

```


230 DEF PROGRAM M%=X%-V%: N%=ABS(X%-U%):
    IF N%=0 ENDPROC
240 S%=(X%-U%)/N%
    K%=V%+I%*M%/N%
250 FOR I%=1 TO N%: J%=U%+I%*S%:
    K%=V%+I%*M%/N%
260 IF J%<0 OR J%>319 THEN 320
270 IF K%>Q%2J% Q%2J%=K%
280 IF K%<R%2J% R%2J%=K%
290 IF A%=0 THEN 320
300 MOVE4*(J%-S%),4*(Q%2(J%-S%)):-
    DRAW4*J%,4*(Q%2J%)
310 MOVE4*(J%-S%),4*(R%2(J%-S%)):-
    DRAW4*J%,4*(R%2J%)
320 NEXT: ENDPROC

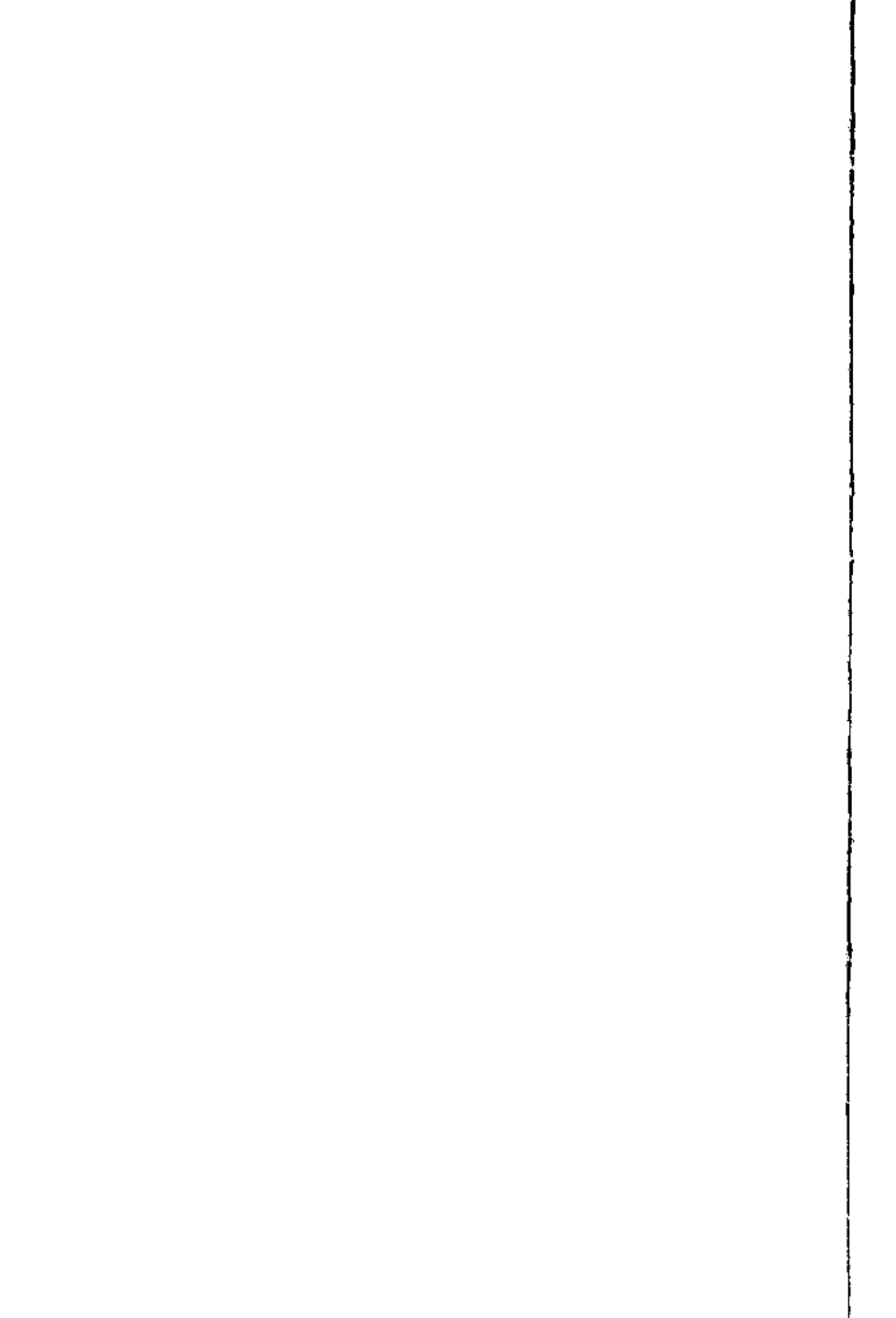
```

# PRACTICAL PROGRAMS

for the BBC Computer and ACORN ATOM

David Johnson-Davies

 Sigma Technical Press



# PRACTICAL PROGRAMS

for the BBC Computer and ACORN ATOM

David Johnson-Davies

 Sigma Technical Press

Copyright © 1982, David Johnson-Davies

All Rights Reserved

No part of this book may be reproduced by any means without the prior permission of the copyright holder. The only exceptions are as provided for by the Copyright (photocopying) Act or for the purposes of review or in order for the software herein to be entered into a computer for the sole use of the owner of this book.

ISBN: 0 905104 14 5

Published by:

Sigma Technical Press, 5 Alton Road, Wilmslow, Cheshire, UK.

Distributed by:

in Europe and Africa-

John Wiley & Sons Ltd, Baffins Lane, Chichester, Sussex, England.

in Australia, New Zealand, South East Asia-

Jacaranda-Wiley Ltd, Jacaranda Press, John Wiley & Sons Inc, GPO Box 859, Brisbane, Queensland 4001, Australia.

# Introduction

The aim of this book is to present a collection of programs that will both be useful in their own right, and illustrate a range of interesting, and in many cases classical, programming problems. The topics cover the entire range of computer applications; from mathematics and graphics, to games, and include a project to develop a compiler for a simple programming language.

Originally the programs were to be presented for one computer, with notes on converting them to run on other machines. However, this soon proved to be impractical due to the wide range of BASIC dialects. Therefore the present format of the book evolved. Each program is presented in two versions; one for the BBC Computer, whose BASIC seems likely to become a standard for the next few years, and the other for the Acorn Atom, its popular predecessor. Readers with other machines should have little trouble converting one of these versions to their particular BASIC.

Finally, I hope that the reader enjoys trying out the programs in the following pages, and is inspired to improve on them, or use them as the basis for more ambitious projects. To this end, many of the programs include a section of 'Further Suggestions'.

## Notes on the Programs

All of the programs in this book will run on the Model A BBC Computer, with 16K of RAM, although the programs of Chapter 2 can be modified to take advantage of the higher-resolution graphics available on the Model B.

The versions for the Atom will run on a machine with 1K of graphics memory, with the exception of the programs of Chapter 2, and the Compiler program, which require the full 6K of graphics memory. A floating-point ROM is needed for Rotation, Surface, and Fractions.

### Acknowledgements

I would like to thank all the people who helped in the preparation of this book, including Roger Wilson for several useful comments, Tim Dobson and Jonathan Griffiths for assisting in converting the programs to BBC BASIC, and Dorothy Armstrong for proof-reading the manuscript.

For help with particular programs I am indebted to the following: Chris Cant, for the Surface program; Nick Toop, for the programs on which Anagrams, Buzz Phrases, and Patterns are based; and David Deutsch, for suggesting improvements to the Fractions program and for commenting on an earlier draft of the book.

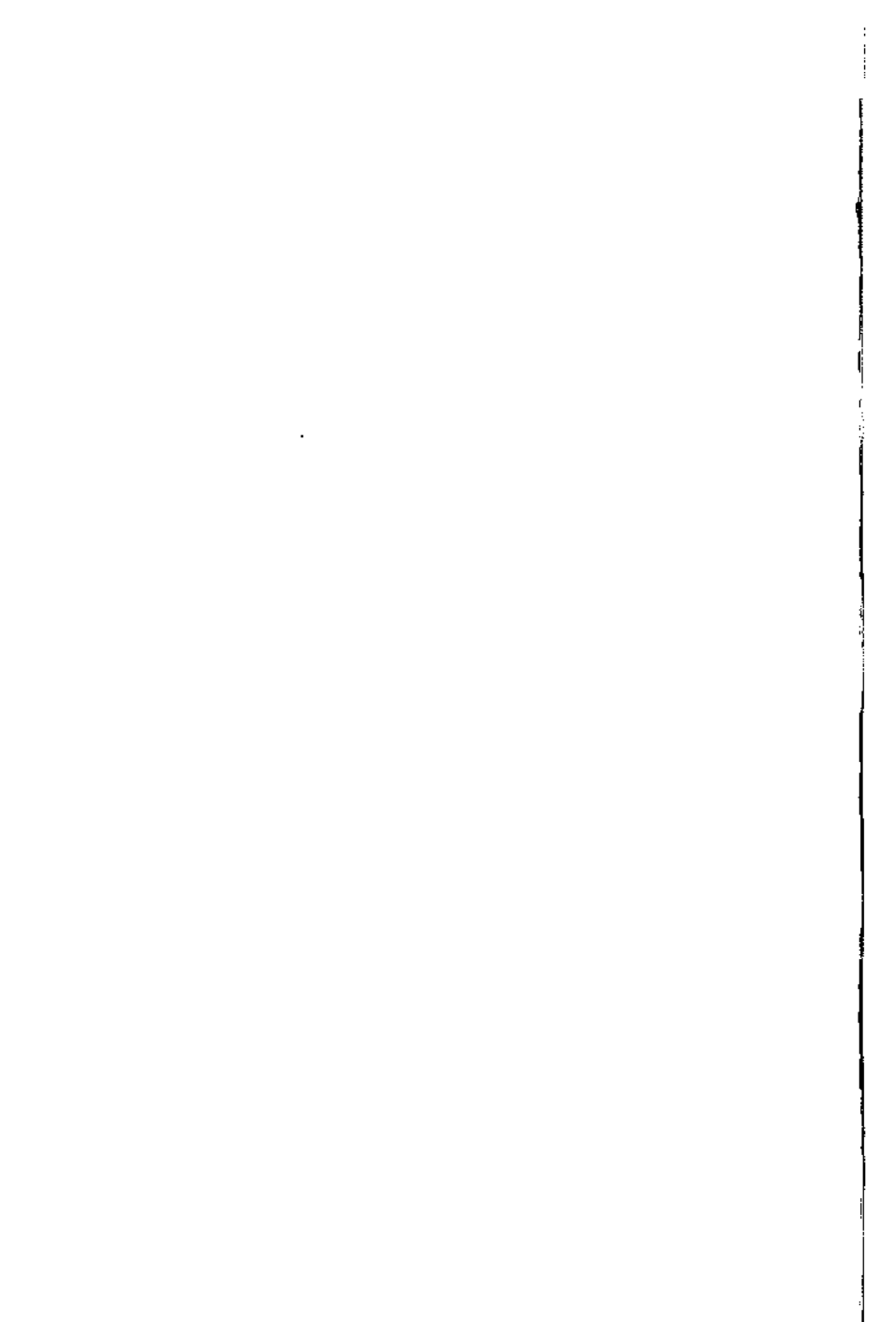
The book was prepared using word processors on Acorn computers, and I would like to thank Acorn for the help provided during this project.

David Johnson-Davies

January 1982

# Contents

Introduction -	3
Chapter 1 - GAMES	7
Silver-Dollar Game	7
Car Maze	13
Card Trick	16
Chapter 2 - GRAPHS	21
Patterns	21
Contours	24
Rotation	25
Surface	29
Chapter 3 - WORDS	35
Anagrams	35
Buzz-Phrases	39
Limericks	44
Catalogue	49
Chapter 4 - NUMBERS	57
Fractions	57
Polynomials	61
Calculator	66
Chapter 5 - COMPILER	73
Assignment	74
SPL	85
SPL Programs:	91
Bubble Sort	91
Crawling Snake	92
Primes	93
Write Hexadecimal	93
Greatest Common Divisor	94
Multiply	94
Mastermind	95
Compiler	96
Bibliography -	119





# 1 Games

SILVER-DOLLAR GAME
--------------------

Many two-player games are games of 'complete information' in which both players know at every stage what the outcome of their alternative moves will be. Thus noughts-and-crosses is such a game, but bridge is not because each player's cards are hidden from the other players. Games of complete information rely for their interest on their complexity; otherwise it would be possible for the players to work out from an early position in the game who has the forced win and how to achieve it.

In theory, one should be able to play any game of complete information perfectly. The strategy is simple: for every possible move at a given stage, examine every possible reply of your opponent's; for every reply look at every possible next move, and so on. In practice most games contain a prohibitively large number of possibilities, ruling out this strategy for even the fastest computers.

Many games of complete information have been 'solved'; in other words, a simple winning strategy has been found which, if known to one of the players, reduces the task of winning to a straightforward calculation. The following game, known as the 'Silver Dollar Game', is one such game, and it provides an excellent example of a game that a computer can play perfectly with a simple winning strategy.

The original Silver Dollar game is played with a number of coins which are moved by the two players along a line of squares. In his turn, a player must move one coin to the left along as many unoccupied squares as he wishes. For example, a possible move would be:



The first player unable to move, when all the coins have reached the left-hand end of the line, loses.

In this version of the game the computer is one player; there are five coins identified with the letters A to E, and they move along a line of 30 dots. The starting position is chosen by the program at random, and the player moves a letter by typing the letter once for each place it is to be moved. The move is entered by typing 'return', and the computer will then make its move. The game continues in this way until one player has won.

For example, the game might start with the position:

AB.....C.....D.....E.....

where the dots represent empty squares. The human player moves the 'D' to:

AB.....CD.....E.....

The computer then moves the 'E':

AB.....CD.....E.....

The human then moves the 'C':

ABC.....D.....E.....

The game continues: computer:

ABC.....DE.....

Human:

ABCD.....E.....

Computer:

ABCDE.....

and the computer wins!

### Program Operation

The strategy used by the computer relies on the fact that there are certain 'safe' positions in the game. No move by the opponent from one of these safe positions can achieve a safe position, so the computer simply waits until the player leaves an unsafe position, and thereafter always moves to safe positions until it has won.

To see if any position in the Silver Dollar Game is safe or unsafe, first determine the following three

numbers:

- The number of dots to the left of A.
- The number of dots between B and C.
- The number of dots between D and E.

Now split these numbers into their component powers of two. If there is an even number of each power of two, the position is safe; otherwise, it is unsafe.

As an example, take the following position:

```
...A...B.....C...D....E
  \ /      \ /      \ /
   3       5       4
  =1+2    =1+4    =4
```

Since there is only one '2', the position is unsafe. However, it can be made safe by moving the A two places to the left:

```
.A.....B.....C...D....E
 |         \ /      \ /
 1         5       4
-1        =1+4    =4
```

The final safe position is the winning position, when all the letters are moved to the left.

### HBC Computer Version

```
5 REM ... Silver Dollar Game ...
10 DIM PP(5),TT(2)
20 PP(0)=-1: FOR N=1 TO 5: PP(N)=PP(N-1)+ABSRND MOD
8+1: NEXT
25 P=R:CLS:PROCPRINT
```

Main loop; keep playing until someone has won.

```
30 REPEAT
35 IF FNWIN PRINT "I WIN!":END
40 PROCMOVE
50 IF FNWIN PRINT "YOU WIN!":END
60 PROC I:UNTIL 0
```

PROCI - Computer's move. Get Nim-sum of gaps, S. If it is zero there is no winning move; otherwise look for move that makes it zero.

```
1000 DEF PROC I S=0:FOR N=0 TO 2
1010 TT(N)=PP(2*N+1)-PP(2*N)-1:S=S EOR TT(N):NEXT
1020 IF S=0 GOTO 1100
1030 N=-1:REPEAT N=N+1: T=TT(N)-(S EOR TT(N))
```

```
1040 UNTIL T>0:N=2*N+1
1050 GOTO 1200
```

Human has got a safe position, so can only make a random move.

```
1100 N=0:REPEAT N=N+1:UNTIL PP(N)-PP(N-1)>=2
1120 T=(PP(N)-PP(N-1)) DIV 2
```

Computer has decided its move - now do it.

```
1200 PROCURSOR
1250 PP(N)=PP(N)-T
1260 FOR J=1 TO T
1270 FOR K=1TO999:NEXT:PROCBUDGE:NEXT:PRINT
CHR$(30);:ENDPROC
```

PROCPRINT - Print board with counters.

```
2000 DEF PROCPRINT
2005 N=0:PRINT CHR$(30)" ";:FOR J=1 TO 5
2010 IF N<PP(J) REPEAT PRINT". ";:N=N+1:UNTIL N=PP(J)
2020 PRINT CHR$(J+ASC("@"));:N=N+1:NEXT:REPEAT
PRINT". ";:N=N+1:UNTIL N=30
2030 PRINT CHR$(30);:ENDPROC
```

PROCMOVE - Input human's move, and move selected counter. Bleep illegal move.

```
3000 DEF PROCMOVE
3010 Q=GET:IF Q<ASC("A")OR Q>ASC("E")PRINT CHR$(7);:
GOTO 3010
3050 N=Q-ASC("@")
3060 IF PP(N)-PP(N-1)<2 PRINT CHR$(7);:GOTO 3010
3070 PROCURSOR
3075 GOTO 3100
3080 Q=GET:IF Q=13 ENDPROC
3090 IF PP(N)-PP(N-1)<2 PRINT CHR$(7);:GOTO 3080
3100 PP(N)=PP(N)-1
3110 PROCBUDGE:GOTO 3080
```

FNWIN - Returns 1 if game is won.

```
4000 DEF FNWIN W=1:FOR N=1TO5:IFPP(N)<>N-1 W=0
4100 NEXT:=W
```

PROCURSOR - Moves cursor to under piece N.

```
5000 DEF PROCURSOR
5010 PRINT CHR$(30);:FORJ=0 TO PP(N):PRINT CHR$(9);:
NEXT:ENDPROC
```

PROCBDUDGE - Move piece N back one place.

```
6000 DEF PROCBDUDGE
6010 PRINT". ";CHR$(8);CHR$(8); CHR$(N+ASC("@"));
CHR$(8);:ENDPROC
```

Variables:

J - Counter

K - Delay counter

N - Counter

PP(0)..PP(5) - PP(0)=-1; PP(1) to PP(5) are the positions of the 5 counters

Q - Character read by GET

S - Nim sum

S - Move needed to make Nim sum zero

TT(0)..TT(2) - Sizes of Nim-heaps corresponding to a position

W - Winner flag; W=1 if game has ended

Atom Version

```
5 ... SILVER DOLLAR GAME ...
10 DIM PP5,TT2,Q0,R-1
20 PP0=-1; FOR N=1 TO 5; PPN=PP(N-1)+ABSRND%8+1;
NEXT
```

Assemble read-character routine at R.

```
25 P=R;PRINT$21;[JSR#FPE3;STAQ;RTS;]
28 PRINT$6$12;GOSUB p
```

Main loop; keep playing until someone has won.

```
30aGOSUB w
35 IFW PRINT'"I WIN!";END
40 GOSUB m;GOSUB w
50 IFW PRINT'"YOU WIN!";END
60 GOSUB i;GOTO a
```

i - Computer's move. Get Nim-sum of gaps, S. If it is zero there is no winning move; otherwise look for move that makes it zero.

```
1000iS=0;FOR N=0 TO 2
1010 TTN=PP(2*N+1)-PP(2*N)-1;S=S:TTN;NEXT
1020 IF S=0 GOTO r
1030 N=-1;DO N=N+1; T=TTN-(S:TTN)
1040 UNTIL T>0;N=2*N+1
1050 GOTO j
```

Human has got a safe position, so can only make a random move.

```
1100rN=0;DO N=N+1;UNTIL PPN-PP(N-1)>=2
1120 T=(PPN-PP(N-1))/2
```

Computer has decided its move - now do it.

```
1200jGOSUB c
1250 PPN=PPN-T
1260 FOR J=1 TO T
1270 FOR K=1TO999;NEXT;GOSUB b; NEXT; PRINT$30;RETURN
```

p - Print board with counters.

```
2000pN=0;PRINT$30 " ";FOR J=1 TO 5
2010 IF N<PPJ DOPRINT ".";N=N+1;UNTILN=PPJ
2020 PRINT$(J+CH"@");N=N+1;NEXT;DOPRINT ".";N=N+1;
UNTILN=30
2030 PRINT$30;RETURN
```

m - Input human's move, and move selected counter.  
Bleep illegal move.

```
3000mLINKR;IF?Q<CH"A"OR?Q>CH"E" PRINT$7;GOTO m
3050 N=?Q-CH"@
3060 IF PPN-PP(N-1)<2 PRINT$7;GOTO m
3070 GOSUB c
3075 GOTO v
3080qLINKR;IF?Q=13 RETURN
3090 IF PPN-PP(N-1)<2 PRINT$7;GOTO q
3100vPPN=PPN-1
3110 GOSUB b;GOTO q
```

w - Sets W to 1 if game is won.

```
4000wW=1;FOR N=1TO5;IFPPN<>N-1 W=0
4100NEXT;RETURN
```

c - Moves cursor to under piece N.

```
5000cPRINT$30;FORJ=0 TO PPN;PRINT$9;NEXT;RETURN
```

b - Move piece N back one place.

```
6000bPRINT"."$8$8$(N+CH"@")$8;RETURN
```

Variables:

J - Counter  
K - Delay counter  
N - Counter

PP(0)..PP(5) - PP(0)=-1; PP(1) to PP(5) are the positions of the 5 counters  
 Q - Location containing character read by R  
 R - Read-character routine; puts character in ?Q  
 S - Nim sum  
 T - Move needed to make Nim sum zero  
 TT(0)..TT(2) - Sizes of Nim-heaps corresponding to a position  
 W - Winner flag; W=1 if game has ended

CAR MAZE

The following game is more a test of rapid thinking than a game of strategy or tactics; you have to drive a car through a maze, which moves steadily up the screen. The maze is randomly generated, but there is always a safe path if you can find it in time. You have controls to move the car forwards down the screen, and to the left or right. If you collide with one of the walls you must start again! To make the game even more difficult the maze moves progressively faster as the game proceeds.

