable after the word NEXT (token &ED) is omitted (PROCnext).

A label is stored in a program in a special coded form (code 141 followed by two bytes) which has to be copied (line 350).

Operating system commands (statements starting with a *) or DATA statements (token &DC) cause the rest of the current line in the source program to be copied (lines 180 and 360).

Variable names are replaced with shortened names as they are encountered. This is done by PROCvariable (called at line 310). Clearly, if the same name is encountered at

several points in the source program, it must always be replaced by the same shortened name in the object program. This is achieved by building up a table of the variable names encountered in the source program. A name is added to the table only if it is not already there (see FNlookup). The position of a name in this table determines the shortened name to be used (see FNmakename). The first 26 entries in the table are given the single letter names 'a', 'b', 'c', ..., 'z'. The remainder are given the nanies 'aa', 'ba', 'ca', ..., 'za', 'ab', 'bb', 'cb', ... Single capital letter variable names are left unaltered - some of these, such as X%, Y%, A%, P% and so on, have special significance in some programs.

Here is an example of a compacted program (the hidden line removal algorithm of Chapter 3, Section 3.5).

```
   10INPUTa,b,c,d
   20PROCe(a,b,c)
   30MODE4:VDU29,640;512;
   32PROCf
   40FORg=360TO-360STEP-20
   50PROCh(g,-360,FNi(g,-360)):MOVEj,k
   55l=(640+j)DIVm:n=k
   60FORo=-340TO360STEP20
   70PROCh(g,o,FNi(g,o)):PROCp
   80NEXT
   90NEXT
  100q=GET:MODE7:END
  110DEFFNi(r,s)=100*(COS(RAD(r))+COS(RAD(s)))
  120DEFPROCh(r,s,t)
  130LOCALu,v,w
  140PROCx(r,s,t)
  150PROCy(u,v,w,d)
  160ENDPROC
  300DEFPROCf
  310LOCALz
  320m=4:aa=1279DIVm
  330DIMba(aa),ca(aa)
  340FORz=0TOaa
  350ba(z)=512:ca(z)=-512
  360NEXT
  370ENDPROC
  400DEFPROCp
  410LOCALda,ea,fa,z
  420da=(j+640)DIVm
  430IFda=lTHENPROCga(da,k)
  440ha=(k-n)/(da-l)
  450fa=n
  460FORz=lTOda
  470fa=fa+ha
  480PROCga(z,fa)
  490NEXT
```

```
  500l=da:n=k
  510ENDPROC
  520DEFPROCga(z,s)
  530LOCALr:r=z*m-640
  540IFz<0ORz>aaTHENMOVEr,s:ENDPROC
  550IFs<ca(z)ANDs>ba(z)THENMOVEr,s:ENDPROC
  560IFs<ba(z)THENba(z)=s
  570IFs>ca(z)THENca(z)=s
  580DRAWr,s
  590ENDPROC
 1000DEFPROCe(a,b,c)
 1010LOCALia,ja,ka,la
 1020ia=SIN(RAD(b)):ja=COS(RAD(b))
 1030ka=SIN(RAD(c)):la=COS(RAD(c))
 1040ma=-ia:na=ja
 1050oa=-ja*la:pa=-ia*la
 1060qa=ka
 1070ra=-ja*ka:sa=-ia*ka
 1080ta=-la:ua=a
 1090ENDPROC
 1100DEFPROCx(r,s,t)
 1110u=ma*r+na*s
 1120v=oa*r+pa*s+qa*t
 1130w=ra*r+sa*s+ta*t+ua
 1140ENDPROC
 1150DEFPROCy(u,v,w,d)
 1160j=d*u/w
 1170k=d*v/w
 1180ENDPROC
```

It is interesting to note that the compacted program could not be typed in the form in which it appears here. Extra spaces would have to be inserted between variable names and keywords so that the system could distinguish them. However, a keyword is stored as a single recognisable token and these spaces can be omitted in the internal version of a program.

Exercises

1 Try applying the compacter to itself! First load the compacter with PAGE=&E00. Then set PAGE=&5000, load the compacter again and run it. Set PAGE=&E00 and you should find a compacted compacter.

2 The compacter runs rather slowly. This is partly due to the complexity of the task that it has to carry out, but its operation on a large source program could be speeded up by using a more efficient table look-upt method. Do this.

3 A further saving in program size could be made by joining consecutive lines of program into a single line (with extra colons inserted). Each time two consecutive lines are joined, three further bytes of memory space are saved. Note the following points:

(a) Any line that includes IF statements must not be joined onto the next line.

(b) Any line starting with DEF must not be joined onto the previous line.

(c) Any line referred to by a label elsewhere in the program must not be joined onto the previous line. Your program will need to build up a list of labels by doing a preliminary scan through the source program.

4 Our LOGO interpreter was not written with efficiency of storage use in mind. Devise a scheme for storing LOGO programs that is similar to that used for storing BASIC programs. For example, invent single one-byte codes for the LOGO keywords and store these in place of the keywords.