THE AMAZING AMSTRAD OMNIBUS

CPC 464
CPC 664
CPC 6128

Martin Fairbanks

The ultimate programming resource for your Amstrad computer!

ARTIFICIAL INTELLIGENCE; creating and playing ADVENTURES; a huge collection of stunning, pre-packaged GRAPHICS to include in your own programs; complete LOGO implementation and tutorial; ENTERTAINMENT and EXPLORATION (games and more); the Amstrad DISC COMPANION; and much, much more! Mountains of ideas for you and your Amstrad to explore in the coming months.

The Amazing Amstrad Omnibus

Interface Publications
LONDON · MELBOURNE

The Amazing Amstrad Omnibus

Martin Fairbanks

First published in the UK by:
Interface Publications,
9–11 Kensington High Street,
London W8 5NP.

Copyright © Tim Hartnell, 1985
First printing September 1985

ISBN 0 907563 82 1

The programs in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. While every care has been taken, the publishers cannot be held responsible for any running mistakes which may occur.

ALL RIGHTS RESERVED

No use whatsoever may be made of the contents of this volume – programs and/or text – except for private study by the purchaser of this volume, without the prior written permission of the copyright holder.
Reproduction in any form or for any purpose is forbidden.

Books published by Interface Publications are distributed in the UK by WHS Distributors, St. John's House East Street, Leicester LE1 6NE (0533 551196) and in Australia and New Zealand by PITMAN PUBLISHING. Any queries regarding the contents of this volume should be directed by mail to Interface Publications, 9–11 Kensington High Street, London W8 5NP.

Printed and Bound by Short Run Press Ltd, Exeter

Contents

Section One — The Amstrad Graphics Companion

Graphic Demonstrations 1
Zebra Windmills, Italic Alphabet/Numbers, Computer Script, 3D Printer Plotter, Lissajoux Figures, Meta-Sphere, Dionysian Spiral, Towering Walls, Warp Factor V, Frequency Waves
User-Defined Graphics Library 18
Space Ships, Alien Invaders, Instruments of War, Little People, Fruit, Aeroplanes, The Zoo, 'Pacman', Curved Corners, Timber!, Alternate Card Suits, Rolling Ball, Fractions, Board Games Pieces, King of the Road, Assorted Shapes and Designs

Section Two — The Amstrad Disc Companion

Getting Started with Discs 30

File Commands 31

Formatting Blank Discs 32

Backing up the System Disc 33

Copying Programs 34

CP/M 35

BIOS Jumpblocks 39

External Commands 40

Amsdos Error Messages 41

Further Reading on CP/M 42

Section Three — Exploring Artificial Intelligence

Part One — Learning and Reasoning 43
The AI Debate, Feedback, How Do Machines Think?

Part Two — A Program which Learns 48
TICTAC Board, Michie and Boxes, Samuel and the Checkerboard, How it Works, Sample Runs, Listing

Part Three — A Program which Reasons 59
Syllogisms, Reaching Conclusions, Internal Storage, How it Works, Listing

Part Four — Search Trees and Snickers 70
Why a Tree?, 'Parallel Processing', Vanishing Acts, Digging Deep, Mini-Maxing, Move Hierachy, Weighted Elements, Alpha-Beta Algorithm, How it Works, Listing

Part Five — The Wider Value of Games 94
Claude Shannon, Real-World Complexities, Other Games and Lessons, Brute Force

Part Six — Understanding Natural Language 98
Shrdlu, Language Parsing, Blockworld Output, Problems, Syntax and Semantics

Part Seven — Blockworld 105
Sample Run, How it Works, Program Modules, Listing, Improving the Program, Shrdlu in Action

Part Eight — The Doctor is In 128
Eliza Sample Run, Weizenbaum, The Russian Connection, Short Sharp Shocks, Listing

Part Nine — Machine Translation 139
Still a Use for People, SYSTRAN in Action, Franglais, The TRANSLATE Program

Part Ten — Hanshan 146
Sample Poems, Listing

Part Eleven — Expert Systems 150
Areas of Application, 'Mechanised' Reasoning, Limitations, Chemical Structures and Dendral

Part Twelve — The Little Spurt 155
Medici, Sample Run, Listing, Little X-Spurt, Mineral Identification, Listing, Choosing a Chip, Listing

Part Thirteen — Self-Learning Systems 175
Self-Learn in Action, How it Works, Listing, Multi-Self-Learn, No Need for Correction, Listing

Section Four — Logo-K

What is Logo? 191

The Program 192

Sample Pictures 199

Primitives .....
Forward, back, Right, Left

Using the Pen 204

Going Home 205

Turning About/Clearscreen 206

Shaping Up 207
Square, Triangle, Rotating Triangles, Rotating Squares, Fundamental Shapes

Repeating Yourself 212

Defining Procedures 215

More Fun and Games 218

Moving About 220

Other Commands 222

Additional Primitives 223

Uncharted Lands 225

Further Reading 230

Section Five — Entertainment

Dominoes 231

Oliver Rand 241

Celestia 245

CPC-Thello 248

Squisher and Squasher 252

Creating Your Own Games 256
Chess, Maharajahs, Holding the Board, Dice Games, Backgammon, Cards

Section Six — Creating Adventures

History of Adventure Games 262

Give Me My Bow 266

The Scenario/Sample Run 268

Creating the Floor Plan 274
Mapping the Environment, Building a Travel Table, Moving About, The Player's Location, Moving Around the Map, Consistency and Reality

Creating the Structure 280
Master Loop, Modular Construction, The Maps

Treasure and Terror 283
Distributing the Treasure, Distributing the Monsters, Adding Variables