### BBC Computer Version

The BBC Version uses the following controls:

Z - move left      X - move right  
                   / - move forwards

```

5 REM ... Car Maze ...
10 H=0:I=6:L=40
12 L%=-226:R%=-195:D%=-233
15 MODE7:VDU28,0,24,39,0
  
```

Set up strings for walls.

```

20 DIM W$(5)
22 W$(0)=" *           *           *           *           "
23 W$(1)=" *           *           *           *           "
24 W$(2)=" * ***** * ***** * ***** "
25 W$(3)=" ***** ***** ***** ***** "
26 W$(4)=" * ***** ***** ***** * "
27 W$(5)=" * * ***** * * * "
  
```

Start of main loop here; start car off on line 24 in column 20.

```

30 REPEAT CLS:PRINTTAB(0,24);
  
```

```

35 Y=20:@%=6:G=0:E=200:V=0
40 X=&7FC0:B=X
50 IFY?X<>32 GOTO 200
60 Y?X=&7F:FOR J=0TO E:NEXT
61 IF G AND 1 GOTO 100
62 PRINT CHR$(10);:X=X-L
63 IF V=0 C=RND(4)+1: D=RND(2)-1:$B=W$(C):V=4:GOTO
100
65 $B=W$(D):V=V-1
100 B?39=32:G=G+1:IFE>0 E=E-1
105 Y?X=32

```

Look for keys; Left decreases Y, Right increases Y, and Down adds L to X.

```

110 IF INKEY(L%) Y=(Y+39)MOD 40
120 IF INKEY(R%) Y=(Y+1)MOD 40
130 IF NOT INKEY(D%) GOTO 50
140 IFY?X<>32 GOTO 200
150 IFX<B X=X+L
160 GOTO 50

```

Crash - bleep, and put up score. Wait for space to play again.

```

200 Y?X=152:PRINT CHR$(7);CHR$(30);"Score",G,
    Highscore",H:IF G>H H=G
220 REPEAT UNTIL GET$=" "
230 UNTIL 0

```

#### Variables:

@ - Numerical field width  
B - Address of bottom line of screen  
D% - Number of / key  
E - Speed. E=0 is maximum speed  
G - Score  
H - Highest score  
I - Number of different walls  
J - Delay counter  
K - Key typed  
L - Screen width  
L% - Number of Z key  
R% - Number of X key  
V - Counter for vertical walls  
W\$(0)..W\$(4) - Strings containing walls  
X - Address of start of line containing car  
Y - Position of car across screen

#### Atom Version

The version for the Atom uses the following keys, which can be read easily from a BASIC program:



SHIFT - move left      REPT - move right  
CTRL - move forwards

```
5 REM ... CAR MAZE ...  
10 H=0;I=6;L=32
```

Set up strings for walls.

```
20 DIM W(I*L)  
22 $W=" * * * *"  
23 $W+32=" * * * *"  
24 $W+64=" * * * * * * * *"  
25 $W+96=" * * * * * * * *"  
26 $W+128=" * * * * * * * *"  
27 $W+160=" * * * * * * * *"  
28 FORN=0TO L*I;IF W?N=#2A W?N=#FF  
29 NEXT
```

Start of main loop here; start car off on line 24  
in column 20.

```
30sPRINT$12;?#E1=0;PRINT"....."  
35 Y=16;@=6;G=0;E=200;V=0  
40 X=#81E0;B=#81E0  
50zIFY?X<>L GOTO x  
60 Y?X=160;FOR J=0TO E;NEXT  
61 IF G&l GOTO v  
62 PRINT$10;X=X-L  
63 IF V=0 C=ABSRND%4+2; D=ABSRND%2;$B=$(W+C*L);V=4;  
GOTO v  
65 $B=$(W+D*L);V=V-1  
100v?#81FF=L;G=G+1;IFE>0E=E-1  
105 Y?X=L
```

Look for keys; Left decreases Y, Right increases  
Y, and Down adds L to X.

```
110 IF?#B001<128 Y=(Y-1)&31  
120 IF?#B002&#40=0 Y=(Y+1)&31  
130 IF?#B001&#40<>0GOTO z  
140 IFY?X<>L GOTO x  
150 IFX<B X=X+L  
160 GOTO z
```

Crash - bleep, and put up score. Wait for space to  
play again.

```
200xY?X=152;PRINT$7$30"score"G"    highscore"H;IF G>H  
H=G  
210 LINK#FFE3;GOTO s
```

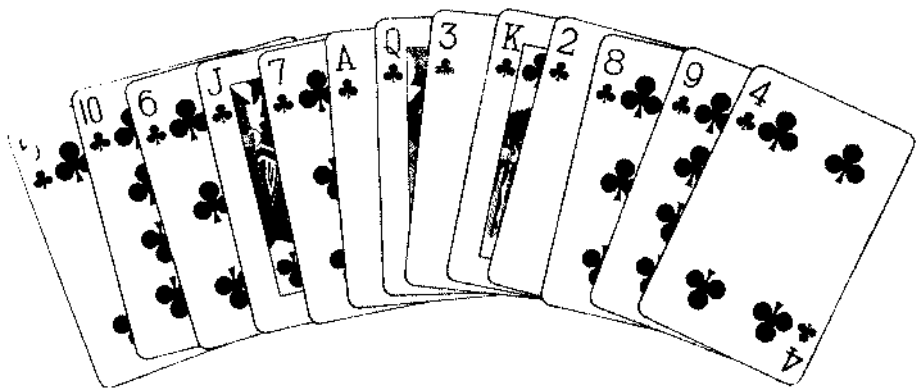
Variables:

- @ - Numerical field width
- B - Address of bottom line of screen
- E - Speed. E=0 is maximum speed
- G - Score
- H - Highest score
- I - Number of different walls
- J - Delay counter
- L - Screen width
- V - Counter for vertical walls
- W - String containing walls.  $$(W+C*L)$  is wall C
- X - Address of start of line containing car
- Y - Position of car across screen

CARD TRICK

Computers seem like magic to many people, but the following program goes one stage further and turns the computer into a magician enabling it to find a card chosen secretly by an onlooker. The presentation of the trick is as follows: the thirteen cards of one suit are fanned out face down, and the onlooker takes one and remembers it. The remaining pile of cards is cut once, and the onlooker then replaces the chosen card wherever he likes. The pile is then divided in two, and the two halves are shuffled together. Finally, the cards are fanned out face-up on the table, and the order of the cards is typed into the computer, representing Ace as 1, Jack as 11, Queen as 12, and King as 13. After a brief pause the computer announces which was the chosen card! The trick can be repeated any number of times, and the computer will almost always be right.

For example, suppose the cards are in the sequence shown below:



Having typed in the sequence of cards the program will print:

YOU PICKED THE 3

The trick depends for its success on the cards being shuffled only once, and the shuffle should be of the sort that divides the packet into two halves, and merges the two halves back into one pile; this shuffle is sometimes called a riffle shuffle. The cards can be cut at any time, but only into two piles.

### Program Operation

The program works by comparing the new order of the cards with their order the previous time the trick was performed; for each card a number is calculated which represents how far the processes of shuffling and cutting have moved that card from its previous neighbours. The higher this score, the more out of sequence is the card concerned. The card with the highest score is likely to be the one that was chosen.

When the programs are first executed they assume that the cards were in numerical order, ace up to king. If the cards are not in order when the trick is performed the computer will, most likely, get the trick wrong at the first attempt, but in some ways this adds to the mystery and can be attributed to "warming up"! Subsequently, the initial order is replaced by the new order of the cards, as typed in; therefore the order of the cards should not be disturbed between presentations of the trick.

There are cases in which the computer cannot be certain about which card was the chosen one. For example, if the card is returned to its original position then no information is available to the computer. Less obviously, if the card is replaced next

to its previous neighbour, it is ambiguous whether it or its neighbour was the chosen card. Cutting the pack before asking the onlooker to replace the card encourages him to replace the card in a different position, minimising the chances of these events occurring.

### BBC Computer Version

```
10 REM ... Card Trick ...
20 DIM A(13),B(13),S(13)
```

The cards are represented by the numbers 1 (for ace) to 13 (for king). First time, assume cards in order. Then the sequence of cards is read in.

```
30 FOR J=1 TO 13: A(J)=J: NEXT J
40 PRINT "ENTER YOUR CARDS"
50 FOR J=1 TO 13: S(J)=0
60 INPUT B(J): NEXT J
```

Each of the cards in the previous sequence A(J) is searched for in the new sequence, and its position there is subtracted from the positions of each of the cards that were its neighbours in the previous sequence. The sequences are considered as circular, so 13 is added to any difference that turns out negative.

```
70 T=A(13): PROCFIND
80 FOR J=1 TO 13: L=X: R=T
90 T=A(J): PROCFIND
100 Q=X-L: IF Q<0 THEN Q=Q+13
```

The cards distance from one neighbour, plus its distance from the other neighbour, is saved as that card's score.

```
110 S(T)=S(T)+Q: S(R)=S(R)+Q
120 NEXT J
```

The card with the maximum score is found and displayed as the chosen card.

```
130 M=0
140 FOR J=1 TO 13: A(J)=B(J)
150 IF S(J)>=M THEN Z=J: M=S(J)
160 NEXT J
170 PRINT "YOU PICKED THE "; Z
180 GOTO 40
```

```
PROCFIND - Find card T in array B, and return  
number in X.
```

```
200 DEF PROCFIND  
210 FOR K=1 TO 13: IF T=B(K) THEN X=K  
220 NEXT K: ENDPROC
```

Variables:

A(1)..A(13) - Old array of cards  
B(1)..B(13) - New array of cards  
I - Counter  
M - Maximum score  
S(1)..S(13) - Scores for each card

**Atom Version**

```
10 REM ... CARD TRICK ...  
20 DIM AA(13),BB(13),SS(13); @=0
```

```
The cards are represented by the numbers 1 (for  
ace) to 13 (for king). First time, assume cards in  
order. Then the sequence of cards is read in.
```

```
30 FOR J=1 TO 13; AA(J)=J; NEXT J  
40 PRINT "ENTER YOUR CARDS"  
50 FOR J=1 TO 13; SS(J)=0  
60 INPUT B; BB(J)=B; NEXT J
```

```
Each of the cards in the previous sequence AA(J)  
is searched for in the new sequence, and its  
position there is subtracted from the positions of  
each of the cards that were its neighbours in the  
previous sequence. The sequences are considered as  
circular, so 13 is added to any difference that  
turns out negative.
```

```
70 T=AA(13); GOSUB f  
80 FOR J=1 TO 13; L=X; R=T  
90 T=AA(J); GOSUB f  
100 Q=X-L; IF Q<0 THEN Q=Q+13
```

```
The cards distance from one neighbour, plus its  
distance from the other neighbour, is saved as  
that card's score.
```

```
110 SS(T)=SS(T)+Q; SS(R)=SS(R)+Q  
120 NEXT J
```

The card with the maximum score is found and displayed as the chosen card.

```
130 M=0
140 FOR J=1 TO 13; AA(J)=BB(J)
150 IF SS(J)>=M THEN Z=J; M=SS(J)
160 NEXT J
170 PRINT "YOU PICKED THE " Z'
180 GOTO 40
```

F - Find card T in array BB, and return number in X.

```
210fFOR K=1 TO 13; IF T=BB(K) THEN X=K
220 NEXT K; RETURN
```

Variables:

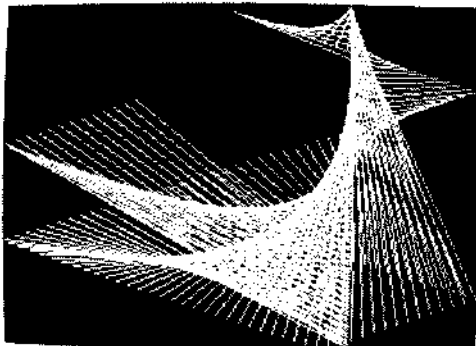
AA(1)..AA(13) - Old array of cards  
B - Card entered  
BB(1)..BB(13) - New array of cards  
J - Counter  
M - Maximum score  
SS(1)..SS(13) - Scores for each card

## 2 Graphs

### PATTERNS

Some very beautiful patterns can be generated from a very simple set of rules. The following program creates a variety of unpredictable patterns, containing many curves, simply by drawing a series of line segments. The lines are drawn between two points which move in straight lines across the screen. When either point reaches the edge of the screen it is reflected, like a billiard ball. On reaching a certain complexity the pattern is undrawn, line by line, resulting in startling effects as parts of the pattern are eliminated before others.

Sample plot:



### BBC Computer Version

The BBC Computer version uses integer variables for maximum speed, and the 4-colour graphics mode

available on the Model A. Successive lines are drawn in different colours to give a rainbow effect.

```
10 REM ... Patterns ...
20 C%=1
30 DIM XX%(3),ZZ%(3),VV%(3),CC%(3),WW%(3)
```

Choose 160x256 4-colour graphics mode. Define logical colour zero (background) as physical colour 4 (blue), and logical colour 3 (white) as physical colour 2 (green); turn cursor off.

```
40 REPEAT MODE5
45 VDUL9,0,4;0;19,3,2;0;23;&200A;0;0;0;
50
CC%(0)=1280:CC%(1)=1024:CC%(2)=CC%(0):CC%(3)=CC%(1)
60 FOR I%=0TO3:XX%(I%)=ABSRND:VV%(I%)=ABSRND MOD4*8
70 XX%(I%)=XX%(I%) MOD
CC%(I%):ZZ%(I%)=XX%(I%)+CC%(I%):NEXT
80 FOR G%=5TO7 STEP2
```

Map two pairs of coordinates (ZZ(0),ZZ(1)) and (ZZ(2),ZZ(3)) onto screen window as (WW(0),WW(1)) and (WW(2),WW(3)).

```
90 REPEAT FOR I%=0TO3:ZZ%(I%)=ZZ%(I%)+VV%(I%)
100 WW%(I%)=ABS(ZZ%(I%)MOD(CC%(I%)*2)-CC%(I%))
110 NEXT
120 C%=C%MOD3+1:GCOL0,C%
130 MOVE WW%(0),WW%(1):PLOTG%,WW%(2),WW%(3)
```

Keep plotting in different colours until return to starting coordinates, then repeat in black to clear picture.

```
140 UNTIL WW%(0)=XX%(0) AND WW%(1)=XX%(1) AND
WW%(2)=XX%(2) AND WW%(3)=XX%(3)
150 NEXT:UNTIL 0
```

Variables:

CC%(0)..CC%(3) - Screen coordinates

C% - Current colour 0-3

G% - Plotting code; G%=5 gives plotting in white, and G%=7 gives plotting in black

I - Index for arrays

VV%(0)..VV%(3) - Vectors by which the two balls move at each stage

WW%(0)..WW%(3) - Coordinates of two bouncing balls

XX%(0)..XX%(3) - Starting coordinates of balls

ZZ%(0)..ZZ%(3) - Coordinates of balls before mapping them to the screen; all positive



## Atom Version

The Atom version uses the 256x192 graphics mode, and plots the lines in white.

```
10 REM ... PATTERNS ...
15 DIM XX3,ZZ3,VV3,CC3,WW3
20 DO CLEAR4
25 CC0=256;CC1=192;CC2=CC0;CC3=CC1
30 FOR I=0TO3;XXI=ABSRND;VVI=ABSRND*4*2
40 XXI=XXI*CCI;ZZI=XXI+CCI;NEXT
45 FOR G=5TO7 STEP2
```

Map two pairs of coordinates (ZZ0,ZZ1) and (ZZ2,ZZ3) onto screen window as (WW0,WW1) and (WW2,WW3).

```
50 DO FOR I=0TO3;ZZI=ZZI+VVI
55 WWI=ABS(ZZI*(CCI*2)-CCI)
60 NEXT
65 MOVE WW0,WW1;PLOTG,WW2,WW3
```

Keep plotting in white until return to starting coordinates, then repeat in black to clear picture.

```
70 UNTIL WW0=XX0 AND WW1=XX1 AND WW2=XX2 AND
WW3=XX3
90 NEXT;UNTIL 0
```

### Variables:

CC0..CC3 - Screen coordinates  
C - Plotting code; G=5 gives plotting in white, and G=7 gives plotting in black  
I - Index for arrays  
VV0..VV3 - Vectors by which the two balls move at each stage  
WW0..WW3 - Coordinates of two bouncing balls  
XX0..XX3 - Starting coordinates of balls  
ZZ0..ZZ3 - Coordinates of balls before mapping them to the screen; all positive

### Further Suggestions

A variant on these patterns can be obtained by replacing line 45 with:

```
45 FOR H%=0TO1;G%=6
```

or its equivalent on the Atom. The lines will then be drawn by inverting the screen.

The spacing of lines on the screen is determined by the values of VV%(0) to VV%(3) set in line 30. In

the versions given above these values are constrained to be even, to limit the life of each pattern, but this can be altered to vary the spacing.

## CONTOURS

The following very simple program plots an extremely intricate and colourful pattern, from a function given in the program. In effect, the program evaluates the function at every point on the screen, and plots a coloured point the colour of which depends on the value of the function at that point. The result is a contour map of the function, with successive contour lines shown in different colours.

The plot shown below is produced by the equation  $z=x^2+y^2+xy$  (rearranged to increase the speed), which produces a series of elliptical contours. The secondary circles near the edges of the pattern are caused by an interaction between the ellipses and the screen matrix:



### BBC Computer Version

For the BBC Computer version the program uses mode 5, available on the model A, which gives 4 colours at a resolution of 160x256. Integer variables are used, for maximum speed:

```
5 REM ... Contours ...
10 MODE 5
```

Turn cursor off, then for every point on the screen plot a point whose colour depends on the function.

```
15 VDU 5
20 FOR X%=-80 TO 80: FOR Y%=-128 TO 128
30 GCOL 0,((X%*(X%+Y%)+Y%*Y%)/100) AND 3
40 PLOT69,(X%+80)*8,(Y%+128)*4
50 NEXT: NEXT
60 END
```

### Atom Version

The Atom version uses mode 4a, which gives 4 colours at a resolution of 128x192:

```
5 REM ... CONTOURS ...
10 CLEAR 4
```

For every point on the screen plot a point whose colour depends on the function.

```
20 FOR X=-64TO64: FOR Y=-96TO96
30 COLOUR((X*(X+Y)+Y*Y)/100)
40 PLOT13,(X+64),(Y+96)
50 NEXT;NEXT
60 END
```

### Further Suggestions

The following functions can be tried, in line 30 of the programs:

Function:	Contour shape:
$(X*X+Y*Y)/100$	Circles
$((X*(X+Y)+Y*Y)/100)$	Ellipses
$(X*X-Y*Y)/100$	Hyperbolae
$(X*(X+Y)-Y*Y)/100$	Bent hyperbolae

The constant, 100, can be increased to increase the width of the coloured bands.

### ROTATION

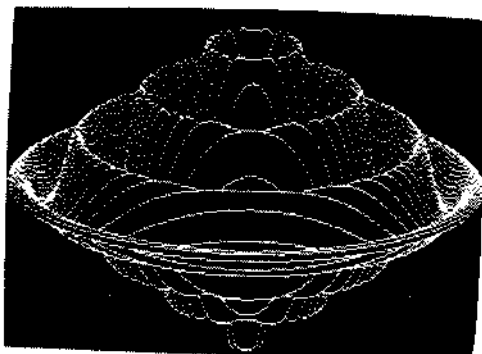
This program plots a three-dimensional view of a surface using high-resolution graphics. To give a

solid appearance to the surface, lines which lie behind the surface are not plotted; in other words, the program incorporates 'hidden-line removal'. The function for evaluation is given in line 80 of the two versions of the program, and can be any surface of revolution, in which the height, Q is simply a function of the radius from the centre, R.

As an example, the equation:

$$Q=(R-1)*\text{SIN}(24*R)$$

where the SIN function gives a rippled effect, and the (R-1) factor causes the ripples to die away towards the edge:



### BBC Version

The BBC Computer version uses the medium-resolution graphics mode, with a resolution of 320x256. The origin is set to 640,512, the centre of the virtual graphics screen.

```
1 REM ... Rotation ...
```

```
Set up graphics resolution.
```

```
10 MODE 4: VDU 29,640;512;: XS=4: YS=4  
20 A=640: B=A*A: C=512
```

```
Plot for every X coordinate.
```

```
30 FOR X=0 TO A STEP XS: S=X*X: P=SQR(B-S)  
50 FOR I=-P TO P STEP 6*YS
```

Calculate  $R$  = distance from centre and solve function;  $Q$  = height. Give  $Y$  coordinate, with correct perspective.

```
70 R=SQR(S+I*I)/A
80 Q=(R-1)*SIN(24*R)
90 Y=I/3+Q*C
```

For first point, set maximum and minimum.

```
95 IF I=-P THEN M=Y: GOTO 110
100 IF Y>M M=Y: GOTO 130
105 IF Y>=N GOTO 140
110 N=Y
```

Plot points symmetrically each side of centre.

```
130 PLOT69,-X,Y: PLOT69,X,Y
140 NEXT I: NEXT X
150 END
```

Variables:

A - X resolution  
B - Square of X resolution  
C - Y resolution  
I - Distance along X axis  
M - Highest point plotted  
N - lowest point plotted  
Q - height of function  
R - radius from centre  
X - X coordinate from centre  
Y - Y coordinate  
XS,YS - Virtual points per screen point

### Atom Version

The Atom version uses the Atom's highest graphics mode, mode 4, which has a resolution of 256x192. The program needs several changes to work with the Atom's floating-point statements. Floating-point variables prefixed by '%' must be used, although some of the variables are kept as integers since these take only integer values.

The first change is to replace the original program's FOR...NEXT loop by a floating-point DO...UNTIL loop, so that:

```
FOR I=-P TO P STEP 6*YS
.
.
NEXT I
```

in the BBC Computer version becomes:

```
%I=-%P; DO
```

```
.
```

```
%I=%I+4; FUNTIL %I>=%P
```

in the Atom version.

The IF statements in the BBC Computer version must be changed to FIF statements to ensure that the comparisons are performed on floating-point numbers. Finally, on the Atom we can take advantage of the fact that lower-case labels can be used in GOTO statements instead of line numbers.

```
1 REM ..ROTATION..
```

```
Set up graphics resolution.
```

```
10 CLEAR4
```

```
20 A=128;B=A*A;C=96
```

```
Plot for every X coordinate.
```

```
30 FOR X=0 TO A
```

```
40 S=X*X; %P=SQR(B-S); %I=-%P
```

```
Calculate %R = distance from centre and solve function; %Q = height. Give Y coordinate %Y, with correct perspective.
```

```
60 DO %R=SQR(S+%I*%I)/A
```

```
80 %Q=(%R-1)*SIN(24*%R)
```

```
90 %Y=%I/3+%Q*C
```

```
For first point, set maximum and minimum.
```

```
95 FIF %I=-%P %M=%Y;GOTO b
```

```
100 FIF %Y>%M %M=%Y;GOTO a
```

```
105 FIF %Y>=%N GOTO c
```

```
110 b %N=%Y
```

```
115 a %Y=C+%Y
```

```
Plot points symmetrically each side of centre.
```

```
120 PLOT13,(A-X),%Y; PLOT13,(A+X),%Y
```

```
135 c %I=%I+4; FUNTIL %I>=%P
```

```
145 NEXT X
```

```
150 END
```

Variables:

A X resolution  
B Square of X resolution  
C Y resolution  
I Distance along X axis  
WM Highest point plotted  
WN - lowest point plotted  
WQ - height of function  
WR - radius from centre  
X X coordinate from centre  
Y Y coordinate

### Further Suggestions

Another function can be obtained by altering lines 80 and 90 of the programs to:

```
80 Q=COS(R*5)*EXP(-R)
90 Y=I/3-Q*C-5
```

for the BBC Computer version, or for the Atom:

```
80 %Q=COS(%R*5)*EXP(-%R)
90 %Y=%I/3-%Q*C-5
```

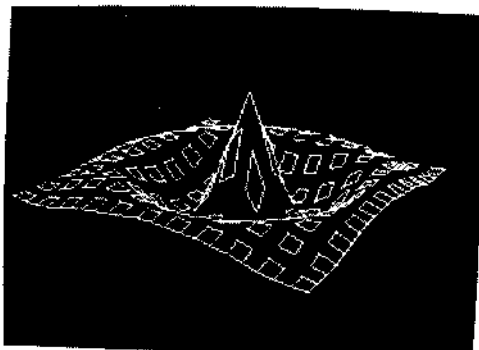
SURFACE

The last program is the most sophisticated of the graphics programs in this chapter. It gives a perspective view of a three-dimensional surface viewed from any specified position, even vertically above the surface. The program gives the appearance of a surface tiled with squares, and removes lines lying behind the surface to give a more realistic plot. The hidden-line routine has wider application, and can be used to add hidden-line removal to any three-dimensional graph.

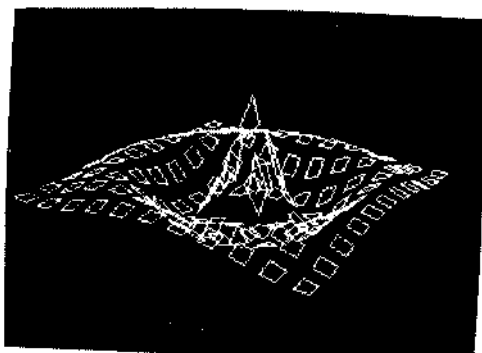
The following examples show a peaked surface produced by the equation:

$$Z = \text{COS}(O/1.5) * \text{EXP}(-O/5) * 5$$

where  $O$  is the radius from the centre of the surface. The program first prompts for the coordinates of the viewing position. The larger the coordinates are, the further the viewing position will be from the curve, and the smaller the resulting surface will be. The diagrams below are obtained with  $X=45$ ,  $Y=25$ , and  $Z=15$ :



The same curve plotted without hidden-line removal is shown below:



#### BBC Computer Version

```
10 REM ...Surface...
20 DIM Q%(319),R%(319),O%(22),P%(22)
```

Choose view position, set upper and lower horizons to top and bottom of screen, and choose 320x256 graphics mode.

```
30 INPUT"View from: ""X="L,"Y="M,"Z="N
```



```

40 S=L*L+M*M: R=SQR(S)
50 T=S+N*N: Q=SQR(T)
60 FOR I%=0 TO 319: R%(I%)=255: Q%(I%)=0: NEXT
70 E%=10: MODE4: VDU5

```

Scan plane, dividing it into squares.

```

80 FOR F%=-E% TO E%+1: FOR G%=-E% TO E%+1: U%=X%:
V%=Y%

```

Calculate O = distance from origin, and solve equation of surface in terms of O.

```

90 O=SQR(F%*F%+G%*G%)
100 X=-G%: Y=-F%: Z=cos(O/1.5)*EXP(-O/5)*5

```

Calculate O = distance of point from eye, and project point onto screen coordinates (X%,Y%).

```

110 O=(T-X*L-Y*M-Z*N)*R
120 IF O<.1 THEN 200
130 X%=400*(Y*L-X*M)*Q/O+160: IF X%<0 X%=0
140 Y%=500*(Z*S-N*(X*L+Y*M))/O+128
170 IF G%=-E% Q%(X%)=Y%: R%(X%)=Y%

```

Draw 2 sides of square on surface, and store coordinates of previous row.

```

180 IF G%+E% A%=G% AND 1: PROCDRAW
190 IF F%+E% U%=O%(G%+E%): V%=P%(G%+E%): A%=F% AND
1: PROCDRAW
200 O%(G%+E%)=X%: P%(G%+E%)=Y%
210 NEXT: NEXT
220 REPEAT UNTIL FALSE

```

PROCDRAW - Draw from U%,V% to X%,Y% with hidden-line removal. Note that line must be drawn away from observer for hidden-line removal to work correctly.

```

230 DEF PROCDRAW M%=Y%-V%: N%=ABS(X%-U%): IF N%=0
ENDPROC
240 S%=(X%-U%)/N%
250 FOR I%=1 TO N%: J%=U%+I%*S%: K%=V%+I%*M%/N%

```

If above upper horizon or below lower horizon make new horizon.

```

270 IF K%>Q%(J%) Q%(J%)=K%
280 IF K%<R%(J%) R%(J%)=K%
290 IF A%=0 THEN 320
300 MOVE4*(J%-S%),4*(Q%(J%-S%)): DRAW4*J%,4*Q%(J%)

```

```
310 MOVE4*(J%-S%),4*(R%(J%-S%)): DRAW4*J%,4*R%(J%)
320 NEXT: ENDPROC
```

#### Variables:

A% - Flag - whether to draw  
E% - Number of squares across surface  
J%,K% - Next point to be plotted by PROCDRAW  
L,M,N - Coordinates of view position  
O%,P% - Vectors holding coordinates of previous row for connecting points across. Dimension should be  $2 * E% + 1$   
Q,R,S,T - Constants for projection  
U%,V% - Previous screen coordinates to be plotted  
X%,Y% - New screen coordinates to be plotted

#### Atom Version

```
10 REM ...SURFACE...
20 DIM Q255,R255,O22,P22
```

Choose view position, set upper and lower horizons to top and bottom of screen, and choose 256x192 graphics mode.

```
30 FINPUT"VIEW FROM:""X="L,"Y="M,"Z="N
40 S=L*L+M*M;R=SQR S
50 T=S+N*N;Q=SQR T
60 FOR I=0TO255; R?I=191; Q?I=0;NEXT
70 E=10; CLEAR4
```

Scan plane, dividing it into squares.

```
80 FOR F=-E TO E+1; FOR G=-E TO E+1; U=X; V=Y
```

Calculate %O = distance from origin, and solve equation of surface in terms of %O.

```
90 %O=SQR(F*F+G*G)
100 %X=-G; %Y=-F; %Z=COS(%O/1.5)*EXP(-%O/5)*S
```

Calculate %O = distance of point from eye, and project point onto screen coordinates (X,Y).

```
110 %O=(%T-%X*L-%Y*M-%Z*N)*R
120 FIF %O<0.1 GOTO m
130 X=(400*(%Y*L-%X*M)*Q/%O)+128; IF X<0 X=0
140 Y=(500*(%Z*S-N*(%X*L+%Y*M))/%O)+96
```

Avoid plotting outside screen area.

```
180 IF Y<0 Y=0
190 IF Y>191 Y=191
```

```
200 IF G=-E Q?X=Y; R?X=Y
```

Draw 2 sides of square on surface, and store coordinates of previous row.

```
210 IF G+E A=G&1; GOSUB d
220 IF F+E U=O?(G+E); V=P?(G+E); A=F&1; GOSUB d
230M=O?(G+E)=X; P?(G+E)=Y
240 NEXT; NEXT
250 PRINT $7; DO UNTIL 0
```

d Draw from U,V to X,Y with hidden-line removal. Note that line must be drawn away from observer for hidden-line removal to work correctly.

```
400dM=Y-V; N=ABS(X-U); IF N=0 RETURN
410 S=(X-U)/N
500 FOR I=1 TO N; J=U+I*S; K=V+I*M/N
503 IFJ<0 OR J>255 GOTO e
```

If above upper horizon or below lower horizon make new horizon.

```
505 IF K>Q?J; Q?J=K
510 IF K<R?J; R?J=K
513 IF A=0 GOTO e
515 MOVE (J-S),(Q?(J-S)); DRAW J,(Q?J)
517 MOVE (J-S),(R?(J-S)); DRAW J,(R?J)
520eNEXT; RETURN
```

Variables:

G Flag - whether to draw  
E Number of squares across surface  
A,K - Next point to be plotted by PROCDRAW  
M,%M,%N - Coordinates of view position  
O,P - Vectors holding coordinates of previous row for connecting points across. Dimension should be 2\*E+1  
Q,%R,%S,%T - Constants for projection  
U,V - Previous screen coordinates to be plotted  
X,Y - New screen coordinates to be plotted

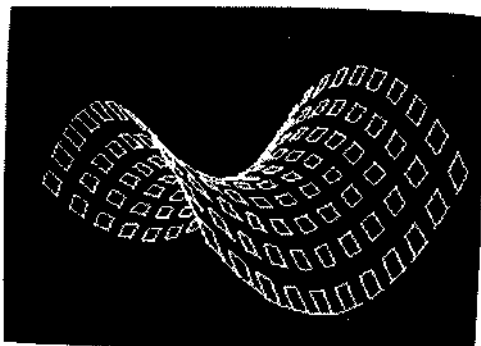
### Further Suggestions

Other pleasing plots can be obtained by altering the function in line 100 of the programs. Interesting functions are:

Function:  
 $Z=(\text{SIN}(X^2/16)+\text{SIN}(Y^2/16))$   
 $Z=.05*(X^2-Y^2)$

Shape:  
Rippled surface  
Saddle curve

A sample plot of the 'saddle curve' is shown below:



The effect of hidden-line removal can be illustrated by replacing the routine PROCDRAW in the BBC Version by:

```
230 DEF PROCDRAW MOVE U%,V%: DRAW X%,Y%: ENDPROC
```

Alternatively, in the Atom version, replace routine d by:

```
230dMOVE U,V; DRAW X,Y; RETURN
```



TOPS  
TOSP  
TPOS  
TPSO  
TSOP  
TSPO

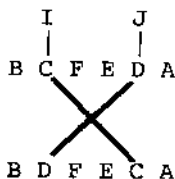
If the original string is in alphabetical order, the program will produce the permutations in alphabetical order.

### Program Operation

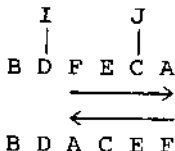
The permutation algorithm used by this program is one of the most efficient in producing a sequence of permutations in alphabetical order. It works as follows:

The first permutation is obviously the characters in alphabetical order; so, taking as an example the characters "ABCDEF", this is the first permutation. The last permutation is also obvious: it is the sequence of characters in reverse order, "FEDCBA".

To obtain the next permutation from any given sequence of letters, such as BCFEDA, we scan right-to-left looking a character that is smaller than the previous character. This will be called character 'I'. This character is then exchanged with the next higher character to its right, which will be called character 'J':



We then reverse the order of all the characters to the right of character I:



The result, "BDACEF", is the next permutation in alphabetical order.

## 100 Computer Version

```
10 REM ... Anagrams ...  
40 INPUT"STRING"A$
```

Set up array of letter positions such that  $CC\%(1)=1$ ,  $CC\%(2)=2$  ...  $CC\%(N)=N$ . Calculate number of permutations, factorial N.

```
50 N=LEN(A$):DIM CC%(N)  
60 F=1  
100 FOR J=1TON:CC%(J)=J:F=F*J:NEXT  
102 PRINT "THERE ARE" F " PERMUTATIONS"
```

Display first permutation. Then permute word, and display.

```
105 PROCDISPLAY  
110 FOR H=2TOF:PROCPERMUTE:PROCDISPLAY:NEXT  
120 END
```

PROCPERMUTE: permute word. Given any permutation from  $CC\%(1)$  to  $CC\%(N)$ , a call to PROCPERMUTE will find the next permutation.

```
200 DEF PROCPERMUTE I=N-1: J=N  
205 IF CC%(I)>=CC%(I+1) I=I-1: GOTO 205  
210 IF CC%(J)<=CC%(I) J=J-1: GOTO 210  
220 PROCSSWAP  
230 I=I+1:J=N:IF I=J ENDPROC  
240 DO PROCSSWAP: I=I+1: J=J-1:UNTIL I>=J  
250 ENDPROC
```

PROCSSWAP: Swap elements I and J.

```
300 DEF PROCSSWAP T=CC%(I):CC%(I)=CC%(J)  
310 CC%(J)=T:S=1-S:ENDPROC
```

PROCDISPLAY: Print permutation.

```
400 DEF PROCDISPLAY PRINT':FOR K=1TON  
410 PRINT MID$(A$,CC%(K),CC%(K+1));  
420 NEXT:ENDPROC
```

Variables:

A\$ Word

CC%(N) - Array being permuted; initially  $CC(N)=N$

F Factorial N, number of permutations

H Current permutation number

I, J Items to be interchanged to get new permutation

N Number of letters in word

S Sign of current permutation

## Atom Version

The Atom version is virtually identical to the BBC version given above, except that the '?' operator is used instead of MID\$ to extract characters from the word in \$A (or A\$).

```
30 DIMA(64)
40 INPUT"WORD"$A
```

Set up array of letter positions such that CC%(1)=1, CC%(2)=2 ... CC%(N)=N . Calculate number of permutations, factorial N.

```
50 N=LENA;DIM CC(N)
60 A=A-1;F=1;S=1
100 FOR J=1TON;CCJ=J;F=F*J;NEXT
102 PRINT "THERE ARE" F " PERMUTATIONS" '
```

Display first permutation. Then permute word, and display.

```
105 GOSUBd
110 FOR H=2TOF;GOSUBp;GOSUBd;NEXT
120 END
```

p - permute word. Given any permutation in CC%(1) to CC%(N), a call to subroutine p will find the next permutation.

```
200pI=N-1; J=N
205 IF CC(I)>=CC(I+1) I=I-1; GOTO 205
210 IF CC(J)<=CC(I) J=J-1; GOTO 210
220 GOSUBs
230 I=I+1;J=N;IF I=J RETURN
240 DO GOSUBs; I=I+1; J=J-1;UNTIL I>=J
250 RETURN
```

s - Swap elements I and J.

```
300sT=CCI;CCI=CCJ;CCJ=T;S=1-S;RETURN
```

d - Print permutation.

```
400dPRINT ' ;FORK=1TON;PRINT $A?CCK;NEXT;RETURN
```

Variables:

\$A - Word

CC(N) - Array being permuted; initially CC(N)=N

F - Factorial N, number of permutations

H - Current permutation number



- 1,1 - Items to be interchanged to get new permutation
- N - Number of letters in word
- S - Sign of current permutation

### Further Suggestions

The present program will give LEE twice in a list of the anagrams of the word EEL. An improvement would be to include a test for repeated letters in the original word.

Although this program will find all the permutations of a sequence of letters very rapidly, it is a very inefficient way of discovering that, for example, ORCHESTRA is an anagram of CARTHORSE because there are no less than 362880 permutations to test, and a much simpler method could be devised to verify that two words share the same letters.

### BUZZ-PHRASES

You may sometimes wonder how it is that large organizations manage to generate manuals and documents which are virtually incomprehensible to anyone who does not already know everything they are talking about. The following program may shed some light on this; it will generate unlimited technical jargon, all of which will sound plausible and can be used to pad out any manual or conference paper. The phrases have been chosen to sound as impelling as possible, but of course the program can be altered to generate random outpourings on any particular subject.

The following paragraphs were produced by successive runs of the Buzz-Phrase generator:

Similarly, a constant flow of effective information necessitates that urgent consideration be applied to the greater fight-worthiness concept.

As a resultant implication, the characterisation of specific criteria requires considerable systems analysis and trade-off studies to arrive at the philosophy of commonality and standardisation.

In respect to specific goals, the fully integrated test program must utilise and be functionally interwoven with the

structural design, based on system engineering concepts.

A particular feature of the program is that it prints the text neatly within the screen-width, without breaking words at the end of a line. This ensures that the Buzz-Phrases will be perfectly legible, even on a 40 or 32 character-per-line screen, and the routine could be useful with other text-output programs.

### BBC Computer Version

In this version the required phrase is selected by doing a RESTORE to the required line. Note that because of this the program should not be renumbered.

```
1 REM ..Buzz-Phrases..
10 W=40
```

Print out four parts of phrase, chosen at random. Wait for a key, then do another.

```
20 FOR I=100 TO 400 STEP 100
30 RESTORE(10*RND(10)+I-10):READS$
40 PROCPRETTY:NEXT I
50 PRINT". "
60 PRINT:I$=GET$:RUN
```

PROCPRETTY - Print as many words as will fit in width W, with 'return' between lines. Look for next space; concatenate "A" on string to get around INSTR bug. Put word into C\$, remainder of string back into S\$.

```
80 DEF PROCPRETTY REPEAT: A=INSTR(S$+"A", " ")
82 C$=LEFT$(S$,A): S$=MID$(S$,A+1)
84 IF A=0 C$=S$
```

If word in C\$ will not fit on the line, do a 'return'. Then print word, until all of string finished.

```
86 IF COUNT+LEN(C$) >= W PRINT
88 PRINT C$;: UNTIL A=0
90 ENDPROC
```

Choices for first part.

```
100 DATA "In particular, "
110 DATA "On the other hand, "
120 DATA "However, "
130 DATA "Similarly, "
140 DATA "As a resultant implication, "
```

- 150 DATA "In this regard, "
- 160 DATA "Based on integral subsystem considerations, "
- 170 DATA "For example, "
- 180 DATA "Thus, "
- 190 DATA "In respect to specific goals, "

Choices for second part.

- 200 DATA "a large portion of the interface coordination communication "
- 210 DATA "a constant flow of effective information "
- 220 DATA "the characterisation of specific criteria "
- 230 DATA "initiation of critical subsystem development "
- 240 DATA "the fully integrated test program "
- 250 DATA "the product configuration baseline "
- 260 DATA "any associated supporting element "
- 270 DATA "the incorporation of additional mission constraints "
- 280 DATA "the independent functional principal "
- 290 DATA "a primary relationship between system and/or subsystem technologies "

Choices for third part.

- 300 DATA "must utilise and be functionally interwoven with "
- 310 DATA "maximizes the probability of project success and minimizes the cost and time required for "
- 320 DATA "adds specific performance limits to "
- 330 DATA "necessitates that urgent consideration be applied to "
- 340 DATA "requires considerable systems analysis and trade-off studies to arrive at "
- 350 DATA "is further compounded, when taking into account "
- 360 DATA "presents extremely interesting challenges "
- 370 DATA "recognises the importance of other systems and the necessity for "
- 380 DATA "effects a significant implementation of "
- 390 DATA "adds overriding performance constraints to "

Choices for last part.

- 400 DATA "the sophisticated hardware"
- 410 DATA "the anticipated fourth generation equipment"
- 420 DATA "the subsystem compatibility testing"
- 430 DATA "the structural design, based on system

engineering concepts"

440 DATA "the preliminary qualification limit"

450 DATA "the evolution of specifications over a given time period"

460 DATA "the philosophy of commonality and standardisation"

470 DATA "the greater fight-worthiness concept"

480 DATA "any discrete configuration mode"

490 DATA "the total system rationale"

Variables:

S\$ - String for next part of phrase

C\$ - current word

I - number of DATA statement

W - screen width

A - position of space in S\$

### Atom Version

The Atom does not have READ...DATA statements, so the strings are selected by assignment statements in subroutines, using a calculated GOSUB to select the required subroutine.

```
1 REM ..BUZZ-PHRASES..  
10 DIM S(128);W=32
```

Print out four parts of phrase, chosen at random.  
Wait for a key, then do another.

```
20 FOR I=100 TO 400 STEP 100  
30 GOSUB (10*(ABSRND%10)+I)  
40 GOSUB p  
50 LINK #FPE3; GOTO 20
```

p - Print as many words as will fit in width W,  
with 'return' between lines. Look for next space;  
current word pointed to by C.

```
80pC=S; DO A=0; DO A=A+1; UNTIL C?A=CH" "
```

Break word at space; if word will not fit print  
'return'. Then print word, until all of string  
finished.

```
82 C?A=CH""; IF COUNT+LENC>=W PRINT '  
84 PRINT $C," ";C=C+LENC+1  
86 UNTIL ?C=CH""  
88 NEXT; PRINT ".";RETURN
```

Choices for first part.

100 \$\$="IN PARTICULAR, "; RETURN  
110 \$\$="ON THE OTHER HAND, "; RETURN  
120 \$\$="HOWEVER, "; RETURN  
130 \$\$="SIMILARLY, "; RETURN  
140 \$\$="AS A RESULTANT IMPLICATION, "; RETURN  
150 \$\$="IN THIS REGARD, "; RETURN  
160 \$\$="BASED ON INTEGRAL SUBSYSTEM CONSIDERATIONS,  
"; RETURN  
170 \$\$="FOR EXAMPLE, "; RETURN  
180 \$\$="THUS, "; RETURN  
190 \$\$="IN RESPECT TO SPECIFIC GOALS, "; RETURN

Choices for second part.

200 \$\$="A LARGE PORTION OF THE INTERFACE  
COORDINATION"  
201 \$\$+LENS=" COMMUNICATION "; RETURN  
210 \$\$="A CONSTANT FLOW OF EFFECTIVE INFORMATION ";  
RETURN  
220 \$\$="THE CHARACTERISATION OF SPECIFIC CRITERIA ";  
RETURN  
230 \$\$="INITIATION OF CRITICAL SUBSYSTEM DEVELOPMENT  
"; RETURN  
240 \$\$="THE FULLY INTEGRATED TEST PROGRAM "; RETURN  
250 \$\$="THE PRODUCT CONFIGURATION BASELINE "; RETURN  
260 \$\$="ANY ASSOCIATED SUPPORTING ELEMENT "; RETURN  
270 \$\$="THE INCORPORATION OF ADDITIONAL"  
271 \$\$+LENS=" MISSION CONSTRAINTS "; RETURN  
280 \$\$="THE INDEPENDENT FUNCTIONAL PRINCIPAL ";  
RETURN  
290 \$\$="A PRIMARY RELATIONSHIP BETWEEN"  
291 \$\$+LENS=" SYSTEM AND/OR SUBSYSTEM TECHNOLOGIES  
"; RETURN

Choices for third part.

300 \$\$="MUST UTILISE AND BE FUNCTIONALLY INTERWOVEN  
WITH "; RETURN  
310 \$\$="MAXIMIZES THE PROBABILITY OF PROJECT SUCCESS  
AND"  
311 \$\$+LENS=" MINIMIZES THE COST AND TIME REQUIRED  
FOR "; RETURN  
320 \$\$="ADDS SPECIFIC PERFORMANCE LIMITS TO ";  
RETURN  
330 \$\$="NECESSITATES THAT URGENT"  
331 \$\$+LENS=" CONSIDERATION BE APPLIED TO "; RETURN  
340 \$\$="REQUIRES CONSIDERABLE SYSTEMS ANALYSIS AND  
TRADE-OFF"  
341 \$\$+LENS=" STUDIES TO ARRIVE AT "; RETURN

350 \$\$S="IS FURTHER COMPOUNDED, WHEN TAKING INTO ACCOUNT "; RETURN

360 \$\$S="PRESENTS EXTREMELY INTERESTING CHALLENGES TO "; RETURN

370 \$\$S="RECOGNISES THE IMPORTANCE OF OTHER SYSTEMS AND THE"

371 \$\$S+LENS=" NECESSITY FOR "; RETURN

380 \$\$S="EFFECTS A SIGNIFICANT IMPLEMENTATION OF "; RETURN

390 \$\$S="ADDS OVERRIDING PERFORMANCE CONSTRAINTS TO "; RETURN

Choices for last part.

400 \$\$S="THE SOPHISTICATED HARDWARE "; RETURN

410 \$\$S="THE ANTICIPATED FOURTH GENERATION EQUIPMENT "; RETURN

420 \$\$S="THE SUBSYSTEM COMPATIBILITY TESTING "; RETURN

430 \$\$S="THE STRUCTURAL DESIGN, BASED ON SYSTEM ENGINEERING"

431 \$\$S+LENS=" CONCEPTS "; RETURN

440 \$\$S="THE PRELIMINARY QUALIFICATION LIMIT "; RETURN

450 \$\$S="THE EVOLUTION OF SPECIFICATIONS OVER A GIVEN TIME"

451 \$\$S+LENS=" PERIOD "; RETURN

460 \$\$S="THE PHILOSOPHY OF COMMONALITY AND"

461 S+LENS=" STANDARDISATION "; RETURN

470 \$\$S="THE GREATER FIGHT-WORTHINESS CONCEPT "; RETURN

480 \$\$S="ANY DISCRETE CONFIGURATION MODE "; RETURN

490 \$\$S="THE TOTAL SYSTEM RATIONALE "; RETURN

Variables:

- A - Pointer to space
- C - Current part of phrase
- I - Random number
- W - Screen width
- S - Phrase string

LIMERICKS

Although it will be a long time before computers can generate sentences of their own accord, it is possible to program a computer to generate sentences that will pass as meaningful provided we restrict ourselves to a small number of possibilities. To illustrate, the

following light-hearted program will construct limericks according to a set of fairly simple rules. Nevertheless, the results are sometimes surprising, and at worst amusing.

Some examples produced by the program are given below:

A GRACEFUL BLAND GROCER FROM KINGS  
ONCE DEMOLISHED SOME CAKES AND GREW WINGS  
HE DEMOLISHED SO LATE  
THAT HE LOOKED FOR A PLATE  
THIS GRACEFUL BLAND GROCER OF KINGS .

A VICIOUS YOUNG LAUNDRESS FROM SPAIN  
ONCE WANTED SOME CAKES ON A TRAIN  
SHE WANTED SO SLOW  
THAT SHE WANTED SOME DOUGH  
THIS VICIOUS YOUNG LAUNDRESS OF SPAIN .

Each word or phrase in the limerick is selected, at random, from six alternatives, each of which has the same number of syllables so that the final limerick will scan correctly. The structure of the limerick is defined as follows, where lower-case words in angled brackets, such as <adjective>, are to be replaced by the actual words selected at random by the computer:

A <adjective w> <adjective x> <noun y> FROM <place z>  
ONCE <verb g> <noun> <qualifier z>  
<pronoun y> <verb g> SO <adverb t>  
THAT <pronoun y> <verb> <noun t>  
THIS <adjective w> <adjective x> <noun y> OF <place z>

Parts of speech, such as <noun>, are replaced by a word or phrase selected at random from six possibilities. Where two parts of speech are labelled with a letter, as in <noun y> and <pronoun y>, the two words are chosen as a pair; for example, if <noun y> were "LAUNDRESS", <pronoun y> would be "SHE". Similarly, the <qualifier z> in line 2 is chosen to rhyme with <place z> in lines 1 and 5; for example, if <place z> is chosen as "FROM KINGS", the second line must end with "AND GREW WINGS". Similarly the <noun t> in line 4 is chosen to rhyme with the <adverb t> in line 3. These simple constraints are sufficient to ensure that the random limericks will rhyme and scan.

#### BBC Computer Version

First line of limerick.

```
20 P=1000:PRINT "A ";
30 PROCRRND:W=R:PROCRRND:X=R:PROCRRND:Y=R
40 PRINT "FROM ";
50 PROCRRND:Z=R:PRINT
```

Second line.

```
60 PRINT "ONCE ";:PROCRRND:G$=C$
70 PROCRRND:R=Z:PROCWORD:PRINT
```

Third and fourth lines.

```
80 R=Y:PROCWORD:H$=C$:PRINTG$;"SO ";
90 PROCRRND:T=R:PRINT"THAT
";H$;:PROCRRND:R=T:PROCWORD: PRINT
```

Last line.

```
110 PRINT"THIS ";:P=1000:R=W:PROCWORD: R=X:PROCWORD:
R=Y:PROCWORD
120 PRINT"OF ";:R=Z:PROCWORD:PRINT". "
140 END
```

PROCRRND - Choose random phrase.

```
200 DEF PROCRRND:R=ABSRND MOD 6:PROCWORD:ENDPROC
```

PROCWORD - Select Rth word in \$C and print it.

```
220 DEF PROCWORD:RESTORE P
230 FOR N=0 TO R:READ C$:NEXT:C$=C$+" "
250 PRINTC$;:P=P+100:ENDPROC
```

Strings of phrases.

```
1000 DATASORDID,GRACEFUL,WILY,VICIOUS,SPARKLING,
REALLY
1100 DATAGREEN,YOUNG,VILE,BLAND,OLD,WILD
1200 DATADUCHESS,GROCER,GLUTTON,FLAUTIST,LAUNDRESS,
SAILOR
1300 DATAWEMBLEY,SPAIN,CHAD,SPEKE,KINGS,FRANCE
1400 DATAWANTED,FOLLOWED,COUNTED,DEMOLISHED,
COLLECTED,SWALLOWED
1500 DATASOME STAMPS,A STOAT,A NUDE,SOME CAKES, A
FROG,SOME MOULD
1600 DATAAND FELT TREMBLY,ON A TRAIN,AND WENT MAD,
TWICE A WEEK,AND GREW WINGS,IN A TRANCE
1700 DATASHE,HE,SHE,HE,SHE,HE
1800 DATAQUICK,SLOW,FEW,HARD,LATE,LONG
```



1900 DATANOTICED,FOLLOWED,ASKED FOR,LOOKED FOR,  
WANTED,LONGED FOR

2000 DATAA BRICK,SOME DOUGH,A SCREW,SOME LARD,A  
PLATE,KING KONG

Variables:

G String containing verb used in line 2  
H String containing HE/SHE  
P Pointer to next selection of phrases  
R Random number 0 to 4  
S String of phrase options  
T Word selected in second line  
W,X,Y,Z - Words selected in first line

### Atom Version

The Atom version uses a string, \$\$, to store the six alternatives for a particular phrase. This string is added to the list of alternatives by a GOSUB to a line which assigns the string to \$\$ . Note that where the string will not fit onto one line the second half is concatenated onto the end of \$\$ by executing:

```
$$LENS="string"
```

on lines 2000 and 2001.

```
10 REM ... LIMERICKS ...  
15 DIM S(100),G(32),H(32)
```

First line of limerick.

```
20 P=1000;PRINT "A "  
30 GOSUB s;W=R;GOSUB s;X=R;GOSUB s;Y=R  
40 PRINT "FROM "  
50 GOSUB s;Z=R;PRINT "
```

Second line.

```
60 PRINT "ONCE ";GOSUB s;$G=$C  
70 GOSUB s;R=Z;GOSUB t;PRINT "
```

Third and fourth lines.

```
80 R=Y;GOSUB t;$H=$C;PRINT $G,"SO "  
90 GOSUB s;T=R;PRINT "THAT ",$H;GOSUB s;R=T;GOSUB  
PRINT "
```

Last line.

```
110 PRINT "THIS ";P=1000;R=W;GOSUB t;R=X;GOSUB t;  
120 GOSUB t  
130 PRINT "OF ";R=Z;GOSUB t;PRINT "."
```

" Make a random choice by extracting a substring from within the string \$\$.

```
200sR=ABSRND%6
```

t - Select the substring corresponding to the value of R. Scan past R commas, put the string from there to the next comma, or 'return', in \$C, and print it.

```
210tGOSUB (P);A=0;IF R=0 G.u
220 FOR N=1 TO R;GOSUB c;NEXT;A=A+1
240uC=A+S;GOSUB c;$$+A=" "
250 PRINT $C;P=P+100; RETURN
```

c - Search for comma or end of string.

```
300cDO A=A+1;UNTIL S?A=CH","OR S?A=CH""; RETURN
```

Strings of phrases.

```
1000 REM WORDS
1001 $$="SORDID,GRACEFUL,WILY,VICIOUS,SPARKLING,
REALLY"; RETURN
1100 $$="GREEN,YOUNG,VILE,BLAND,OLD,WILD"; RETURN
1200 $$="DUCHESS,GROCCER,GLUTTON,FLAUTIST,LAUNDRESS,
SAILOR"; RETURN
1300 $$="WEMBLEY,SPAIN,CHAD,SPEKE,KINGS,FRANCE";
RETURN
1400 $$="WANTED,FOLLOWED,COUNTED,"
1401 $$+LENS="DEMOLISHED,COLLECTED,SWALLOWED"; RETURN
1500 $$="SOME STAMPS,A STOAT,A NUDE,"
1501 $$+LENS="SOME CAKES,A FROG,SOME MOULD"; RETURN
1600 $$="AND FELT TREMBLY,ON A TRAIN,AND WENT MAD,"
1601 $$+LENS="TWICE A WEEK,AND GREW WINGS,IN A
TRANCE"; RETURN
1700 $$="SHE,HE,SHE,HE,SHE,HE"; RETURN
1800 $$="QUICK,SLOW,FEW,HARD,LATE,LONG"; RETURN
1900 $$="NOTICED,FOLLOWED,ASKED FOR,"
1901 $$+LENS="LOOKED FOR,WANTED,LONGED FOR"; RETURN
2000 $$="A BRICK,SOME DOUGH,A SCREW,"
2001 $$+LENS="SOME LARD,A PLATE,KING KONG"; RETURN
```

Variables:

A - Pointer to find commas  
 G - String containing verb used in line 2  
 H - String containing HE/SHE  
 P - Pointer to next selection of phrases  
 R - Random number 0 to 4  
 S - String of phrase options

T Word selected in second line.  
W,X,Y,Z - Words selected in first line.

CATALOGUE
-----------

The following program allows you to build up a catalogue of books, records, or telephone numbers. It illustrates how a computer can be used to store information, sort it, and retrieve it on command.

A collection of records of information on a computer is called a "database"; in the following examples we will assume that the records consist of names and telephone numbers. The program allows you to associate a telephone number with each name. You can then find out someone's telephone number by typing their name, or as many letters of their name as are needed to identify them uniquely. As an example, assume we are entering the telephone numbers of five people. The symbol '>' is used to prefix a name to be entered:

```

RUN
COMMAND?>DEWAR J
123 2234
COMMAND?>SMITH P S
0223 314341
COMMAND?>NORTH Q
119 2389 x191
COMMAND?>BOND J
456 7789
COMMAND?>WEST A
145 3456
```

We can then ask for an alphabetical list of the whole database, using the "\*" command:

```

COMMAND?*
BOND J          456 7789
DEWAR J        123 2234
NORTH Q        119 2389 x191
SMITH P S      0223 314341
WEST A         145 3456
```

Alternatively, we can ask for the telephone number of any person in the database:

```

COMMAND?NORT
NORTH Q          119 2389 x191
COMMAND?S
SMITH P S        0223 314341
```

COMMAND?DEWER  
NOT FOUND

An attempt to enter a name already entered will give a warning message:

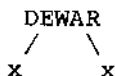
COMMAND?>BOND J  
BOND J ALREADY EXISTS

### Program Operation

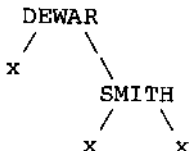
The simplest way to store the records in the computer would be as a straightforward list. However, it would then be necessary to search through the whole list every time a new record was entered, or a record was being searched for. It would also be difficult to produce an alphabetical list of records, without first sorting all the records into order, which would be very time-consuming.

The Catalogue program therefore holds the records in a more sophisticated structure, called a 'tree'. Associated with every record are two 'pointers', which can be set to point to other records, or can be marked as pointing to nothing. As new records are entered, a tree is built up. Names lying earlier in the alphabet are inserted on the left-hand side of the tree, and names later in the alphabet on the right-hand side of the tree.

To see how this works in practice, a tree is built up with the five names given above. The first record goes at the top of the tree, and its two pointers are set to zero, indicated here by 'x', to indicate that there are no further records below it:

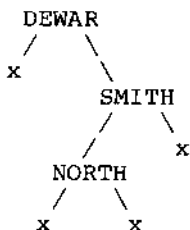


Suppose we then add SMITH. This is later in the alphabet than DEWAR, and so it is attached to the right-hand branch of the tree:

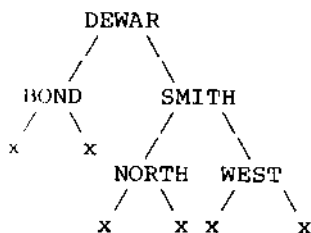


Next we add NORTH. First the name is compared with DEWAR, and since it is later in the alphabet we follow the right-hand branch. Then it is compared with SMITH. Since it is before SMITH in the alphabet we follow the

left-hand branch. We have reached the end of the tree, and so NORTH is added there:



Finally, we add BOND and WEST, and the tree looks like this:



When searching the tree for a name we follow the same procedure, until either the name is found, or the end of a branch is reached in which case the message 'NOT FOUND' is given.

The advantage gained by using a tree structure depends somewhat on the shape of the tree; with a perfectly balanced tree of 31 records, a maximum of five comparisons need to be made to find any record, as opposed to 31 with a simple list of records. As the number of records increases, the saving becomes even greater.

A tree can be printed in alphabetical order by using the following simple recursive procedure:

1. print the tree starting at a certain name

2. print the tree pointed to by the left-hand pointer,

3. print the name,

4. and print the tree pointed to by the right-hand pointer.

In the case of the tree above we obtain:

BOND, DEWAR, NORTH, SMITH, WEST

## BBC Computer Version

```
5 REM ... Catalogue ...
10 DIM M%64,T%3:Z%=&3C00:F%=&1500:HIMEM=F%
20 !T%=0
30 REPEAT PRINT CHR$(12)"command"
```

Input command line. Commands:

- \* - Print tree
- > - Add name to tree.

```
40 INPUT $M%
50 IF?M%=ASC("*")PROCPRINT(!T%):GOTO 130
60 IF?M%=ASC(">")GOTO 100
```

Name on its own - search for it. If a close match is found print it (PROCPRINT), else say 'NOT FOUND'.

```
70 N%=M%;X%=T%;S%=1:E%=0:PROCSEARCH
80 IF E% PROCPUT:GOTO 130
90 PRINT "NOT FOUND":GOTO 130
```

Add name to tree. If no match (not E) then all is well; otherwise the name already exists.

```
100 N%=M%+1:X%=T%;E%=0:S%=0:PROCSEARCH
110 IF NOT E% UNTIL FALSE
120 PRINT $N%;" ALREADY EXISTS"
130 K%=GET:UNTIL FALSE
```

PROCSEARCH - Search tree. Operation depends on value of S:  
S%=0 - Add name \$N% to tree and input telephone number when the end of a branch is reached.  
S%=1 - Search tree for name, and return on a match of first few characters.

```
1000 REM SEARCH TREE
1005 DEF PROCSEARCH
1010 IF !X%=0 GOTO 1100
```

Follow down pointer at X%, and compare name \$J on tree with \$N. If they match, return. If \$N>\$J, move X% to right-hand pointer; then keep searching down tree.

```
1020 X%=!X%:J%=X%+8:PROCCOMP
1030 IF E% ENDPROC
1040 IF C% X%=X%+4
1050 GOTO 1010
```

End of branch. If S=1 give up since the name is not found; otherwise add the name \$N to the tree here, input the telephone number.

```
1100 IF S% ENDPROC
1110 Y%=X%:!X%=F%:X%=!X%:!X%=0:X%!4=0
1120 X%=X%+8:$X%=$N%:X%=X%+LEN($X%)+1
1130 INPUT $X%:X%=X%+LEN($X%)+1
1140 IF(X% AND &FFFF)>Z% PRINT "NO
ROOM":!Y%=0:K%=GET:ENDPROC
1150 F%=X%:ENDPROC
```

PROCCOMP - Compare strings \$J% and \$N% character by character, and set C% to 1 if \$N%>\$J%. If searching the tree, settle for \$N% matching the first few characters of \$J%; if adding a name to the tree, insist on an exact match.

```
2000 REM COMPARE $J% AND $N%
2010 DEF PROCCOMP I%=-1:REPEAT I%=I%+1
2020 UNTIL J%?I%<>N%?I% OR N%?I%=13
2030 C%=(J%?I%<N%?I%)
2040 IF S% E%=(N%?I%=13):ENDPROC
2050 E%=(J%=$N%):ENDPROC
```

PROCPRINT - Print tree from Q% downwards. Print tree on left-hand branch, print record at Q%, then print tree on right-hand branch.

```
3000 REM PRINT TREE
3005 DEF PROCPRINT(Q%)
3010 IF Q%=0 ENDPROC
3020 PROCPRINT(!Q%)
3040 J%=Q%+8:PROCPUT:Q%=Q%+4
3050 PROCPRINT(!Q%):ENDPROC
```

PROCPUT - Print record at J%. Tabulate telephone number to column 20.

```
4000 REM PRINT RECORD
4010 DEF PROCPUT PRINT $J%;
4020 J%=J%+LEN($J%)+1:PRINT TAB(20);$J%;ENDPROC
```

Variables:

- \* Equal flag; E%=1 if a match is found
- \* Next free memory location
- \* Name string on tree
- \* Command line string
- \* - Name string
- \* Local parameter for PROCPRINT
- \* - Search flag: S%=1 means search tree; S%=0 means

add to tree  
T% - Pointer to top of tree  
X% - Pointer to current position on tree  
Y% - Pointer before adding name to tree  
Z% - Top of available memory

### Atom Version

This version of the catalogue program is substantially identical to the version for the BBC Computer. The only major difference is that the routine to print the tree has to save the value of X before re-entering itself recursively, since on the Atom procedures do not have local parameters.

```
5 REM ... CATALOGUE ...  
10 DIM M(64),T(3),F(-1)  
20 !T=0;Z=#3BFF  
30 DO PRINT $!2"command"
```

Input command line. Commands:

\* - Print tree  
> - Add name to tree.

```
40 INPUT $M  
50 IF?M=CH"*"X=!T;GOSUB p;GOTO n  
60 IF?M=CH">"GOTO a
```

Name on its own - search for it. If a close match is found print it (GOSUB q), else say 'NOT FOUND'.

```
70 N=M;X=T;S=1;E=0;GOSUB s  
80 IF E GOSUB q;GOTO n  
90 PRINT "NOT FOUND";GOTO n
```

a - Add name to tree. If no match (not E) then all is well; otherwise the name already exists.

```
100aN=M+1;X=T;E=0;S=0;GOSUB s  
110 IF E:1 UNTIL 0  
120 PRINT $N" ALREADY EXISTS"  
130nLINK#FFE3;UNTIL 0
```

s - Search tree. Operation depends on value of S:  
S=0 - Add name \$N to tree and input telephone number when the end of a branch is reached.  
S=1 - Search tree for name, and return on a match of first few characters.

```
1000sREM SEARCH TREE  
1010 IF !X=0 GOTO b
```



Follow down pointer at X, and compare name \$J on tree with \$N. If they match, return. If \$N>\$J, move X to right-hand pointer; then keep searching down tree.

```
1020 X=1X;J=X+8;GOSUB c
1030 IF E RETURN
1040 IF C X=X+4
1050 GOTO s
```

b - End of branch. If S=1 give up since the name is not found; otherwise add the name \$N to the tree here, input the telephone number.

```
1100bIF S RETURN
1110 Y=X;!X=F;X=!X;!X=0;X!4=0
1120 X=X+8;$X=$N;X=X+LENX+1
1130 INPUT$X;X=X+LENX+1
1140 IF X&#xFFFF>Z PRINT "NO ROOM";
1150 F=X;RETURN
```

c - Compare strings \$J and \$N character by character, and set C to 1 if \$N>\$J. If searching the tree, settle for \$N matching the first few characters of \$J; if adding a name to the tree, insist on an exact match.

```
2000cREM COMPARE $J AND $N
2010 I=-1;DO I=I+1
2020 UNTIL J?I<>N?I OR N?I=13
2030 C=(J?I<N?I)
2040 IF S E=(N?I=13);RETURN
2050 E=($J=$N);RETURN
```

p - Print tree from X downwards. First save X on stack. Print tree on left-hand branch, recover X and print record at X, then print tree on right-hand branch.

```
4000pREM PRINT TREE
4010 IF X=0 RETURN
4020 !F=X;F=F+2;X=!X;GOSUB p
4030 F=F-2;X=!F
4040 J=X+8;GOSUB q;X=X+4
4050 X=!X;GOTO p
```

q - Print record at J. Tabulate telephone number to column 20.

```
5000qREM PRINT RECORD
```

```
5010 PRINT $J;DOPRINT " ";UNTIL COUNT=20
5020 J=J+LENJ+1;PRINT $J';RETURN
```

#### Variables:

E - Equal flag; E=1 if a match is found  
F - Next free memory location  
J - Name string on tree  
M - Command line string  
N - Name string  
S - Search flag: S=1 means search tree; S=0 means add to tree  
T - Pointer to top of tree  
X - Pointer to current position on tree  
Y - Pointer before adding name to tree  
Z - Top of available memory

#### Further Suggestions

The program is not restricted to names and telephone numbers; in fact either field may be any sequence of up to 64 characters, and the whole of the first field is used for the alphabetical ordering.

A useful extension to the program would allow the database to be saved and loaded to and from tape. The values of !T% and F% (or !T and F in the Atom version) should be saved with the tree, since these give the address of the top of the tree and the next free location in memory respectively.

## 4 Numbers

It comes as no surprise that computers can perform numerical calculations. For example, most computers will allow you to type in:

```
PRINT 29*173
```

and the result 5017 will be printed. However, there are some distinct limitations to the calculations that the computer can perform unaided. For example, suppose we type:

```
PRINT 1/2 + 1/3
```

The computer will probably print 0.833333 (or even less helpfully, 0) rather than the answer we would like, namely 5/6.

Alternatively, suppose we type in:

```
PRINT (X-1)*(X*X+X+1)
```

the result will depend on the particular value of X; a more useful answer would be the simplified form of the expression,  $X^3-1$  (where '^' means "raised to the power").

Finally, suppose we enter:

```
PRINT 9999*9999*9999
```

desiring the exact answer 999700029999. In fact we will get 9.99700030E11 on the BBC Computer, or the less obvious answer 1120133679 on the Atom.

The three programs in this chapter solve these problems, and illustrate some of the different ways of representing numbers on a computer.

FRACTIONS
-----------

Most computer languages, including BASIC, provide functions and operations involving integers such as

127, or floating-point numbers such as 12.73, or both. However, on some occasions we may wish to perform calculations involving fractions, such as:

$$1/2 + 1/3$$

with the results given as an exact fraction, such as  $5/6$ , rather than the less obvious, and less accurate, decimal 0.83333333.

The following program will take a calculation involving fractions, and give an exact fractional result (where possible). For example:

```
EVALUATE: 1/4 + 5 * 7/68  
1/4 + 5 * 7/68 = 13/17
```

The result can be given either in improper form, such as  $7/6$ , or proper form, such as  $1+1/6$ . The program detects if the result cannot be expressed as a fraction with sufficient accuracy, as shown in the following two examples:

```
EVALUATE: 3.14159292  
3.14159292 = 355/113
```

```
EVALUATE: PI  
PI = IRRATIONAL
```

The program would be useful for mathematical analysis, or for teaching the concept of fractions to children.

### Description

The program evaluates the INPUT line using floating-point arithmetic, and converts the result from a floating-point number into a fraction. The method involves the construction of what is called a 'continued fraction', by repeating the following two simple steps until an integer is obtained:

- Take the reciprocal (i.e. one divided by the number)
- Subtract the integer part.

This can be illustrated by the following example, which converts 0.764705882 into a continued fraction:

$$\begin{aligned}x &= 0.764705882 \\ 1/x &= 1.30769231 \\ 1/x - 1 &= 0.307692307 \\ 1/(1/x - 1) &= 3.25 \\ 1/(1/x - 1) - 3 &= 0.25 \\ 1/(1/(1/x - 1) - 3) &= 4\end{aligned}$$

Finally, solving the equation for  $x$  in the last line gives the exact answer that  $x = 13/17$ .

## BBC Computer Version

In this version an integer variable, I%, is used to calculate the integer part of the number at each stage to avoid having to use the INT function. The remaining variables are left as floating-point variables for brevity. An "ON ERROR" statement detects syntax errors when the input equation is evaluated using EVAL.

1 REM ... Fractions ...

Set up error return for errors other than escape (17).

```
10 ON ERROR IF ERR<>17 PRINT"WHAT?":GOTO60 ELSE END
20 @%=1
```

Select whether fraction is to be displayed in proper or improper form.

```
30 REPEAT INPUT'"PROPER (P) OR IMPROPER (I)?' P$
40 UNTIL P$="P" OR P$="I"
60 REPEAT E=9E-8
70 INPUT "EVALUATE:"EQ$: PRINT EQ$;" = ";
80 S$="+": X=EVAL(EQ$): IF X<0 S$="-"
90 X=ABS X: I%=X: R=X-I%
```

Multiply accuracy by integer part. If fractional part negligible, treat as integer. If no integer part, more error tolerated.

```
95 IF I%<>0 THEN E=X*E
100 IF R<=E GOTOL10
101 IF I%=0 THEN E=E*R
102 IF ABS(R-1)>E GOTOL20
104 I%=I%+1
110 IF S$="-" PRINT S$;
112 PRINT I%: UNTIL FALSE
```

Now do fractional part.

```
120 IF S$="-" PRINT S$;
130 IF I%<>0 AND P$="P": PRINT I%,S$;: I%=0: X=R
190 K=1:L=1:M=0:J=I%
```

Work out continued fraction for R.

```
200 REPEAT R=1/R: I%=R
204 R=R-I%
210 N=J: J=J*I%+K: K=N
220 N=L: L=L*I%+M: M=N
230 UNTIL E>=ABS(X-J/L)
```

Test whether irrational. Choose 0.0033 so that PI comes out as irrational, but 355/113 is real.

```
240 IF ABS(J*L)*ABS(X-J/L)/X > 0.0033
PRINT "IRRATIONAL": UNTIL FALSE
250 PRINT J"/"L: UNTIL FALSE
```

Variables:

I% - Integer part of %X  
J - Numerator of fractional representation  
L - Denominator of fractional representation  
P\$ - Proper "P" or improper "I" flag  
R - Real part of %X  
S\$ - Sign of %X  
X - Absolute value of rational

### Atom Version

The Atom does not have an EVAL function, but fortunately the FINPUT statement in Atom BASIC will accept an expression in the input line. The input line is entered at address 320, so PRINT \$320 in line 70 will echo the line.

```
1 REM .. FRACTIONS ..
10 DIM P(1),S(1),E(100)
```

Set up line to be executed on an error.

```
20 @=0;?16=E;?17=E/256;$E="PRINT ""WHAT?""";G.S"
```

Select whether fraction to be displayed in proper or improper form.

```
30 DO INPUT "PROPER (P) OR IMPROPER (I)"$P
40 UNTIL $P="P" OR $P="I"
50$DO %E=9E-8
70 PRINT "EVALUATE"; FINPUT %X; PRINT $320 = "
80 S$="+";FIF%X<0 S$="-"
90 %X=ABS%X;I=%X;%R=%X-I
```

Multiply accuracy by integer part. If fractional part negligible, treat as integer. If no integer part, more error tolerated.

```
95 IF I<>0 THEN %E=%X*%E
100 FIF %R<=%E GOTO i
101 IF I=0 THEN %E=%E*%R
102 FIF ABS(%R-1)>%E GOTO r
104 I=I+1
110iIF S$="-"PRINT S$
```

```
112 PRINT I;UNTIL 0
```

Now do fractional part.

```
120rIF $$="-" PRINT $$
130 IF I<>0 AND $P="P" PRINT I,$$; I=0; %X=%R
180 $E="PRINT "IRRATIONAL""';GOTO s"
190 K=1;L=1;M=0;J=I
```

Work out continued fraction for %R.

```
200 DO %R=1/%R;I=%R
204 %R=%R-I
210 N=J;J=J*I+K;K=N
220 N=L;L=L*I+M;M=N
230 FUNTIL %E>=ABS(%X-J/L)
```

Test whether irrational. Choose 0.0033 so that PI comes out as irrational, but 355/113 is real.

```
240 FIF ABS(J*L)*ABS(%X-J/L)/%X >0.0033 PRINT
"IRRATIONAL"; UNTIL 0
250 PRINT J"/"L;UNTIL 0
```

Variables:

I - Integer part of %X  
J - Numerator of fractional representation  
L - Denominator of fractional representation  
\$P - Proper "P" or improper "I" flag  
%R - Real part of %X  
\$\$ - Sign of %X  
%X - Absolute value of rational

## POLYNOMIAL

Polynomials are important in several branches of mathematics because many continuous functions are represented by polynomials. The familiar quadratic equation is a polynomial of degree 2, and its general form is:

$$ax^2 + bx + c$$

where '^' means "raised to the power".

The present program is a general-purpose polynomial manipulator. It will simplify an equation to a polynomial of integer coefficients, and can add, subtract, multiply, and divide polynomials. The

program first asks for the degree of the polynomial; that is, the highest power of X represented. For example:

```
DEGREE?4  
SIMPLIFY:(X+2)*(3-X)*(X+1)^2
```

will give:

$$-X^4 - X^3 + 7X^2 + 13X + 6$$

and:

```
DEGREE?3  
SIMPLIFY:(X^3+1)/(X+1)
```

will give:

$$X^2 - X + 1$$

### Program Description

The program works by substituting different values of X in the equation to be simplified, to obtain a series of values of Y. For example, the first equation above produces:

X:	0	1	2	3	4
Y:	6	24	36	0	-150

A 2-stage procedure is then used to obtain the polynomial coefficients from these values. This procedure involves taking differences between successive members of the series, and these differences are divided by the number of the row:

X:	0	1	2	3	4
Y:	6	24	36	0	-150
		18	12	-36	-150
C=1		18	12	-36	-150
		-6	-48	-114	
C=2		-3	-24	-57	
		-21	-33		
C=3		-7	-11		
			-4		
C=4			-1		

The next stage involves taking the series of numbers on the left-hand side of this triangle of differences, namely:

6, 18, -3, -7, -1.

These numbers are then transformed into the coefficients of the polynomial by the following procedure:

a. Start with N one less than the degree (3 in this case).



b. Subtract N times the last number from its predecessor.

c. Subtract (N-1) times the last two numbers from their predecessors.

c. Repeat step b until N=0.

In the present example the series of coefficients produced is:

6, 13, 7, -1, -1.

### BBC Computer Version

The BBC Computer BASIC allows negative numbers to be raised to integer powers; therefore the '^' operator can be used in the expression to be simplified, as in:  $(X-1)^6$ .

```
10 INPUT "DEGREE?"N
20 INPUT "SIMPLIFY:" X$
30 DIM CX(N):@%=1
```

Evaluate equation for successive values of X.

```
40 FOR X=0 TO N:PROCE:NEXT
```

Keep taking differences.

```
50 FOR C=1 TO N
60 FOR J=N TO C STEP-1
70 CX(J)=(CX(J)-CX(J-1)) DIV C
80 NEXT J:NEXT C
```

Transform differences into coefficients of polynomial.

```
90 FOR C=N-1 TO 0 STEP-1
100 FOR J=C TO N-1
110 CX(J)=CX(J)-CX(J+1)*C
120 NEXT J:NEXT C
```

Print non-zero terms of polynomial, with a leading "-" if first non-zero term is negative.

```
125 S=0
130 FOR C=N TO 0 STEP-1:IF CX(C)=0 GOTO 160
135 IF CX(C)<0 PRINT " - ";ELSE IF S PRINT " + ";
140 IF ABS(CX(C))>1 OR C=0 PRINT ABS(CX(C));
145 IF C>1 PRINT "X^";C;
148 IF C=1 PRINT "X";
150 S=1
160 NEXT C:PRINT '
```

PROCE - Evaluate equation here.

```
200 DEF PROCE CX(X)=EVAL(X$): ENDPROC
```

Variables:

C - Coefficient number

CX(0)..CX(N) - Coefficients of  $X^N$  in polynomial

J - Counter

N - Maximum degree of polynomial

S - Flag for printing "+" sign

T - Pointer to equation string

X,Y - Unknowns in equation

### Atom Version

In the Atom version, powers of numbers and polynomials should be represented by repeated multiplication; thus, instead of ' $(X+1)^2$ ' write ' $(X+1)*(X+1)$ '.

```
10 INPUT "DEGREE"N
```

Insert equation into line 200 of program.

```
15 T=TOP;DO T=T-1; UNTIL ?T=CH"e"
```

```
20 T=T+3;INPUT "SIMPLIFY"$T
```

```
25 T=T+LENT;$T=";R.";T?4=#FF
```

```
30 DIM T(64),XX(N);@=0
```

Evaluate equation for successive values of X.

```
40 FOR X=0 TO N;GOSUB e;XX(X)=Y;NEXT
```

Keep taking differences.

```
50 FOR C=1 TO N
```

```
60 FOR J=N TO C STEP -1
```

```
70 XX(J)=(XX(J)-XX(J-1))/C
```

```
80 NEXT J;NEXT C
```

Transform differences into coefficients of polynomial.

```
90 FOR C=N-1 TO 0 STEP -1
```

```
100 FOR J=C TO N-1
```

```
110 XX(J)=XX(J)-XX(J+1)*C
```

```
120 NEXT J;NEXT C
```

Print non-zero terms of polynomial, with a leading  
"-" if first non-zero term is negative.

```
125 S=0
130 FOR C=N TO 0 STEP -1;IF XX(C)=0 GOTO z
135 IF XX(C)<0 PRINT " - "; GOTO s
137 IF S PRINT " + "
140sIF ABS XX(C)>1 OR C=0 PRINT ABS XX(C)
145 IF C>1 PRINT "X^" C
148 IF C=1 PRINT "X"
150 S=1
155zNEXT C;PRINT ''
170 END
```

Poke equation to be evaluated into program here.

```
200eY=X;RETURN
```

Variables:

C - Coefficient number  
J - Counter  
N - Maximum degree of polynomial  
S - Flag for printing "+" sign  
T - Pointer to equation string  
XX(0)..XX(N) - Coefficients of  $X^N$  in polynomial  
X,Y - Unknowns in equation

### Further Suggestions

With slight modification the program can be used to generate a polynomial of specified degree from a set of data points. For the BBC version, alter line 200 to:

```
200 DEF PROC PRINT "F("X");:INPUTY:ENDPROC
```

For the Atom version delete lines 15, 20, and 25, and alter line 200 to:

```
200ePRINT "F("X");:INPUTY;RETURN
```

The program will then prompt for the coefficients:

```
DEGREE?4
F(0)?6
F(1)?24
F(2)?36
F(3)?0
F(4)?-150
```

This will print:

```
- X^4 - X^3 + 7X^2 + 13X + 6
```

The program could also be modified to handle polynomials with real coefficients, and the Fractions program could be used to give the coefficients as fractions where possible.

CALCULATOR

The following program acts as a calculator, with the ability to add, multiply, and divide. The unusual feature provided by the program is that it will calculate to an unlimited accuracy, restricted only by the amount of memory available to the program.

The program uses reverse Polish notation, also known as Polish suffix notation, in which the operator follows the operand or operands. This notation is used on some scientific calculators because it allows any expression to be entered without the need for brackets.

An expression in reverse Polish notation is:

1 2 + 4 5 + \*

To understand how this is evaluated, imagine the expression read from left to right. When an operator, such as '+' and '\*', is encountered it removes the two numbers to its left, performs the operation, and replaces them with the result. Successive stages in the evaluation of this equation are:

```

1 2 + 4 5 + *
   3 4 5 + *
     3     9 *
                   27
  
```

The equivalent expression in algebraic notation is  $(1+2)*(4+5)$ . The great advantage of this notation is that the order of evaluation is unambiguous, and no brackets are needed to indicate how the expression is to be evaluated. Some other equations in algebraic form are shown below together with their equivalents in reverse Polish:

Algebraic:	Reverse Polish:
2 + 3 + 4	2 3 + 4 +
	or 2 3 4 + +
3 * (4 + 5)	3 4 5 + *
(3 * 4) + 5	3 4 * 5 +

The above example would be entered into the calculator program as shown below. The '?' is the prompt, and after each number or operator is entered RETURN should be typed. The program prints out the result after each operation:

```
?1
?2
?+
= 3
?4
?5
?+
= 9
?*
= 27
```

### Sample run

In the following example using larger numbers the calculator works out 10000000001/99009901:

```
?10000000001
?99009901
?/
= 101
```

A number can be squared, by duplicating it on the stack with " and then multiplying:

```
?11111111111111
?"
= 11111111111111
?*
= 12345679012320987654321
```

Finally, we divide the result by the square of 111111:

```
?1111111
?"
= 1111111
?*
= 12345654321
?/
= 1000002000001
```

These results are all exact, and could not of course be obtained with a conventional calculator.

### Program Operation

In these programs the long numbers are represented as strings of ASCII characters. The arithmetic routines

to multiply, divide, and add, take two strings and generate a result in the form of a third string. This representation was chosen so that the standard BASIC string operations could be used to manipulate long numbers and print them out; other representations would probably require special routines for these functions, but might give faster calculation.

### BBC Computer Version

```
5 REM ... Calculator ...
10 DIM M%240,SS%(10),F%1023
20 S%=0:SS%(S%)=F%:Z%=ASC("0")
30 REPEAT INPUT LINE$M%
```

Look for digit, or one of the following commands:  
" - duplicate item on stack  
\* - multiply  
+ - add  
/ - divide

```
40 IF?M%>=ASC("0")AND?M%<=ASC("9")$F%=$M%:
PROCUP:PROCF:UNTILO
45 IFS%>0AND?M%=ASC("A")A%=SS%(S%):
D%=SS%(S%-1):PROCUP:PROCRESULT:UNTILO
```

Make sure there are at least 2 items on the stack, and then set up:  
D - top of stack (for result).  
B - first operand on stack  
A - second operand on stack

```
46 IFS%<2 PRINT"STACK EMPTY":UNTIL 0
48 D%=SS%(S%):B%=SS%(S%-1):A%=SS%(S%-2)
50 IF?M%=ASC("**")PROC MUL:PROC DONE:UNTILO
55 IF?M%=ASC("+")PROC ADD:PROC DONE:UNTILO
58 IF?M%=ASC("/")PROC DIV:PROC DONE:UNTILO
60 PRINT"ERROR":UNTILO
```

PROC DONE - Decrement stack, then drop through to put result on stack.

```
90 DEF PROC DONE S%=S%-1
```

PROC RESULT - Result is in D. Remove leading zeros; then copy result down stack, and print result.

```
92 DEF PROC RESULT
93 D%=D%-1:REPEAT D%=D%+1:UNTIL?D%<>Z% ORD%?1=13
95 $A%=$D%:F%=A%:PROCF
100 PRINT " = "$SS%(S%-1)
110 ENDPROC
```

PROCF - Make room for result on top of stack.

```
500 DEF PROC F%=F%+LEN($F%)+1:SS$(S%)=F%:ENDPROC
```

PROC MUL - Multiply:  $SD\% = SA\% * SB\%$   
First set  $L\%$  to length of result, and zero  $SD\%$ .  
Then do long multiplication.

```
1000 DEF PROC MUL
1005 L%=LEN($A%)+LEN($B%):FOR N%=0 TO L%-1:
D%?N%=Z%:NEXT:D%?L%=13
1010 FOR J%=LEN($B%)-1 TO 0 STEP -1: C%=0:G%=B%?J%-Z%
1020 V%=D%+J%+1
1030 FOR L%=LEN($A%)-1 TO 0 STEP -1: H%=A%?L%-Z%
1040 Q%=G%*H%+C%+(V%?L%-Z%)
1050 V%?L%=Q% MOD 10+Z%:C%=Q%/10:NEXT
1060 V%?L%=C%+Z%
1070 NEXT:ENDPROC
```

PROC ADD - Addition:  $SD\% = SA\% + SB\%$   
Set result to longest operand, and add in other operand.

```
2000 DEF PROC ADD
2005 W%=A%:V%=B%:J%=LEN($A%)-LEN($B%): IF J%<0
W%=B%:V%=A%:J%=-J%
2010 $(D%+1)=$W%:D%=Z%:C%=0:W%=D%+J%+1
2020 FOR L%=LEN($V%)-1 TO 0 STEP -1
2030 Q%=W%?L%+V%?L%-2*Z%
2040 W%?L%=Q% MOD 10+Z%:C%=Q%/10
2050 NEXT:W%?L%=W%?L%+C%:ENDPROC
```

PROC DIV - Division:  $SD\% = SA\% / SB\%$   
Keep subtracting divisor, counting in  $V\%$ , using  $C\%$   
as a borrow this time, until overflows ( $C\%=0$ );  
then add divisor back in once.

```
4000 DEF PROC DIV
4005 FOR J%=0 TO LEN($A%)-LEN($B%):W%=A%+J%:V%=-1
4010 REPEAT V%=V%+1:C%=1
4020 FOR L%=LEN($B%)-1 TO -J% STEP -1
4025 Q%=Z%:IF L%>=0 Q%=B%?L%
4030 Q%=W%?L%-Q%+C%+9
4032 W%?L%=Q% MOD 10+Z%:C%=Q%/10
4035 NEXT
4040 UNTIL C%=0
4050 FOR L%=LEN($B%)-1 TO -J% STEP -1
4055 Q%=Z%:IF L%>=0 Q%=B%?L%
4060 Q%=W%?L%+Q%-2*Z%+C%
4065 W%?L%=Q% MOD 10+Z%:C%=Q%/10:NEXT
4070 D%?J%=V%+Z%:NEXT:D%?J%=13
```

## PROCUP - Increment stack

```
6000 DEF PROCUP IFS%>10PRINT"STACK FULL":ENDPROC
6010 S%=S%+1:ENDPROC
```

## Variables:

\$A%, \$B% - Strings containing the two operands used by the arithmetic routines  
 C% - Carry/borrow  
 \$D% - String into which result is put  
 H% - Temporary variable  
 J%, L% - Loop counters  
 M% - Input line  
 Q% - Intermediate result in calculations  
 S% - Next free stack pointer  
 SS%(0)..SS%(10) - Pointers to number strings on stack  
 V%, W% - Pointers used by arithmetic routines  
 Z% - Equal to ASC("0")

## Atom Version

```
5 REM ... CALCULATOR ...
10 DIM M(64),SS(10),F(-1)
20 S=0;SSS=F;Z=CH"0"
30 DO INPUT$M
```

Look for digit, or one of the following commands:  
 " - duplicate item on stack  
 \* - multiply  
 + - add  
 / - divide

```
40 IF?M>=CH"0"AND?M<=CH"9"$F=$M;GOSUB u;GOSUB
f;GOTO x
45 IFS>0AND?M=CH""A=SSS;D=SS(S-1);GOSUB u;GOTO w
```

Make sure there are at least 2 items on the stack, and then set up:  
 D - top of stack (for result).  
 B - first operand on stack  
 A - second operand on stack

```
46 IFS<2 PRINT"STACK EMPTY";GOTO x
48 D=SSS;B=SS(S-1);A=SS(S-2)
50 IF?M=CH"*"GOSUB m;GOTO z
55 IF?M=CH"+"GOSUB a;GOTO z
58 IF?M=CH"/"GOSUB d;GOTO z
60 PRINT"ERROR";GOTO x
90zS=S-1
```



w - Result is in D. Remove leading zeros, then copy result down stack, and print result.

```
92wD=D-1;DOD=D+1;UNTIL?D<>Z ORD?1=13
95 $A=$D;F=A;GOSUB f
100 PRINT " = "$SS(S-1)'
110xUNTIL0
```

f - Make room for result on top of stack.

```
500fF=F+LENF+1;SSS=F;RETURN
```

m - Multiply:  $\$D = \$A * \$B$   
First set L to length of result, and zero \$D. Then do long multiplication.

```
1000mL=LENA+LENB;FOR N=0TO L-1; D?N=Z;NEXT;D?L=13
1010 FOR J=LENB-1 TO 0 STEP -1;C=0;G=B?J-Z
1020 V=D+J+1
1030 FOR L=LENA-1 TO 0 STEP -1;H=A?L-Z
1040 Q=G*H+C+(V?L-Z)
1050 V?L=Q%10+Z;C=Q/10;NEXT
1060 V?L=C+Z
1070 NEXT;RETURN
```

a - Addition:  $\$D = \$A + \$B$   
Set result to longest operand, and add in other operand.

```
2000aW=A;V=B;J=LENA-LENB;IFJ<0 W=B;V=A;J=-J
2010 $(D+1)=$W;?D=Z;C=0;W=D+J+1
2020 FORL=LENV-1 TO 0 STEP -1
2030 Q=W?L+V?L-2*Z
2040 W?L=Q%10+Z;C=Q/10
2050 NEXT;W?L=W?L+C;RETURN
```

d - Division:  $\$D = \$A / \$B$   
Keep subtracting divisor, counting in V, using C as a borrow this time, until overflows (C=0); then add divisor back in once.

```
4000dFORJ=0 TO LENA-LENB;W=A+J;V=-1
4010 DO V=V+1;C=1
4020 FOR L=LENB-1 TO -J STEP -1
4025 Q=Z;IF L>=0 Q=B?L
4030 Q=W?L-Q+C+9
4032 W?L=Q%10+Z;C=Q/10
4035 NEXT
4040 UNTIL C=0
4050 FORL=LENB-1 TO -J STEP -1
4055 Q=Z;IF L>=0 Q=B?L
```

```
4060 Q=W?L+Q-2*Z+C
4065 W?L=Q%10+Z;C=Q/10;NEXT
4070 D?J=V+Z;NEXT;D?J=13
4080 RETURN
```

u - Increment stack

```
6000uIFS>10PRINT"STACK FULL";RETURN
6010 S=S+1;RETURN
```

#### Variables:

\$A,\$B - Strings containing the two operands used by the arithmetic routines  
C - Carry/borrow  
\$D - String into which result is put  
H - Temporary variable  
J,L - Loop counters  
M - Input line  
Q - Intermediate result in calculations  
S - Next free stack pointer  
SS(0)..SS(10) - Pointers to number strings on stack  
V,W - Pointers used by arithmetic routines  
Z - Equal to CH"0"

#### Further Suggestions

The calculator could be extended to handle negative numbers, represented by a string starting with a "-" sign, and subtraction could then be implemented. As a more ambitious undertaking, the calculator could be extended to give arbitrary-precision versions of all the integer functions of a standard pocket calculator, including  $X^Y$ , factorials, and probability functions.

A more ambitious extension would make these routines the basis of an interpreter, which would allow programs to be written manipulating numbers to unlimited accuracy.

# 5 Compiler

The final chapter in this book is devoted to an ambitious project to write a compiler which will convert programs written in a high-level language, similar to Pascal, into the machine code of the Atom and BBC Computer.

In order to keep the compiler as straightforward as possible, and to enable it to run in the memory of the standard Atom or BBC Computer, several simplifications were made. First of all, the compiler is limited to 8-bit numbers; i.e. numbers in the range 0 to 255. Secondly, it handles a restricted set of statements and operators.

The program was primarily developed to illustrate the problems involved in designing a high-level language compiler. It should also serve as a good introduction to recursively-defined languages such as Pascal, and shows the relationship between a statement in such a language and the corresponding machine-code. Finally, the compiler will compile into efficient machine code, and so could be used to develop machine-code programs for applications such as machine control.

## Compilers and Interpreters

The BASIC running on the Atom and BBC computers is an 'interpreter'; that is, it executes each statement in the program as it encounters it. A 'compiler', on the other hand, takes a program in one language and converts it into machine code - the language of the processor. The compiled, machine-code version of the program can then be run without further needing the presence of the compiler program. Also, since they are running in the language of the computer itself, compiled programs are likely to run very much faster than interpreted versions of the same programs.

The compiler to be described is written in BASIC, and makes use of the mnemonic assembler built into the Atom and the BBC Computer. The advantage of using the

built-in assembler is that an assembler listing is automatically produced, so the code generated for a particular high-level language program is comprehensible and easily checked. However, the compiler could be altered to run on other computers by generating machine code directly; it could also be used to write programs for microprocessors other than the 6502.

## ASSIGNMENT

As a preliminary step in designing the compiler, a program is presented that will take a series of assignment statements such as  $A=A+2$  and assemble them into 6502 machine code. This program, Assignment, will then form the basis for the complete compiler.

This program compiles in a single pass, reading the input line and generating assembler statements as it goes. The following operations, which work on 8-bit numbers, are handled by the program:

+	: add	-	: subtract
&	: logical AND		: logical OR
>>	: right shift	<<	: left shift

All these operators have the same priority, and brackets can be used to alter the order of evaluation. Note that the shift operators must have a constant as their right-hand operand, as in:

```
A=B>>2
```

Variable names can have up to six upper-case letters, and are automatically assigned to zero-page memory locations by the program.

The program uses a stack to hold intermediate results during compilation. Addresses are represented by numbers in the range 1 to FFFF (hexadecimal), and constants as the negative of their value; i.e. by numbers in the range 0 to -255. When the compiler reads an operator it performs the following sequence:

- pull the location of the previous results from the stack,
- assemble code to calculate the new result,
- push onto the stack the location of the new result.

The intermediate results during the compilation of a

complicated expression, such as  $A=(B+2)\&(C+3)$ , are associated with temporary locations, TT(0), TT(1) etc.

Using this procedure the program would compile the statement:

```
MAX=31&(VAR+15)
```

as:

```
LDA VAR
CLC
ADC @15
STA TT(0)
LDA @31
AND TT(0)
STA TT(1)
LDA TT(1)
STA MAX
```

The superfluous STA TT(1) and LDA TT(1) are eliminated by keeping track of the address whose contents are in the accumulator at any time. Instead of generating code to load the accumulator, a call is made to a subroutine that first checks to make sure that the accumulator does not already contain the required value. Furthermore, code is only generated to store the accumulator's contents when absolutely necessary; that is, when a new value is to be loaded into the accumulator.

### Sample Run

The following section shows a sample run of the Assignment program, on the Atom, for various statements which are typed in after the '?' prompt. In the listing the first column, which is only present in the Atom version, shows the line in the compiler program that generated the assembler code. The next column gives the address of the machine code, followed by the one, two, or three bytes of the instruction, and the assembler statement. The symbols L, U, and H in the assembler statements are used by the program, and take different values at different points in the program.

```
>RUN
?VAR=1
```

```
7210 3A00 A9 01    LDA @-L
7200 3A02 85 51    STA H
```

The program has allocated location #51 for the variable VAR.

```
?VAR=VAR+1
```

```

7040 3A04 A5 51   LDA L
3070 3A06 18     CLC
3070 3A07 69 01   ADC @-U
7200 3A09 85 51   STA H

```

?MAX=31&(VAR+15)

```

7040 3A0B A5 51   LDA L
3070 3A0D 18     CLC
3070 3A0E 69 0F   ADC @-U
7200 3A10 85 80   STA H
7210 3A12 A9 1F   LDA @-L
3065 3A14 25 80   AND U
7200 3A16 85 52   STA H

```

The program has allocated location #52 for MAX. The bracketed expression is evaluated first, and stored in a temporary location #80.

?MIN=(1|VAR)-(MAX>>4)

```

7210 3A18 A9 01   LDA @-L
3060 3A1A 05 51   ORA U
7200 3A1C 85 80   STA H
7040 3A1E A5 52   LDA L
3180 3A20 4A     LSR A
3180 3A21 4A     LSR A
3180 3A22 4A     LSR A
3180 3A23 4A     LSR A
7200 3A24 85 81   STA H
7040 3A26 A5 80   LDA L
3055 3A28 38     SEC
3055 3A29 E5 81   SBC U
7200 3A2B 85 53   STA H

```

Here MIN is allocated location #53, and two temporary locations are used. Note that the right-hand operand of '>>' or '<<' must be a constant.

```

?X=1+(VAR-)
BRACKET MISSING
X=1+(VAR-)

```

Finally, a statement which generated an error.

### BBC Computer Version

```

5 REM ... Assignment ...
10 HIMEM=&2800
20 DIM SS(20),ID$(30),JJ(30)
30 DIM TT(20),X&7,Z&256:A$=CHR$(6)

```

Important addresses:

MC - Start address for machine code.

VARS - Start address for variables and arrays.

TEMPS - 20 temporary locations.

```
35 MC=&3800:VARS=&50:TEMPS=&80:PRARG=&94:SADD=&2800
115 FOR N=0TO20:TT(N)=0:NEXT
140 I=0:P=MC:S=0:R=0:H=0:T=0
```

One pass of compilation. Initialise pointers, and make sure accumulator is stored finally.

```
200 REPEAT INPUT$Z%:A=Z%
210 PROCSTMT:PROCSTA
220 UNTIL FALSE
```

PROCSTMT - Statement. Skip blanks, then read symbol.

```
1000 DEF PROCSTMT
1010 PROCSP:PROCSYM
1110 PROCSP
1135 PROCV
1160 PROCRHS:H=V:PROCSTA:ENDPROC
```

PROCRHS - Right-hand side of assignment statement.

```
1180 DEF PROCRHS
1185 IF ?A<>ASC"=" PRINTA$"NO =" :PROCERR
1190 A=A+1:PROCEXP:L=FNPUL
1195 PROCLDA:V=FNPUL:ENDPROC
```

PROCIDENT - Read an identifier.

```
2000 DEF PROCIDENT
2010 PROCSYM:PROCV:ENDPROC
```

PROCV - Look up identifier \$X% in symbol table. If symbol does not already exist (N=I) allocate address for it. Push address to stack.

```
2020 DEF PROCV
2030 IF N=0 ENDPROC
2040 PROCLOOK
2050 IFN=I:I=I+1:R=R+1:JJ(N)=R+VARS
2070 IFI>30PRINT"TOO MANY VARIABLES":PROCERR
2080 U=JJ(N):PROCPSH(U):ENDPROC
```

PROCONST - Read a decimal constant. If not found, N=0. If found, push minus its value.

```

2100 DEF PROCONST
2105 PROCSP
2110 N=-1:C=0:REPEAT D=C:N=N+1:C=C*10
2120 C=C+A?N-ASC"0"
2130 UNTILA?N<ASC"0"ORA?N>ASC"9"
2140 IFN=0 ENDPROC
2150 A=A+N
2160 U=-D:PROCP SH(U):N=1:ENDPROC

```

PROCLOOK - Look up \$X% in symbol table, ID\$(0), ID\$(1) ... If not found, N=I.

```

2400 DEF PROCLOOK
2410 ID$(I)=$X%:N=-1
2420 REPEAT N=N+1:UNTILID$(N)=ID$(I):ENDPROC

```

PROCEXP - Assemble code to calculate an expression, of the form:

<factor> <operator> <factor>

where <operator> is one of:

+	: add	-	: subtract
	: OR	&	: AND
<<	: left shift	>>	: right shift

Then push the address of the result on the stack.

```

3000 DEF PROCEXP PROCSP:PROCFACOR
3010 PROCSP
3020 IF?A=ASC"+"OR?A=ASC"- "OR?A=ASC"&"
OR?A=ASC"|"O=?A:A=A+1:GOTO 3035
3025 IF NOT((?A=ASC">"AND A?1=ASC">")OR (?A=ASC"<"AND
A?1=ASC"<"))ENDPROC
3030 O=?A:A=A+2
3035 PROCP SH(O)
3040 PROCFACOR:U=FNPUL:O=FNPUL:L=FNPUL:PROCLDA
3045 IF U<=0 GOTO 3070
3050 IFO=ASC"+"[CLC:ADC U:]
3055 IFO=ASC"- "[SEC:SBC U:]
3060 IFO=ASC"|" [ORA U:]
3065 IFO=ASC"&" [AND U:]
3068 GOTO 3190
3070 IFO=ASC"+"[CLC:ADC #-U:]
3075 IFO=ASC"- "[SEC:SBC #-U:]
3080 IFO=ASC"|" [ORA #-U:]
3085 IFO=ASC"&" [AND #-U:]
3160 IFO=ASC"<"FOR N=1TO-U:[ASL A:]:NEXT
3180 IFO=ASC">"FOR N=1TO-U:[LSR A:]:NEXT
3190 L=U:PROCRELEASE(L):PROCTEMP
3195 GOTO 3010

```



PROCFACOR - Factor. Check for symbol, constant, or bracketed expression. If the symbol is followed by '(' or '[' then it is a function or an array respectively.

```
3600 DEF PROCFACOR
3610 PROCSYM:IFN=0GOTO3630
3620 PROCV
3625 ENDPROC
3630 PROCONST:IF N ENDPROC
3635 IF?A<>ASC "(" PRINTA$"BRACKET MISSING":PROCERR
3640 A=A+1:PROCEXP:PROCSP
3650 IF?A<>ASC ")" PRINTA$"BRACKET MISSING":PROCERR
3660 A=A+1:ENDPROC
```

PROCPSH - Push argument onto stack.

```
5020 DEF PROCPSH(U) SS(S)=U:S=S+1:IFS<21 ENDPROC
5021 PRINTA$"STACK FULL":PROCERR
```

FNPUL - Pull from stack.

```
5030 DEF FNPUL:S=S-1:IFS>=0 =SS(S)
5031 PRINTA$"STACK ERROR":PROCERR
```

PROCSP - Skip blanks.

```
5040 DEF PROCSP
5042 IF?A=32 REPEAT A=A+1:UNTIL?A<>32
5049 ENDPROC
```

PROCTEMP - Generate a temporary location TT(N); return its address in T, set H to the address, and push the address.

```
5100 DEF PROCTEMP
5110 N=-1:REPEAT N=N+1: IF N>20PRINTA$"NOT ENOUGH
TEMP":PROCERR
5120 UNTILTT(N)=0
5130 T=N+TEMPS:TT(N)=T:U=T:H=T:PROCPSH(U):ENDPROC
```

PROCSYM - Read a symbol into \$X% from \$A. Returns N=0 if no symbol found.

```
6000 DEF PROCSYM
6010 PROCSP:N=-1:REPEAT N=N+1: N?X%=A?N
6020 UNTILA?N>ASC "Z"ORA?N<ASC "A"ORN=7
6030 IF N=0 ENDPROC
6040 IF N<7 N?X%=&D:A=A+N:ENDPROC
6050 PRINTA$"SYMBOL TOO LONG":PROCERR
```

PROCLDA - Assemble code to load the accumulator with L. If accumulator already contains L (L=H) then do nothing; otherwise store its previous contents (PROCSTA) and load new contents.

```
7000 DEF PROCLDA
7010 IFL=H AND L>0 PROCRELEASE(L);ENDPROC
7020 PROCSTA
7030 IFL<=0 [LDA #-L:];ENDPROC
7040 [LDA L:];PROCRELEASE(L)
7050 ENDPROC
```

PROCSTA - Assemble code to store accumulator's contents to location H.

```
7100 DEF PROCSTA
7200 IFH>0[STA H:]:H=0
7210 ENDPROC
```

PROCRELEASE - Release temporary variable for re-use.

```
7300 DEF PROCRELEASE(L)
7310 IF L>=TEMPS AND L<TEMPS+20:TT(L-TEMPS)=0
7320 ENDPROC
```

PROCERR - Output error.

```
9000 DEF PROCERR
9010 PRINT '$Z%'
9030 PRINT TAB(A-Z%-1);"^-";END
```

#### Variables:

A - Pointer to current position in expression being compiled

C - Used to evaluate constant

H - Address whose contents are currently in accumulator. H=0 means ignore previous contents

I - Number of next free symbol

ID\$(0)..ID\$(30) - Pointers to symbol names

JJ(0)..JJ(30) - Addresses of symbols

L - Value or address to be loaded into accumulator; used by PROCLDA

N - Temporary variable

O - Operator read by PROCEXP

P - Program location counter, used by assembler

RR(0)..RR(2) - Constant addresses

R - Number of variable locations used up

S - Next free location on SS stack

SS(0)..SS(20) - Stack used by compiler

T - Temporary location assigned by PROCTEMP

TT(0)..TT(20) - Flags for temporary locations; value=0  
if location is free for use  
U - Value to be pushed by PROCPSH  
V - Used by FNPUL  
X% - String into which symbols and keywords are read  
by PROCSYM  
Z% - Input buffer

### Atom Version

```
10 ... ASSIGNMENT ...
20 DIM SS(20),LL(20),II(30),JJ(30)
30 DIM X(7),TT(20),RR(2)
```

Important addresses:

RR0 - Start address of machine code.  
RR1 - Start address of variables and arrays.  
RR2 - 20 temporary locations.

```
35 RR0=#3A00;RR1=#50;RR2=#80
40 F.N=0TO30;DIMI(6);IIN=I;JJN=RR0;N.
115 F.N=0TO20;TTN=0;LLN=RR0;N.
140 Z=#100;G=0;I=0;P=RR0;S=0;R=0;H=0;T=0
```

One pass of compilation. Initialise pointers, and  
make sure accumulator is stored finally.

```
200 DO INPUT$Z;A=Z
210 GOS.s;GOS.m
220 UNTIL 0
```

s - Statement. Skip blanks, then read symbol.

```
1000sREM STATEMENT
1010 GOS.b;GOS.x
1110 GOS.b
1135 GOS.j
1160 GOS.d;H=V;GOS.m;R.
```

d - Right-hand side of assignment statement.

```
1180dIF?A<>CH"=" P.$6"NO =" ;G.o
1190 A=A+1;GOS.e;GOS.v;L=V;GOS.l;GOS.v;R.
```

i - Read an identifier.

```
2000iREM IDENTIFIER
2010 GOS.x
```

j - Look up identifier \$X in symbol table. If symbol does not already exist (N=I) allocate address for it. Push address to stack.

```
2030jIF N=0 R.
2040 GOS.y
2050 IFN=I;I=I+1;R=R+1;JJN=R+RR1
2070 IFI>30P.$6"TOO MANY VARIABLES";G.o
2080 U=JJN;GOS.u;R.
```

c - Read a decimal constant. If not found, N=0. If found, push minus its value.

```
2100cREM CONSTANT
2105 GOS.b
2110 N=-1;C=0;DO D=C;N=N+1;C=C*10
2120 C=C+A?N<CH"0"
2130 U.A?N<CH"0"ORA?N>CH"9"
2140 IFN=0 R.
2150 A=A+N
2160 U=-D;GOS.u;N=1;R.
```

y - Look up \$X in symbol table, \$II(0), \$II(1) ...  
If not found, N=I.

```
2400yREM LOOKUP
2410 $III=$X;N=-1
2420 DO N=N+1;U.$IIN=$III;R.
```

e - Assemble code to calculate an expression, of the form:

<factor> <operator> <factor>

where <operator> is one of:

+ : add                   - : subtract

| : OR                     & : AND

<< : left shift   >> : right shift

Then push the address of the result on the stack.

```
3000eREM EXPRESSION
3010 GOS.b;GOS.f
3015 GOS.b
3020 IF?A=CH"+"OR?A=CH"-OR?A=CH"&"OR
?A=CH"|O=?A;A=A+1;G.3035
3025 IF((?A=CH">"A.A?1=CH">))OR
(?A=CH"<"A.A?1=CH"<)):1 R.
3030 O=?A;A=A+2
3035 U=O;GOS.u
3040 GOS.f;GOS.v;U=V;GOS.v;O=V;GOS.v;L=V;GOS.l
3045 IF U<=0 G.3070
3050 IFO=CH"+"[CLC;ADC U;]
3055 IFO=CH"-[SEC;SBC U;]
```

```

3060 IFO=CH"|" [ORA U;]
3065 IFO=CH"&" [AND U;]
3068 G.3190
3070 IFO=CH"+" [CLC;ADC @-U;]
3075 IFO=CH"-" [SEC;SBC @-U;]
3080 IFO=CH"|" [ORA @-U;]
3085 IFO=CH"&" [AND @-U;]
3160 IFO=CH"<" F.N=1TO-U; [ASL A;];N.
3180 IFO=CH">" F.N=1TO-U; [LSR A;];N.
3190 L=U;GOS.r;GOS.t
3195 G.3015

```

f - Factor. Check for symbol, constant, or bracketed expression. If the symbol is followed by '(' or '[' then it is a function or an array respectively.

```

3600fREM FACTOR
3610 GOS.x;IFN=0G.3630
3620 GOS.j
3625 R.
3630 GOS.c;IF N R.
3635 IF?A<>CH(" P.$6"BRACKET MISSING";G.o
3640 A=A+1;GOS.e;GOS.b
3650 IF?A<>CH")" P.$6"BRACKET MISSING";G.o
3660 A=A+1;R.

```

u - Push U onto stack.

```

5020uSSS=U;S=S+1;IFS<21 R.
5021 P.$6"STACK FULL";G.o

```

v - Pull V from stack.

```

5030vS=S-1;IFS>=0V=SSS; R.
5031 P.$6"STACK ERROR";G.o

```

b - Skip blanks.

```

5040bIF?A=32 DO A=A+1;U.?A<>32
5043 R.

```

t - Generate a temporary location TTN; return its address in T, set H to the address, and push the address.

```

5100tREM TEMP. LOC.
5110 N=-1;DO N=N+1; IF N>20P.$6"NOT ENOUGH TEMP";G.o
5120 U.TTN=0
5130 T=N+RR2; TTN=T; U=T; H=T; G.u

```

x - Read a symbol into \$X from \$A. Returns N=0 if no symbol found.

```
6000xREM READ SYMBOL
6010 GOS.b;N=-1;DO N=N+1; N?X=A?N
6020 U.A?N>CH"Z"ORA?N<CH"A"ORN=7
6030 IF N=0 R.
6040 IF N<7 N?X=#D;A=A+N;R.
6050 P.$6"SYMBOL TOO LONG";G.o
```

l - Assemble code to load the accumulator with L. If accumulator already contains L (L=H) then do nothing; otherwise store its previous contents (GOS.m) and load new contents.

```
7000lREM LOAD ACCUMULATOR
7010 IFL=H AND L>0 G.r
7020 GOS.m
7030 IFL<=0 [LDA @-L;];R.
7040 [LDA L;];G.r
```

m - Assemble code to store accumulator's contents to location H.

```
7100mREM STORE ACCUMULATOR
7200 IFH>0[STA H;];H=0
7210 R.
```

r - Release temporary variable with address L for re-use.

```
7300rREM RELEASE VARIABLE
7310 IF L>=RR2 AND L<RR3;TT(L-RR2)=0
7320 R.
```

o - Output error.

```
9000oREM ERROR
9020 P.'$Z'
9030 F.N=Z TOA-2;P." ";N.;P."^";E.
```

Variables:

A - Pointer to current position in expression being compiled

C - Used to evaluate constant

H - Address whose contents are currently in accumulator. H=0 means ignore previous contents

I - Number of next free symbol

II(0)..II(30) - Pointers to symbol names

JJ(0)..JJ(30) - Addresses of symbols

L - Value or address to be loaded into accumulator;

used by subroutine l  
N - Temporary variable  
O - Operator read by subroutine e  
P - Program location counter, used by assembler  
RR(0)..RR(2) - Constant addresses  
R - Number of variable locations used up  
S - Next free location on SS stack  
SS(0)..SS(20) - Stack used by compiler  
T - Temporary location assigned by subroutine t  
TT(0)..TT(20) - Flags for temporary locations; value=0  
if location is free for use  
U - Value to be pushed by subroutine u  
V - Value to be pulled by subroutine v  
X - String into which symbols and keywords are read by  
subroutine x  
Z - Input buffer

SPL

The next step in developing the complete compiler is to define the language that it will compile. The language designed for this purpose is called SPL - Simple Programming Language. In some respects it is a subset of the popular languages Pascal and Algol, but with the restriction that numbers are limited to the range 0 to 255, and that the language includes only the essential types of statement.

### Syntax of SPL

Programs in SPL are written as lines of text; the line numbers have no significance, except when editing. Spaces must be used to separate words, and must not occur within words, but otherwise they are ignored and can be added to make the structure of programs clearer.

### Symbols

All variable names, array names, procedure names, and labels, can consist of up to six letters, all of which are significant. None of the language words may be used as symbols. On the BBC Computer all symbols and language words are in lower case; however, on the Atom, and in the following examples, upper case is used.

## Comments

Any text between brackets '{' and '}' is ignored by the compiler, and can be inserted, to comment SPL programs, anywhere spaces are permitted.

## Programs

Programs normally consist of a procedure such as the following:

```
PROC MAIN();
BEGIN
    stmt;
    stmt;
    ...
    stmt
END
```

where 'stmt' represents any of the statements described below. The statements in the body of the procedure between BEGIN and END are separated by semi-colons.

## SPL Statements

SPL contains the following statements:

### Array Declaration

Arrays are declared with the ARRAY statement. For example:

```
ARRAY MAX[3],B[2]
```

reserves space for arrays with elements:

```
MAX[0], MAX[1], MAX[2], MAX[3], and
B[0], B[1], and B[2].
```

These array elements can then be used in the same way as simple variables. Arrays can have up to 255 elements, and there is no checking that arrays are within bounds.

### Procedure Declaration

Any number of procedures, with one parameter, can be declared. For example:

```
PROC ADD(X); statement
```

and the procedure is called by:

```
ADD(expression)
```

The parameter is a local parameter; i.e. the use of X in the procedure ADD above does not affect the value of X outside the procedure. Thus, in the following example:



```
PROC INC(X); X=X+1
ENTER: WRHEX(X); INC(X); WRHEX(X)
```

the two calls to WRHEX, which print the value of X, both print the same value.

The single parameter is optional; thus a procedure can be defined:

```
PROC INC(); J=J+1
```

which alters the value of J outside the procedure. Note that a procedure cannot be declared inside another procedure which has a parameter.

### Function Declaration

A function is identical to a procedure, except that its last statement is a RETURN statement specifying the result to be returned. For example:

```
PROC ADDONE(N); RETURN N+1
```

defines a function ADDONE whose value is one greater than its argument. Thus:

```
WRHEX(2+ADDONE(3))
```

would print '06'.

### Assignment

The assignment statement is of the form:

```
variable = expression
```

where the variable is an identifier or an array element. For example:

```
TIME=TIME+1
```

```
SYMBOL[RDCH()]=A-3
```

### Operators

Expressions can use any of the following operations, which work on 8-bit numbers:

+	: add	-	: subtract
&	: logical AND		: logical OR
>>	: right shift	<<	: left shift

All these operators have the same priority, and brackets can be used to alter the order of evaluation. The shift operators shift the left-hand operand the number of places specified by the right-hand operand, which must be a constant as in:

```
A=B>>3      (equivalent to A=B/8)
```

```
A=B<<4      (equivalent to A=B*16)
```

## GOTO Statement

Any statement can be prefixed by a label:

```
LOOP: A=A+1
```

A jump can be made to the labelled statement by means of the GOTO statement, as in:

```
GOTO LOOP
```

## IF Statements

The IF statement has the form:

```
IF condition THEN statement
```

in which the statement is only executed if the condition is true. There may also be an ELSE clause:

```
IF condition THEN statement ELSE statement
```

in which case the second statement will only be executed if the condition is false. Note that if one of the statements is itself an IF...THEN...ELSE statement the ELSE statement associates with the nearest IF condition. For example:

```
IF A=1 THEN  
  IF A>0 THEN WRHEX(1)  
            ELSE WRHEX(2)
```

will, if A=2, write nothing.

## Conditions

The condition in an IF statement is of the form:

```
expression comparison expression
```

where the comparison is one of:

> : greater than	<= : less than or equal
< : less than	>= : greater than or equal
= : equal	<> : not equal

## BEGIN...END Block

Any number of statements can be grouped together within a BEGIN...END block, which has the format:

```
BEGIN  
  statement;  
  statement;  
  ...  
  statement  
END
```

The entire block then has the same status as a single statement. Note that the semi-colons are used as

statement separators, not as statement terminators, and so there is no need for a semi-colon before the END statement.

### Pre-defined Symbols

The following symbols are defined in both versions of the compiler:

Symbol	Operation	Example
RDCH	Function to read a character	A=RDCH()-48
WRCH	Procedure to write a character	WRCH(32)
SCREEN	Array of 256 screen locations	SCREEN[0]=0

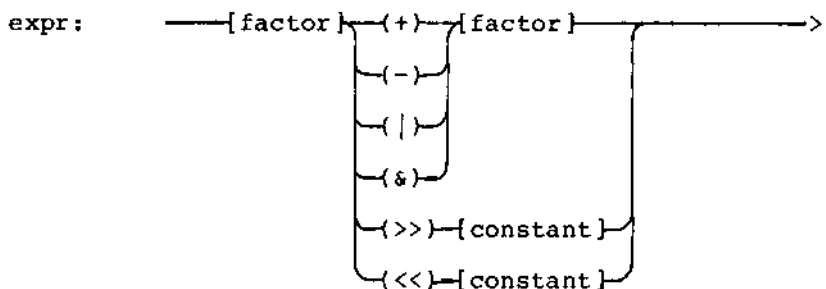
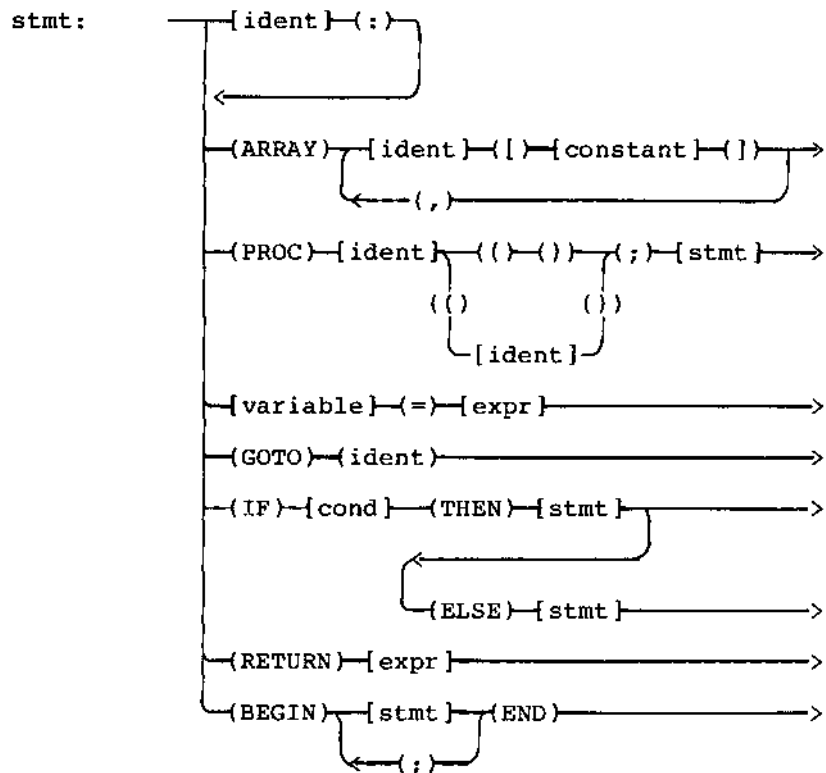
In addition, in the Atom version, the following symbols are defined:

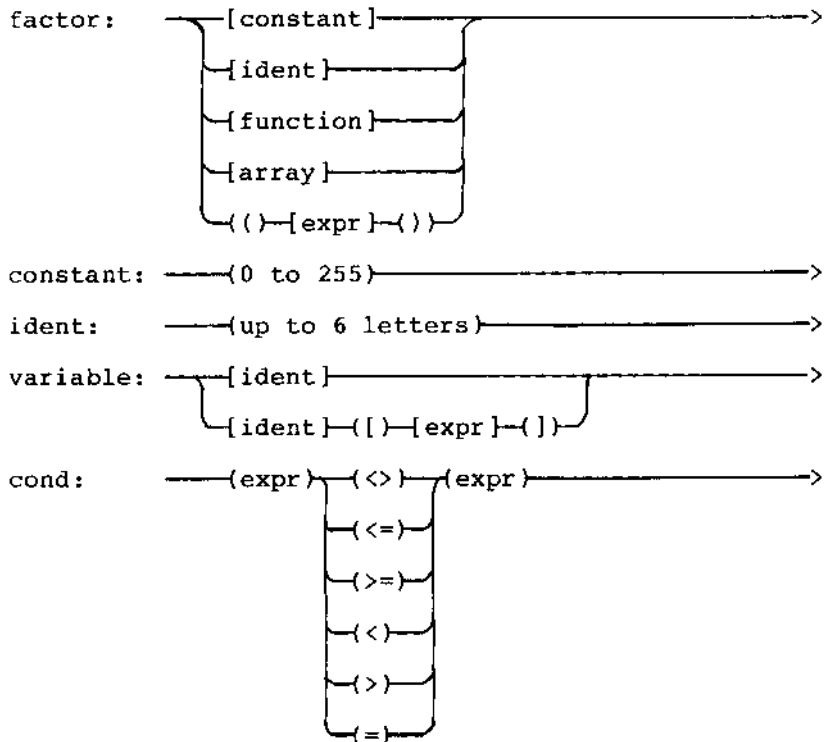
Symbol	Operation	Example
WRHEX	Procedure to print in hex	WRHEX(255)
PORT	Array of I/O ports	PORT[2]=4

### SPL Syntax Diagrams

A more formal definition of the syntax of a language like SPL can be given using 'bead' diagrams as shown below. The diagrams can be used to work out whether a given program is legal in terms of the language. Each construct, such as 'expr:', is defined by travelling along the line to its right, following the arrows. Constructs in square brackets, such as [stmt], are defined by referring to another diagram. Constructs in round brackets, such as (ARRAY) or (:), refer to keywords or symbols in the language. The language is defined recursively, so certain constructs, such as 'stmt:', contain references to themselves.

program: — { stmt } —>





## SPL PROGRAMS

The following demonstration programs can all be compiled into machine-code by the Compiler program to be described, and run on the Atom or BBC Computer. They illustrate some of the features of SPL.

### Bubble Sort

The first program performs a bubble sort of the characters in the top half of the screen memory. The sort works by successively comparing pairs of adjacent locations; if two are in the wrong order they are exchanged, and then the smaller one is moved back to its correct position in the locations that have already been sorted.

```
5 {BUBBLE SORT OF SCREEN}
10 PROC BUBBLE();
```

```

15 BEGIN I=0;
20 LOOP:J=I;
25 LOOPA:IF SCREEN[J]>SCREEN[J+1] THEN
28 BEGIN
30 TEMP=SCREEN[J];
31 SCREEN[J]=SCREEN[J+1];
32 SCREEN[J+1]=TEMP;
35 IF J=0 THEN GOTO OK;
40 J=J-1;GOTO LOOPA
42 END;
45 OK:I=I+1;IF I<255 THEN GOTO LOOP
60 END

```

When the compiled program is executed the characters on the top part of the screen will be sorted into order.

### Crawling Snake

The following SPL program moves a snake across the top half of the screen, and demonstrates the use of the language's shift operators '>>' and '<<'. The program is only suitable for the Atom, and uses a routine WAIT to make sure that the screen accesses are noise-free. After being compiled it should be executed by linking to the address corresponding to the label ENTER.

```

5 {CRAWLING SNAKE}
10 PROC SNAKE();
15 BEGIN
20 PROC CLEAR(K);
25 {CLEAR SCREEN TO CHARACTER K}
28 BEGIN
30 X=0;CLR:SCREEN[X]=K;
35 X=X+1;IF X<>0 THEN GOTO CLR
40 END;
42 PROC WAIT();
43 {WAIT FOR FLYBACK SYNC}
44 BEGIN
45 WA:IF PORT[2]>127 THEN GOTO WA
46 END;
47 ENTER:L=0;CLEAR(64);
48 SCREEN[0]=127;SCREEN[1]=127;
49 SCREEN[2]=127;SCREEN[3]=127;
50 RUN:X=0;
60 LOOP:WAIT();
70 C=(SCREEN[X]&63)<<2;
80 SCREEN[X]=C|192+L;L=C>>6;
90 X=X+1;
95 IF X<>0 THEN GOTO LOOP ELSE GOTO RUN
100 END

```

## Write Hexadecimal

The following procedure will print a number as two hexadecimal numbers on the screen. On the Atom this routine is pre-defined.

```
40 proc wrhex(n);
50 begin
60   if n>=160 then wrch(n>>4+55)
70   else wrch(n>>4+48);
80   n=n&15;
90   if n>=10 then n=n+7;
100  wrch(n+48)
110 end;
```

When compiled and executed the procedure will print the contents of the accumulator in hexadecimal.

## Prime Numbers

The following SPL program finds all the prime numbers up to 127, and prints them in hexadecimal:

```
10 PROC MAIN();
20 BEGIN
30   PROC PRIME(N);
35   {PRINT N IF PRIME}
40   BEGIN D=1;
60     TRY:D=D+1;E=N;
65     IF D<N-1 THEN
68       BEGIN
70         TEST:E=E-D;
75         IF E<>0 THEN
80           IF E<128 THEN GOTO TEST
85           ELSE GOTO TRY
88         END
90         ELSE WRHEX(N)
110      END;
115 {MAIN PROGRAM}
120   ENTER:T=1;
125   ALL:PRIME(T);WRCH(32);
130   IF T<128 THEN
140     BEGIN
150       T=T+1;GOTO ALL
170     END
190 END
```

On the BBC Computer the definition of the 'wrhex' routine should be inserted at the start of the program between lines 20 and 30. The compiled program, when executed, prints the following sequence:

```

01 02 03 05 07 0B 0D 11 13 17 1D 1F
    25 29 2B 2F 35 3B 3D 43
47 49 4F 53 59 61 65 67 6B
    6D 71 7F

```

It prints a space for every number tested, so the primes can be seen to become more sparse as they get larger.

### Greatest Common Divisor

The following function finds the greatest common divisor (GCD) of the numbers A and B using Euclid's algorithm:

```

5 PROC TEST(); BEGIN
10 PROC GCD(); {GCD OF A,B IN B}
20 BEGIN LOOP:IF A<>B THEN
25 BEGIN
30 IF A<B THEN B=B-A
35 ELSE A=A-B END;
40 RETURN B END;
45 ENTER:A=9; B=12; WRHEX(GCD())
50 END

```

The routine, once compiled, should be entered at ENTER, when the GCD of 9 and 12, i.e. 3, will be displayed. Again, for the BBC Computer version the 'wrhex' routine should be included since it is not pre-defined.

### Multiply Routine

The following test program demonstrates an 8-bit multiply routine written in SPL:

```

5 PROC TEST();
8 BEGIN
10 PROC MULT();
15 {A*B - RESULT IN C}
20 BEGIN C=0;
25 MULL:IF B>0 THEN
28 BEGIN
30 IF B&1=1 THEN C=C+A;
40 A=A<<1;B=B>>1;GOTO MULL
45 END
50 END;
8 ENTER:A=6;B=19;MULT();WRHEX(C)
60 END

```

The machine code is executed from the address corresponding to the label ENTER, and should print out



'72'; in other words,  $6*19=114$ , or 72 in hexadecimal.

### Mastermind

In the following SPL Mastermind program the computer generates a random 4-digit code, which the player must guess. The guess is entered as four decimal digits, and the computer displays the result as two digits: the first digit gives the number of digits correctly guessed in the correct position; the second digit gives the number of correct digits incorrectly placed. When the computer's code is correctly guessed, with a score of '40', the program gives a 'bleep'.

The following sample run shows each of the player's 4-digit guesses followed by the computer's 2-digit reply:

```
1122 00
3344 10
5566 00
7788 20
9900 10
3780 02
7948 40
```

The version of the program shown below is for the BBC Computer; on the Atom all the symbols and variables should be in upper case, and lines 40 to 110 can be omitted.

```
10 {mastermind}
20 proc mastr();
30 begin array my[3],your[3],temp[3];
40 proc wrhex(n);
50 begin
60   if n>=160 then wrch(n>>4+55)
70   else wrch(n>>4+48);
80   n=n&15;
90   if n>=10 then n=n+7;
100  wrch(n+48)
110 end;
120 proc rnd();
130 begin rndy:rndx=rndx<<2+rndx+7;
140   if rndx&15>9 then goto rndy;
150   return rndx&15
160 end;
170 proc input();
180 begin n=0;
190   readc:j=rdch();wrch(j);j=j-48;
200   if j>9 then goto readc;
210   your[n]=j;n=n+1;
220   if n<4 then goto readc
```

```

230 end;
240 {main program}
250 enter:n=0;
260 myno:my[n]=rnd();n=n+1;
270 if n<4 then goto myno;
280 try:input();n=0;
290 copy:temp[n]=my[n];n=n+1;
300 if n<4 then goto copy;
310 n=0;score=0;
320 bull:if temp[n]=your[n] then
330   begin temp[n]=10;your[n]=11;
340     score=score+16
350   end;
360 n=n+1;if n<4 then goto bull;
370 n=0;
380 cow:m=0;
390 cowx:if temp[n]=your[m] then
400   begin temp[n]=10;your[m]=11;
410     score=score+1
420   end;
430 m=m+1;if m<4 then goto cowx;
440 n=n+1;if n<4 then goto cow;
450 wrch(32);wrhex(score);wrch(10);wrch(13);
460 if score<>64 then goto try;
470 wrch(7)
480 end

```

## COMPILER

The last program in this book is the complete compiler which will take a program written in SPL and convert it into machine code for the 6502.

The compiler program, and the SPL program to be compiled, are first entered into different parts of memory. On running the compiler the machine-code will be generated, and put into memory where it can be executed. As given below the two versions of the compiler use the following memory areas:

Use:	BBC Computer:	Atom:
Compiler program	&E00-&27FF	#8200-#9800
SPL program	&2800-&3700	#2900-#39FF
Machine code	&3800-&3BFF	#3A00-#3BFF

The procedure for using the compiler is as follows:

On the BBC Computer first type:

PAGE=&2800

NEW

and enter the SPL program as you would a BASIC program. Since the symbols are in lower case there is no danger of them being converted into tokens by BASIC. Having done this, type:

PAGE=&E00

NEW

and either type in, or load from tape, the Compiler program. Then type:

RUN

The compiler performs two identical passes so that the assembler will resolve forward references. The first pass is performed with the screen turned off; then the message:

PRINT?

is given to allow CTRL-B to be typed to turn on the printer for a listing. Typing RETURN will then give the assembler listing statement by statement. After the second pass a symbol table will be printed, showing the addresses corresponding to all the symbols used by the program. Finally, to execute the machine-code generated by the compiler type:

CALL &3800

where &3800 is the start of the machine code. Some of the programs should be entered not at the start of the machine code, but at the address corresponding to the label 'enter'; this address can be found from the symbol table.

On the Atom the corresponding sequence is as follows. First type:

?18=#29

NEW

and enter the SPL program. Then type:

?18=#82

NEW

and load the compiler. RUN as above, and then to execute the machine code type:

LINK #3A00

or the address corresponding to the label 'ENTER', if present.

### Sample Run

The following run shows the assembler listing produced by the Atom version of the Compiler for the Bubble

Sort program given above:

```
{BUBBLE SORT OF SCREEN}
PROC BUBBLE();
BEGIN I=0;

7210 3A00 A9 00    LDA @-L,
7200 3A02 85 51    STA H

    LOOP:J=I;

7040 3A04 A5 51    LDA L
7200 3A06 85 52    STA H

    LOOPA:IF SCREEN[J]>SCREEN[J+1] THEN

7040 3A08 A5 52    LDA L
3685 3A0A AA        TAX
3685 3A0B BD 00 80 LDA V,X
7200 3A0E 85 81    STA H
7040 3A10 A5 52    LDA L
3070 3A12 18        CLC
3070 3A13 69 01    ADC @-U
3685 3A15 AA        TAX
3685 3A16 BD 00 80 LDA V,X
7200 3A19 85 82    STA H
7040 3A1B A5 81    LDA L
7320 3A1D C5 82    CMP M
4201 3A1F F0 02    BEQ P+4
4201 3A21 B0 03    BCS P+5
4201 3A23 4C 60 3A JMP LLG

    BEGIN
    TEMP=SCREEN[J];

7040 3A26 A5 52    LDA L
3685 3A28 AA        TAX
3685 3A29 BD 00 80 LDA V,X
7200 3A2C 85 53    STA H

    SCREEN[J]=SCREEN[J+1];

7040 3A2E A5 52    LDA L
3070 3A30 18        CLC
3070 3A31 69 01    ADC @-U
3685 3A33 AA        TAX
3685 3A34 BD 00 80 LDA V,X
1330 3A37 A6 52    LDX L
1330 3A39 9D 00 80 STA V,X

    SCREEN[J+1]=TEMP;

7040 3A3C A5 52    LDA L
3070 3A3E 18        CLC
3070 3A3F 69 01    ADC @-U
7200 3A41 85 81    STA H
7040 3A43 A5 53    LDA L
1330 3A45 A6 81    LDX L
```

1330 3A47 9D 00 80 STA V,X

IF J=0 THEN GOTO OK;

7040 3A4A A5 52 LDA L  
4110 3A4C C9 00 CMP @-M  
4202 3A4E F0 03 BEQ P+5  
4202 3A50 4C 56 3A JMP LLG  
2270 3A53 4C 60 3A JMP U  
1445 3A56 :LLV

J=J-1;GOTO LOOPA

7040 3A56 A5 52 LDA L  
3075 3A58 38 SEC  
3075 3A59 E9 01 SBC @-U  
7200 3A5B 85 52 STA H  
2270 3A5D 4C 08 3A JMP U

END;

1445 3A60 :LLV

OK:I=I+1;IF I<255 THEN GOTO LOOP

7040 3A60 A5 51 LDA L  
3070 3A62 18 CLC  
3070 3A63 69 01 ADC @-U  
7200 3A65 85 51 STA H  
7040 3A67 A5 51 LDA L  
4110 3A69 C9 FF CMP @-M  
4204 3A6B 90 03 BCC P+5  
4204 3A6D 4C 73 3A JMP LLG  
2270 3A70 4C 04 3A JMP U

END

1445 3A73 :LLV  
6040 3A73 60 RTS

SYMBOLS:

FFE6 RDCH  
FFF4 WRCH  
F802 WRHEX  
8000 SCREEN  
B000 PORT  
3A00 BUBBLE  
51 I  
3A04 LOOP  
52 J  
3A08 LOOPA  
53 TEMP  
3A60 OK

BBC Computer Version

5 REM ... Compiler ...

```
10 HIMEM=&2800
20 DIM SS(20),LAB(20),ID$(30),JJ(30)
30 DIM TT(20),X&7:A$=CHR$(6)
```

Important addresses:

```
MC - Start address for machine code.
VARS - Start address for variables and arrays.
TEMPS - 20 temporary locations.
PRARG - Location for use by procedures for
argument.
SADD - Source program address.
```

```
35 MC=&3800:VARS=&50:TEMPS=&80:PRARG=&94:SADD=&2800
```

Pre-defined symbols:

```
rdch()      Function reads a character.
wrch(X)     Procedure writes character X in ASCII.
screen[0] ... screen[255] Array to access screen.
```

```
40 ID$(0)="rdch":JJ(0)=&FFEO
50 ID$(1)="wrch":JJ(1)=&FFEE
60 ID$(2)="screen":JJ(2)=&7C00
70 FOR N=0TO20:TTN=0:NEXT
```

Now do compilation; first pass with screen disabled, and second pass with screen enabled. Finally print symbol table.

```
200 PRINT CHR$(21):PROCOMPILE
220 PRINTA$"PRINT";:INPUTB$:PROCOMPILE
225 PRINT "SYMBOLS:"
230 FORN=0TOI-1:PRINT"JJ(N), " ",ID$(N):NEXT
240 END
```

PROCOMPILE - One pass of compilation. Initialise pointers, with I=3 since there are 3 pre-defined symbols. Then compile statement, and make sure accumulator is stored finally.

```
900 DEF PROCOMPILE:G=0:A=SADD:I=3:P&=MC
910 S=0:R=0:H=0:T=0
920 PROCSTMT:PROCSTA:ENDPROC
```

PROCSTMT - Statement. Skip blanks, read symbol, then check for keywords. Ignore 'end' if found.

```
1000 DEF PROCSTMT
1010 PROCSP:PROCSYM
1020 IF$X&="if"GOTO1400
1030 IF$X&="begin"GOTO1200
1040 IF$X&="goto"GOTO1500
1045 IF$X&="end"A=A-3:ENDPROC
```

```
1050 IF$X%="proc"GOTO1700
1060 IF$X%="array"GOTO1800
1070 IF$X%="return"GOTO1900
```

If the symbol is not a keyword then it must be a label, an assignment statement, or a procedure call.

```
1100 REM IDENT STATEMENT
1110 PROCSP
1120 IF?A=ASC": "A=A+1:PROCVAR:JJ(N)=P%:PROCSTMT:
ENDPROC
1130 IF?A=ASC>("PROCVAR:PROCPSE(U):PROCBODY:ENDPROC
1135 PROCV:IF?A=ASC("[") GOTO 1300
1160 PROCRRHS:H=V:PROCSTA:ENDPROC
```

PROCRHS - Right-hand side of assignment statement.

```
1180 DEF PROCRHS
1185 IF ?A<>ASC"=" PRINTA$"NO =" :PROCERR
1190 A=A+1:PROCEXP:L=FNPUL
1195 PROCLDA:V=FNPUL:ENDPROC
```

'begin' - Deal with 'begin' ... 'end' block.

```
1200 REM BEGIN...END
1210 A=A-1: REPEAT A=A+1
1220 PROCSTMT:PROCSP
1230 UNTIL ?A<>ASC";"
1240 PROCSYM:IF$X%="end"ENDPROC
1250 PRINT"NO END":PROCERR
```

Array element on the left-hand-side of an assignment statement.

```
1300 REM ARRAY=
1310 A=A+1:PROCEXP:IF?A<>ASC]"PRINTA$"NO ]":PROCERR
1320 A=A+1:PROCRHS:L=V:V=FNPUL
1325 IFL<=0[LDX @-L:STA V,X]:H=0:ENDPROC
1330 [LDX L:STA V,X]:H=0:ENDPROC
```

'if' - Assemble code to evaluate condition, and following 'then' assemble code to execute a statement. Pull label from stack and assemble label. Deal with 'else' clause.

```
1400 REM IF...THEN...ELSE
1410 PROCLOGICAL:PROCSP
1420 PROCSYM:IF$X%="then"GOTO1430
1425 PRINTA$"NO then":PROCERR
1430 PROCSTMT:V=FNPUL:PROCSP
1440 PROCSYM:IF$X%="else"GOTO1460
```

```
1445 A=A-N:[.LAB(V):]:ENDPROC
1460 G=FNLAB:U=G:PROCP SH(U):[JMP LAB(G):]
1470 [.LAB(V):]:PROCSTMT
1490 V=FNPUL:[.LAB(V):]:ENDPROC
```

'goto' - Get label and assemble jump to it.

```
1500 REM GOTO
1510 PROCLABEL:[JMP U:]
1520 ENDPROC
```

'proc' - Get name and set its value to entry address P. Then get dummy parameter.

```
1700 REM PROC
1710 PROCLABEL:JJ(N)=P%:IF?A<>ASC>("PRINTA$"MISSING
BRACKET":PROCERR
1720 A=A+1:JJ(I)=1:PROCIDENT:IFN=0GOTO1780
1730 U=N:PROCP SH(U)
1740 T=PRARG:H=T:JJ(U)=T:PROCSTA
1745 IF?A<>ASC("PRINTA$"NO BRACKET":PROCERR
1750 A=A+1:PROCP SP:IF?A<>ASC(";PRINTA$"NO ;":PROCERR
1760 A=A+1:PROCSTMT
1770 V=FNPUL:N=V:V=FNPUL:JJ(N)=V:[RTS:]:ENDPROC
```

Come here if procedure has no parameter.

```
1780 IF?A<>ASC("PRINTA$"NO BRACKET":PROCERR
1782 A=A+1:PROCP SP:IF?A<>ASC(";PRINTA$"NO ;":PROCERR
1785 A=A+1:PROCSTMT:[RTS:]:ENDPROC
```

'array' - Look up array name; assign space from VARS onwards. Allow multiple declarations, separated by commas.

```
1800 REM ARRAY
1810 A=A-1: REPEAT A=A+1
1820 PROCP SP:PROCSYM:PROCLook
1830 IFN<>I PRINTA$"ARRAY DECLARED":PROCERR
1840 IF?A<>ASC("[PRINTA$"BRACKET MISSING":PROCERR
1850 A=A+1:PROCONST:IFN=0PRINTA$"CONSTANT
MISSING":PROCERR
1860 V=FNPUL:JJ(I)=VARS+R:I=I+1:R=R-V+1
1870 IF?A<>ASC("]PRINTA$"BRACKET MISSING":PROCERR
1880 A=A+1:PROCP SP:UNTIL?A<>ASC(",":ENDPROC
```

'return' - Assemble code to load accumulator with expression.

```
1900 REM RETURN
1910 PROCP SP:V=FNPUL:L=V:PROCLDA:H=0:ENDPROC
```



PROCIDENT - Read an identifier.

```
2000 DEF PROCIDENT
2010 PROCSYM:PROCV:ENDPROC
```

PROCV - Look up identifier \$X in symbol table. If symbol does not already exist (N=I) allocate address for it. Push address to stack.

```
2020 DEF PROCV
2030 IF N=0 ENDPROC
2040 PROCLOOK
2050 IFN=I:I=I+1:R=R+1:JJ(N)=R+VARS
2070 IFI>30PRINT"TOO MANY VARIABLES":PROCERR
2080 U=JJ(N):PROCP SH(U):ENDPROC
```

PROCONST - Read a decimal constant. If not found, N=0. If found, push minus its value.

```
2100 DEF PROCONST
2105 PROCSP
2110 N=-1:C=0:REPEAT D=C:N=N+1:C=C*10
2120 C=C+A?N-ASC"0"
2130 UNTILA?N<ASC"0"ORA?N>ASC"9"
2140 IFN=0 ENDPROC
2150 A=A+N
2160 U=-D:PROCP SH(U):N=1:ENDPROC
```

PROCLABEL - Read label.

```
2200 DEF PROCLABEL
2210 PROCSP:PROCSYM
2220 IF N=0 PRINT"LABEL MISSING":PROCERR
2225 PROCVAR:ENDPROC
```

PROCVAR - Look up label in symbol table. If not found (N=I) put it in. Return its address in U.

```
2230 DEF PROCVAR:PROCLOOK
2250 IFN=I:I=I+1
2260 IFI>30PRINTA$"TOO MANY VARIABLES":PROCERR
2270 U=JJ(N):ENDPROC
```

PROCLOOK - Look up \$X% in symbol table, ID\$(0), ID\$(1) ... If not found, N=I.

```
2400 DEF PROCLOOK
2410 ID$(I)=$X%:N=-1
2420 REPEAT N=N+1:UNTILID$(N)=ID$(I):ENDPROC
```

PROCEXP - Assemble code to calculate an expression, of the form:

<factor> <operator> <factor>

where <operator> is one of:

+ : add                   - : subtract  
| : OR                     & : AND  
<< : left shift         >> : right shift

Then push the address of the result on the stack.

```
3000 DEF PROCEXP PROCSP:PROCFACOR
3010 PROCSP
3020 IF?A=ASC"+"OR?A=ASC"- "OR?A=ASC"&"
OR?A=ASC"|"O=?A:A=A+1:GOTO 3035
3025 IF NOT((?A=ASC">"AND A?1=ASC">")OR (?A=ASC"<"AND
A?1=ASC"<"))ENDPROC
3030 O=?A:A=A+2
3035 PROCPSH(O)
3040 PROCFACOR:U=FNPUL:O=FNPUL:L=FNPUL:PROCLDA
3045 IF U<=0 GOTO 3070
3050 IFO=ASC"+"[CLC:ADC U:]
3055 IFO=ASC"~"[SEC:SBC U:]
3060 IFO=ASC"|" [ORA U:]
3065 IFO=ASC"&"[AND U:]
3068 GOTO 3190
3070 IFO=ASC"~"[CLC:ADC #-U:]
3075 IFO=ASC"~"[SEC:SBC #-U:]
3080 IFO=ASC"|" [ORA #-U:]
3085 IFO=ASC"&"[AND #-U:]
3160 IFO=ASC"<"FOR N=1TO-U:[ASL A:]:NEXT
3180 IFO=ASC">"FOR N=1TO-U:[LSR A:]:NEXT
3190 L=U:PROCRELEASE(L):PROCTEMP
3195 GOTO 3010
```

PROCBODY - Procedure body. Check for ')'. If there is a parameter first assemble code to calculate parameter, load it into the accumulator, and then JSR. Assume subroutine alters accumulator, so set H=0.

```
3200 DEF PROCBODY
3210 IFA?1=ASC")"A=A+2:GOTO3230
3220 PROCFACOR:V=FNPUL:L=V:PROCLDA
3230 V=FNPUL:[JSR V:]:H=0:ENDPROC
```

PROCFACOR - Factor. Check for symbol, constant, or bracketed expression. If the symbol is followed by '(' or '[' then it is a function or an array element respectively.

```
3600 DEF PROCFACOR
3610 PROCSYM:IFN=0GOTO3630
```

```

3615 IF?A=ASC>("GOTO3690
3620 PROCV:IF?A=ASC["GOTO3670
3625 ENDPROC
3630 PROCONST:IF N ENDPROC
3635 IF?A<>ASC(" PRINTA$"BRACKET MISSING":PROCERR
3640 A=A+1:PROCEXP:PROCSP
3650 IF?A<>ASC)" PRINTA$"BRACKET MISSING":PROCERR
3660 A=A+1:ENDPROC

```

Evaluate array element. Assemble code to evaluate array index and load it into the accumulator; then TAX and load indexed by the base address.

```

3670 REM ARRAYS
3675 A=A+1:PROCEXP:IF?A<>ASC]"PRINTA$"NO
BRACKET":PROCERR
3680 A=A+1:L=FNPUL:PROCLDA:V=FNPUL
3685 [TAX:LDA V,X:]:PROCTEMP:ENDPROC

```

Call function here.

```

3690 PROCVAR:PROCPSH(U):PROCBODY:PROCTEMP:ENDPROC

```

PROCLOGICAL - Logical expression. Look for:  
<expression> <comparison> <expression>

```

4000 DEF PROCLOGICAL
4010 PROCSP:PROCEXP:PROCSP

```

Expect a comparison here; look for '<', '>', and '=' and set value of U depending on sequence:

```

> : 1           = : 2
>= : 3          < : 4
<> : 5          <= : 6

```

Then use a computed GOTO to assemble code for each case.

```

4020 U=0
4030 IF?A=ASC"<"A=A+1:U=4
4040 IF?A=ASC">"A=A+1:U=U+1
4050 IF?A=ASC"="A=A+1:U=U+2
4060 IFU=0 OR U>6 PRINT"ILLEGAL TEST":PROCERR
4070 PROCPSH(U):PROCEXP
4080 M=FNPUL:U=FNPUL
4090 L=FNPUL:PROCLDA
4100 IFM>0[CMP M:]
4110 IFM<=0 [CMP #-M:]

```

First generate a label LAB(G). Then assemble code for the comparison. Note that if the condition is true we branch around a jump to LAB(G). Push value of LAB(G) for use by IF...THEN statement.

```
4120 PROCRELEASE(M):G=FNLAB:GOTO(4200+U)
4201 [BEQ P#+4:BCS P#+5:]:GOTO4210
4202 [BEQ P#+5:]:GOTO4210
4203 [BCS P#+5:]:GOTO4210
4204 [BCC P#+5:]:GOTO4210
4205 [BNE P#+5:]:GOTO4210
4206 [BCC P#+7:BEQ P#+5:]:GOTO4210
4210 [JMP LAB(G):]
4220 U=G:PROCPSH(U):H=0:ENDPROC
```

PROCPSH - Push argument onto stack.

```
5020 DEF PROCPSH(U):SS(S)=U:S=S+1:IFS<21 ENDPROC
5021 PRINTA$"STACK FULL":PROCERR
```

FNPUL - Pull from stack.

```
5030 DEF FNPUL:S=S-1:IFS>=0 =SS(S)
5031 PRINTA$"STACK ERROR":PROCERR
```

PROCSP - Skip blanks, line numbers, and comments between '{' and '}'.

```
5040 DEF PROCSP
5042 IF?A=32 REPEAT A=A+1:UNTIL?A<>32
5046 IF?A=13A=A+4:PRINT $A:GOTO5042
5048 IF?A=ASC{"REPEAT A=A+1:UNTIL?A=ASC"}":
A=A+1:GOTO5042
5049 ENDPROC
```

FNLAB - Return a new label number. Label is LAB(G).

```
5070 DEF FNLAB:G=G+1:IF G<20 =G
5071 PRINTA$"TOO MANY LABELS":PROCERR
```

PROCTEMP - Return the address of a temporary location TT(N); return its address in T, set H to the address, and push the address.

```
5100 DEF PROCTEMP
5110 N=-1:REPEAT N=N+1: IF N>20PRINTA$"NOT ENOUGH
TEMP":PROCERR
5120 UNTILTT(N)=0
5130 T=N+TEMPS:TT(N)=T:U=T:H=T:PROCPSH(U):ENDPROC
```

PROCSYM - Read a symbol into \$X% from \$A. Returns N=0 if no symbol found.

```
6000 DEF PROCSYM
6010 PROCSP:N=-1:REPEAT N=N+1: N?X%=A?N
6020 UNTILA?N>ASC "z"ORA?N<ASC "a"ORN=7
6030 IF N=0 ENDPROC
6040 IF N<7 N?X%=&D:A=A+N:ENDPROC
6050 PRINTA$"SYMBOL TOO LONG":PROCERR
```

PROCLDA - Assemble code to load the accumulator with L. If accumulator already contains L (L=H) then do nothing; otherwise store its previous contents and load new contents.

```
7000 DEF PROCLDA
7010 IFL=H AND L>0 PROCRELEASE(L):ENDPROC
7020 PROCSTA
7030 IFL<=0 [LDA #-L:]:ENDPROC
7040 [LDA L:]:PROCRELEASE(L)
7050 ENDPROC
```

PROCSTA - Assemble code to store accumulator's contents to location H.

```
7100 DEF PROCSTA
7200 IFH>0[STA H:]:H=0
7210 ENDPROC
```

PROCRELEASE - Release specified temporary variable for re-use.

```
7300 DEF PROCRELEASE(L)
7310 IF L>=TEMPS AND L<TEMPS+20:TT(L-TEMPS)=0
7320 ENDPROC
```

PROCERR - Output error. Print line containing error and '^' pointing to approximate position.

```
9000 DEF PROCERR
9010 N=A:X=0:REPEAT N=N-1:X=X+1:UNTIL?N=13:@%=5
9020 PRINT'N?1*256+N?2,$(N+4)
9030 PRINT TAB(X+2);"^":END
```

Variables:

- A - Pointer to current position in expression being compiled
- C - Used to evaluate constant
- G - Number of next free label LAB(G)
- H - Address whose contents are currently in accumulator. H=0 means ignore previous contents

I - Number of next free symbol  
 ID\$(0)..ID\$(30) - Symbol names  
 JJ(0)..JJ(30) - Addresses of symbols  
 L - Value or address to be loaded into accumulator;  
 used by PROCLDA  
 LAB(0)..LAB(20) - Labels for use in assembly  
 MC - Assemble machine code to here  
 N - Temporary variable  
 O - Operator read by PROCEXP  
 P - Program location counter, used by assembler  
 PRARG - Location for use by procedures for argument  
 R - Number of variable locations used up  
 S - Next free location on SS stack  
 SADD - Source program address  
 SS(0)..SS(20) - Stack used by compiler  
 T - Temporary location assigned by PROCTEMP  
 TEMPS - 20 temporary locations start here  
 TT(0)..TT(20) - Flags for temporary locations; value=0  
 if location is free for use  
 U - Value pushed by PROCPSH  
 V - Used by FNPUL  
 VARS - Allocate variables and arrays starting here  
 X% - String into which symbols and keywords are read  
 by PROCSYM

### Atom Version

To save program space in this version of the compiler  
 abbreviated forms of many of the BASIC statements and  
 commands have been used, and for convenience these are  
 listed below:

Abbreviation:	Keyword:
A.	AND
E.	END
F.	FOR
G.	GOTO
GOS.	GOSUB
N.	NEXT
P.	PRINT
R.	RETURN
U.	UNTIL

```

10 REM ... COMPILER ...
20 DIM SS(20),LL(20),II(30),JJ(30)
30 DIM X(7),TT(20),RR(4)
  
```

Important addresses:

RR0 - Start of machine code.  
RR1 - Start of variables and arrays.  
RR2 - 20 temporary locations.  
RR3 - Location for use by procedures for argument.  
RR4 - Source program address.

35 RR0=#3A00;RR1=#50;RR2=#80;RR3=#94;RR4=#2900  
40 F.N=0TO30;DIMI(6);IIN=I;JJN=RR0;N.

Pre-defined symbols:

RDCH() Function reads a character.  
WRCH(X) Procedure writes character X.  
WRHEX(X) Procedure writes X as two hex digits.  
SCREEN[0] ... SCREEN[255] Array to access screen.  
PORT[0] ... PORT[3] Array to access I/O ports.

50 \$IIO="RDCH";JJO=#FFE3;\$I11="WRCH";JJ1=#FFF4  
60 \$I12="WRHEX";JJ2=#F802;\$I13="SCREEN";JJ3=#8000  
70 \$I14="PORT";JJ4=#B000  
115 F.N=0TO20;TTN=0;LLN=RR0;N.

Now do compilation; first pass with screen disabled, and second pass with screen enabled. Finally print symbol table.

210 P.\$21;GOS.a  
220 P.\$6'"PRINT";IN.\$100;GOS.a  
228 P.'"SYMBOLS:"'  
230 F.N=0TOI-1;P.&JJN," ",\$IIN';N.  
240 E.

a - One pass of compilation. Initialise pointers, with I=5 since there are 5 pre-defined symbols. Then compile statement, and make sure accumulator is stored finally.

900aG=0;A=RR4;I=5;P=RR0  
910 S=0;R=0;H=0;T=0  
920 GOS.s;GOS.m;R.

s - Statement. Skip blanks, read symbol, then check for keywords. Ignore END if found.

1000sREM STATEMENT

1010 GOS.b;GOS.x  
1020 IF\$X="IF"G.1400  
1030 IF\$X="BEGIN"G.1200  
1040 IF\$X="GOTO"G.1500  
1045 IF\$X="END"A=A-3;R.  
1050 IF\$X="PROC"G.1700

```
1060 IF$X="ARRAY"G.1800
1070 IF$X="RETURN"G.1900
```

If the symbol is not a keyword then it must be a label, an assignment statement, or a procedure call.

```
1100 REM IDENT STATEMENT
1110 GOS.b
1120 IF?A=CH": "A=A+1;GOS.h;JJN=P;G.s
1130 IF?A=CH"("GOS.h;GOS.u;G.p
1135 GOS.j;IF?A=CH"[ "G.1300
1160 GOS.d;H=V;GOS.m;R.
```

d - Right-hand side of assignment statement.

```
1180dIF?A<>CH"=" P.$6"NO =" ;G.o
1190 A=A+1;GOS.e;GOS.v;L=V;GOS.l;GOS.v;R.
```

BEGIN - Deal with BEGIN ... END block.

```
1200 REM BEGIN...END
1210 A=A-1;DO A=A+1
1220 GOS.s;GOS.b
1230 U. ?A<>CH";"
1240 GOS.x;IF$X="END"R.
1250 P.$6"NO END";G.o
```

Array element on the left-hand side of an assignment statement.

```
1300 REM ARRAY=
1310 A=A+1;GOS.e;IF?A<>CH"]"P.$6"NO ]";G.o
1320 A=A+1;GOS.d;L=V;GOS.v
1325 IFL<=0[LDX @-L;STA V,X;];H=0;R.
1330 [LDX L;STA V,X;];H=0;R.
```

IF - Assemble code to evaluate condition, and following THEN assemble code to execute a statement. Pull label from stack and assemble label. Deal with ELSE clause.

```
1400 REM IF...THEN...ELSE
1410 GOS.q;GOS.b
1420 GOS.x;IF$X="THEN"G.1430
1425 P.$6"NO THEN";G.o
1430 GOS.s;GOS.v;GOS.b
1440 GOS.x;IF$X="ELSE"G.1460
1445 A=A-N;[:LLV;];R.
1460 GOS.g;U=G;GOS.u;[JMP LLG;]
1470 [:LLV;];GOS.s
1490 GOS.v;[:LLV;];R.
```



GOTO - Get label and assemble jump to it.

```
1500 REM GOTO
1510 GOS.k:[JMP U;];R.
```

PROC - Get name and set its value to entry address P. Then get dummy parameter.

```
1700 REM PROC
1710 GOS.k;JJN=P;IF?A<>CH>("P.$6"MISSING BRACKET";G.o
1720 A=A+1;JJI=1;GOS.i;IFN=0G.1780
1730 U=N;GOS.u
1740 T=RR3;H=T;JJU=T;GOS.m
1745 IF?A<>CH)"P.$6"NO BRACKET";G.o
1750 A=A+1;GOS.b;IF?A<>CH";"P.$6"NO ";";G.o
1760 A=A+1;GOS.s
1770 GOS.v;N=V;GOS.v;JJN=V;[RTS;];R.
```

Come here if procedure has no parameter.

```
1780 IF?A<>CH)"P.$6"NO BRACKET";G.o
1782 A=A+1;GOS.b;IF?A<>CH";"P.$6"NO ";";G.o
1785 A=A+1;GOS.s;[RTS;];R.
```

ARRAY - Look up array name; assign space from RRI onwards. Allow multiple declarations, separated by commas.

```
1800 REM ARRAY
1810 A=A-1;DO A=A+1
1820 GOS.b;GOS.x;GOS.y
1830 IFN<>I P.$6"ARRAY DECLARED";G.o
1840 IF?A<>CH["P.$6"BRACKET MISSING";G.o
1850 A=A+1;GOS.c;IFN=0P.$6"CONSTANT MISSING";G.o
1860 GOS.v;JJI=RR1+R;I=I+1;R=R-V+1
1870 IF?A<>CH]"P.$6"BRACKET MISSING";G.o
1880 A=A+1;GOS.b;UNTIL?A<>CH";";R.
```

RETURN - Assemble code to load accumulator with expression.

```
1900 REM RETURN
1910 GOS.e;GOS.v;L=V;GOS.l;H=0;R.
```

i - Read an identifier.

```
2000iREM IDENTIFIER
2010 GOS.x
```

j - Look up identifier \$X in symbol table. If symbol does not already exist (N=I) allocate address for it. Push address to stack.

```
2030jIF N=0 R.
2040 GOS.y
2050 IFN=I;I=I+1;R=R+1;JJN=R+RR1
2070 IFI>30P.$6"TOO MANY VARIABLES";G.o
2080 U=JJN;GOS.u;R.
```

c - Read a decimal constant. If not found, N=0. If found, push minus its value.

```
2100cREM CONSTANT
2105 GOS.b
2110 N=-1;C=0;DO D=C;N=N+1;C=C*10
2120 C=C+A?N<CH"0"
2130 U.A?N<CH"0"ORA?N>CH"9"
2140 IFN=0 R.
2150 A=A+N
2160 U=-D;GOS.u;N=1;R.
```

k - Read label.

```
2200kREM LABELS
2210 GOS.b;GOS.x
2220 IF N=0 P.$6"LABEL MISSING";G.o
```

h - Look up label in symbol table. If not found (N=I) put it in. Return its address in U.

```
2230hGOS.y
2250 IFN=I;I=I+1
2260 IFI>30P.$6"TOO MANY VARIABLES";G.o
2270 U=JJN;R.
```

y - Look up \$X in symbol table, \$II(0), \$II(1) ... If not found, N=I.

```
2400yREM LOOKUP
2410 $III=$X;N=-1
2420 DO N=N+1;U.$IIN=$III;R.
```

e - Assemble code to calculate an expression, of the form:

<factor> <operator> <factor>

where <operator> is one of:

+ : add                   - : subtract  
| : OR                     & : AND  
<< : left shift         >> : right shift

Then push the address of the result on the stack.

```
3000eREM EXPRESSION
3010 GOS.b;GOS.f
3015 GOS.b
3020 IF?A=CH"+"OR?A=CH"-OR?A=CH"&" OR?A=CH"|"O=?A;
A=A+1;G.3035
3025 IF((?A=CH">"A.A?1=CH">))OR (?A=CH"<"A.
A?1=CH"<")):1 R.
3030 O=?A;A=A+2
3035 U=O;GOS.u
3040 GOS.f;GOS.v;U=V;GOS.v;O=V;GOS.v;L=V;GOS.l
3045 IF U<=0 G.3070
3050 IFO=CH"+"[CLC;ADC U;]
3055 IFO=CH"-"[SEC;SBC U;]
3060 IFO=CH"|" [ORA U;]
3065 IFO=CH"&"[AND U;]
3068 G.3190
3070 IFO=CH"+"[CLC;ADC @-U;]
3075 IFO=CH"-"[SEC;SBC @-U;]
3080 IFO=CH"|" [ORA @-U;]
3085 IFO=CH"&"[AND @-U;]
3160 IFO=CH"<"F.N=1TO-U;[ASL A;];N.
3180 IFO=CH">"F.N=1TO-U;[LSR A;];N.
3190 L=U;GOS.r;GOS.t
3195 G.3015
```

p - Procedure body. Check for ')'. If there is a parameter first assemble code to calculate parameter, load it into the accumulator, and then JSR. Assume subroutine alters accumulator, so set H=0.

```
3200pREM PROC BODY
3210 IFA?1=CH")"A=A+2;G.3230
3220 GOS.f;GOS.v;L=V;GOS.l
3230 GOS.v;[JSR V;];H=0;R.
```

f - Factor. Check for symbol, constant, or bracketed expression. . If the symbol is followed by '(' or '[' then it is a function or an array element respectively.

```
3600fREM FACTOR
```

```

3610 GOS.x;IFN=0G.3630
3615 IF?A=CH("G.3690
3620 GOS.j;IF?A=CH["G.3670
3625 R.
3630 GOS.c;IF N R.
3635 IF?A<>CH(" P.$6"BRACKET MISSING";G.o
3640 A=A+1;GOS.e;GOS.b
3650 IF?A<>CH)" P.$6"BRACKET MISSING";G.o
3660 A=A+1;R.

```

Evaluate array element. Assemble code to evaluate array index and load it into the accumulator; then TAX and load indexed by the base address.

```

3670 REM ARRAYS
3675 A=A+1;GOS.e;IF?A<>CH]"P.$6"NO BRACKET";G.o
3680 A=A+1;GOS.v;L=V;GOS.l;GOS.v
3685 [TAX;LDA V,X;];G.t

```

Call function here.

```

3690 GOS.h;GOS.u;GOS.p;G.t

```

q - Logical expression. Look for:  
<expression> <comparison> <expression>

```

4000qREM LOGICAL
4010 GOS.b;GOS.e;GOS.b

```

Expect a comparison here; look for '<', '>', and '=' and set value of U depending on sequence:

```

> : 1           = : 2
>= : 3          < : 4
<> : 5           <= : 6

```

Then use a computed GOTO to assemble code for each case.

```

4020 U=0
4030 IF?A=CH"<"A=A+1;U=4
4040 IF?A=CH">"A=A+1;U=U+1
4050 IF?A=CH"="A=A+1;U=U+2
4060 IFU=0 OR U>6 P.$6"ILLEGAL TEST";G.o
4070 GOS.u;GOS.e
4080 GOS.v;M=V;GOS.v;U=V
4090 GOS.v;L=V;GOS.l
4100 IFM>0[ CMP M;]
4110 IFM<=0[ CMP @-M;]

```

First generate a label LLG. Then assemble code for the comparison. Note that if the condition is true we branch around a jump to LLG. Push value of LLG for use by IF...THEN statement.

```
4120 L=M;GOS.r;GOS.g;G.(4200+U)
4201 [BEQ P+4;BCS P+5;];G.z
4202 [BEQ P+5;];G.z
4203 [BCS P+5;];G.z
4204 [BCC P+5;];G.z
4205 [BNE P+5;];G.z
4206 [BCC P+7;BEQ P+5;];G.z
4210z[JMP LLG;]
4220 U=G;GOS.u;H=0;R.
```

u - Push U onto stack.

```
5020uSSS=U;S=S+1;IFS<21 R.
5021 P.$6"STACK FULL";G.o
```

v - Pull V from stack.

```
5030vS=S-1;IFS>=0V=SSS; R.
5031 P.$6"STACK ERROR";G.o
```

b - Skip blanks, line numbers, and comments between '{' and '}'.

```
5040bIF?A=32 DO A=A+1;U.?A<>32
5041 IF?A=13A=A+3;P.$A';G.b
5042 IF?A=CH{"DOA=A+1;U.?A=CH"}";A=A+1;G.b
5043 R.
```

g - Generate a new label number in G. Label is LLG.

```
5070gG=G+1;IF G<20 R.
5072 P.$6"TOO MANY LABELS";G.o
```

t - Generate a temporary location TTN; return its address in T, set H to the address, and push the address.

```
5100tREM TEMP. LOC.
5110 N=-1;DO N=N+1; IF N>20P.$6"NOT ENOUGH TEMP";G.o
5120 U.TTN=0
5130 T=N+RR2; TTN=T; U=T; H=T; G.u
```

x - Read a symbol into \$X from \$A. Returns N=0 if no symbol found.

```

6000xREM READ SYMBOL
6010 GOS.b;N=-1;DO N=N+1; N?X=A?N
6020 U.A?N>CH"Z"ORA?N<CH"A"ORN=7
6030 IF N=0 R.
6040 IF N<7 N?X=#D;A=A+N;R.
6050 P.$6"SYMBOL TOO LONG";G.o

```

l - Assemble code to load the accumulator with L. If accumulator already contains L (L=H) then do nothing; otherwise store its previous contents (GOS.m) and load new contents.

```

7000lREM LOAD ACCUMULATOR
7010 IFL=H AND L>0 G.r
7020 GOS.m
7030 IFL<=0 [LDA @-L;];R.
7040 [LDA L;];G.r

```

m - Assemble code to store accumulator's contents to location H.

```

7100mREM STORE ACCUMULATOR
7200 IFH>0[STA H;];H=0
7210 R.

```

r - Release temporary variable with address L for re-use.

```

7300rREM RELEASE VARIABLE
7310 IF L>=RR2 AND L<RR3;TT(L-RR2)=0
7320 R.

```

o - Output error. Print line containing error and '^' pointing to approximate position.

```

9000oREM ERROR
9010 N=A;X=0;DO N=N-1;X=X+1;U.?N=13;@=5
9020 P.'N?1*256+N?2,$N+3'
9030 F.N=0TOX+1;P." ";N.;P."^";E.

```

Variables:

- Pointer to current position in expression being compiled
- Used to evaluate constant
- Number of next free label LLG
- Address whose contents are currently in accumulator. H=0 means ignore previous contents
- Number of next free symbol
- (0)..II(30) - Pointers to symbol names
- (0)..JJ(30) - Addresses of symbols
- Value or address to be loaded into accumulator; used by subroutine l

LL(0)..LL(20) - Labels for use in assembly  
N - Temporary variable  
O - Operator read by subroutine e  
P - Program location counter, used by assembler  
RR(0)..RR(2) - Constant addresses  
R - Number of variable locations used up  
S - Next free location on SS stack  
SS(0)..SS(20) - Stack used by compiler  
T - Temporary location assigned by subroutine t  
TT(0)..TT(20) - Flags for temporary locations; value=0  
if location is free for use  
U - Value to be pushed by subroutine u  
V - Value pulled by subroutine v  
X - String into which symbols and keywords are read by  
subroutine x

### Further Suggestions

The compiler could usefully be extended in two directions. Firstly, the definition of SPL could be enlarged to include some or all of the REPEAT..UNTIL, WHILE..DO, FOR..DO, and CASE statements of Pascal, AND and OR connectives in the IF statement, and multi-parameter procedures. Secondly, the compiler could be enlarged to deal with other data types, such as character strings and two-byte integers. Multi-byte operations, including multiply and divide, could then be implemented by compiling calls to routines which would be included in the machine code generated by the compiler.

Alternatively, the compiler could be extended into a special-purpose language, for applications such as machine control, by adding extra statements for reading and setting bits on the computer's input and output ports, and for setting up interrupt-service routines.

The compiler can also be modified to generate machine code for other processors, such as the 6809. To do this, each assembler statement in the compiler should be replaced by an equivalent BASIC statement that will store the relevant machine code into memory. For example, line 3050 in the BBC Computer version of the Compiler program:

```
3050 IFO=ASC"+"[CLC:ADC U:]
```

would be altered to:

```
3050 IFO=ASC"+": ?P%=&B9: P%?1=U/256: P%?2=U AND &FF:  
P%=P%+3
```

where &B9 is the code for the 'ADC A' instruction on the 6809. The Compiler program could thus be used for developing software on other processors without the

need for an assembler.



# Bibliography

Readers interested in further information on some of the programs in this volume may find the following list of references useful:

## Silver Dollar Game

"On Numbers and Games," by J. H. Conway, p. 123, Academic Press, 1976.

## Surface

"Atomic Theory and Practice," by David Johnson-Davies, p. 166, Acorn Computer Ltd., 1979.

## Anagrams

"Generation of Permutations in Lexicographical Order," Shen, M. K., Alg 202, CACM, 6:9, p. 517, 1963.

## Polynomials

"Seminumerical Algorithms, The Art of Computer Programming, Vol. 2," by D. Knuth, p. 360 ff., Addison Wesley, 1969.

## Compiler

"An Introduction to Compiler Writing," by J. S. Rohl, Macdonald & Jane's, 1975.

"Algorithms in Snobol 4," James F. Gimpel, Ch. 18, John Wiley and Sons, 1976.

## General

"The BBC Microcomputer User Guide," by John Coll, BBC Publications, 1981.



# A message from the publisher

---

Sigma Technical Press is a rapidly expanding British publisher. We work closely in conjunction with John Wiley & Sons Ltd. who provide excellent marketing and distribution facilities.

Would you like to join the winning team that published this and the other highly successful books listed on the back cover? Specifically, **could you write a book that would be of interest to the new, mass computer market?**

Our most successful books are linked to particular computers, and we intend to pursue this policy. We see an immense market for books relating to such machines as:

**The BBC Computer**  
**PET**  
**Apple**  
**Tandy**  
**Sinclair**  
**Osborne**  
**Atari**  
**IBM**  
**Sirius . . . and many others**

If you think you can write a book around one of these or any other popular computer — or on more general themes — we would like to hear from you.

Please write to: Graham Beech,  
Sigma Technical Press,  
5 Alton Road,  
Wilmslow,  
Cheshire, SK9 5DY,  
United Kingdom.

Or, telephone 0625-531035.



## Errata List

### to 'Practical Programs for the BBC Computer and Acorn Atom'

---

---

Some minor errors have been detected in the program listings, as follows:

- p. 9 Line 25 — delete 'P = R:'
- p. 30 To run the 'Surface' program on the BBC computer Model A, type:  
PAGE = &900  
before entering the program.
- p. 37 Line 240 — change 'DO' to 'REPEAT'  
Line 410 — PRINT MID\$(A\$, CC%(K), 1);
- p.100 Remove line 70.
- p.101 Line 1310 — ';' should be ':'  
Line 1325 — '@' should be '#'

If you have any suggestions regarding the programs in this book, please contact the publisher.

## About This Book

The BBC Computer is a powerful and versatile machine that has been developed as part of the BBC computer literacy project. It has numerous hardware options and a range of high quality graphic modes. These features, together with an extended, structured BASIC make it a very attractive machine for home, business or school use.

The programs in this book illustrate many of the features of the BBC Computer and its close relative, the Acorn Atom. They include games, language manipulation, mathematics, and sophisticated graphics. An example of the latter was used for the cover design of this book. Users of this book are encouraged to understand how the programs work, so each program is explained in great detail.

The programs are listed in both BBC Computer and Acorn Atom formats; users of other computers will be able to adapt most of them quite easily.

David Johnson-Davies is Managing Director of Acornsoft Ltd, a firm dedicated to Acorn applications software. David is one of the original designers of the Acorn Atom and is the author of the book "Atomic Theory and Practice".

### About Our Other Books

#### SOFTWARE SECRETS

*Input, output and data storage techniques, by G. Beech*

£5.95

#### LIVING WITH THE MICRO

*by M. Banks*

£4.50

#### SUCCESSFUL SOFTWARE FOR SMALL COMPUTERS

*by G. Beech*

£5.95

#### COMPUTER PROGRAMS THAT WORK

*by J.D. Lee, (3rd Edn) G. Beech and T.D. Lee*

£4.95

#### BYTEING DEEPER INTO YOUR ZX81

*by M. Harrison*

£4.95

#### PRACTICAL PASCAL FOR MICROCOMPUTERS

*by R. Graham (January 1982)*

£6.50

#### SHARP SOFTWARE TECHNIQUES

*Programming the MZ-80K by D. Trowsdale (February 1982)*

£5.95

#### THE BROADWATER ECONOMICS SIMULATIONS

*(Software package) by G. Addis (January 1982)*

£25.00

#### CP/M- THE SOFTWARE BUS

*by A. Clarke & D. Powys-Lybbe (April 1982)*

£8.50

#### Z80 INSTANT PROGRAMS - SECOND REVISED EDITION

*by J. Hopton (February 1982)*

£7.50

*All prices are correct at December 1981.*