The Final Reckoning 286

Maintaining the Status Quo 287
Let There Be Light, Objects in Disarray, A Limited Vocabulary, Fight or Flight, Other Tongues

Making Magic 291
Picking Up Treasure, The Quartermaster's Store

The Big Fight 293
Be Prepared, Taking Up Arms, Tumult Dies

Room Descriptions 294
Special Handling

Listing: WEREWOLVES AND WANDERER 295

The Satan Blinder 307
Puzzles, Two-Word Sentences

Vocabulary Handling 308
Go, Fight, Help, Quit

Maps 311

Listing: THE SATAN BLINDER 312

Adding Excitement 341
Map-Making, Structured Programming, Purpose, Player Character, Magic, Research, Time Limits, Climate

Further Reading 346

Section Seven — Structured Programming

Flow Charts 349

Programming in Modules 350

Explicit Prompts 352

REM Statements, Variables 353

Checking Input, Documentation 354

Section Eight — Utility and Practical Programs

Business Programs 356

MINICALC 357

Listing 359

Sample Run 363

Personal Finance 365

Percentage Change, Depreciation 370

Mortgage Repayments 371

Section Nine — Crackers and Corkers

Disenable BREAK Key 373

Large Writing 373

Sample Loading Screen 374

Score Frame 374

D.I.Y. Teletext 375

Generating Sound 382
Parameters, Frequency Table

Sound Effects 385
Outburst, Alien Attack, Musical Siren, Alert!, Motor Boat, Explosion

The Phantom Composer 386

Appendices:

A — BASIC Keywords 390

B — Inks and Colours 391

C — 464 ROM Routines 393
Memory Map, Main Firmware Jumpblock, Disc BIOS Jumpblocks, External Disc Commands

D — Coping with Error Messages 413

E — Key Numbering 414

F — User-Defined Graphics Forms 415

Section One
The Amstrad Graphics Companion

Your computer is equipped with sophisticated graphics capabilities, which exceed — to a significant extent — most other personal computers now on the market. You can do some pretty exciting things with this graphics capability, as we will demonstrate convincingly in this section of the book.

As well as dramatic swirling 'three-dimensional' designs, we'll look at multi-coloured Lissajous figures, and a number of sine wave combinations. A completely new character set, which renders all the letters (upper and lower case), as well as the numbers, in italics is provided. As well, character definitions are given to turn the numbers into 'computer script', similar to those which are on the bottom of cheques.

I'm sure you've made use of the Amstrad's SYMBOL function, which redefines any characters you choose. Our 'U.D.G. Library' gives you well over 100 sets of data for everything from pacmen through asteroid ships and dancing aliens to male and female symbols, bombs, tanks, barrels and ladders. You're sure to have a lot of fun experimenting with these sets of data, modifying them as you wish, and incorporating them into your own programs.

Zebra Windmills

We'll start this section of the book with a great demonstration program, which is worth having on cassette or disc just to show friends when they want to see your computer in action. Written by Steven Loze, *Zebra Windmills* uses the wide Amstrad pallette to create whirling, spinning shapes.

```
10 REM ZEBRA WINDMILLS
20 REM BY STEVEN LOZE
30 REM ---------------
40 RANDOMIZE TIME
```

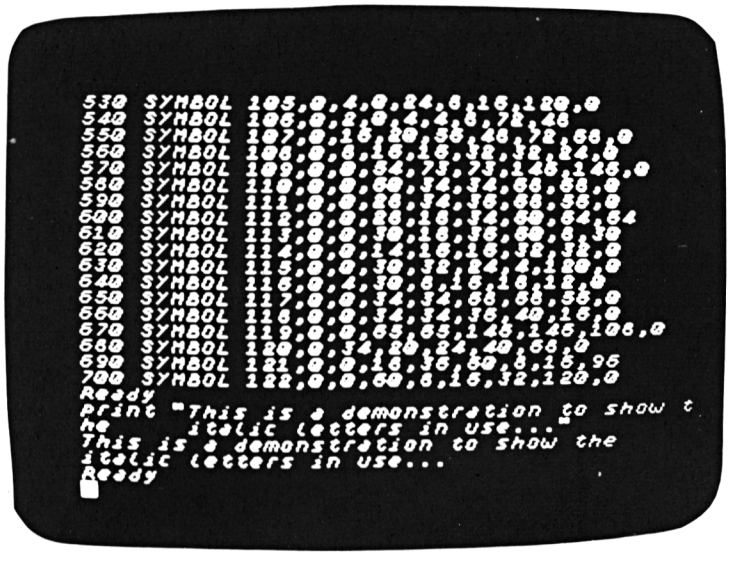
```
50 GOSUB 280:REM INITIALISE
60 WHILE NOT finished
70 FOR CHANGE=1 TO 2
80 RAD:IF CHANGE=2 THEN DEG
90 CLS
100 FOR AMSTRAD=1 TO 360 STEP 10*RND
110 PLOT 320,200
120 DRAW 320+250*COS(AMSTRAD),200+190*SI
N(AMSTRAD),((AMSTRAD MOD 3)+1)
130 NEXT AMSTRAD
140 GOSUB 170:REM PEN CHANGES
150 NEXT CHANGE
160 WEND
170 REM PEN CHANGES
180 FOR J=1 TO 26
190 FOR K=1 TO 3
200 INK K,J
210 FOR Z=1 TO 100:NEXT Z
220 INK K,0
230 NEXT K
240 NEXT J
250 GOSUB 280
260 RETURN
270 REM ----------
280 REM INITIALISE
290 MODE 1
300 INK 0,0:INK 1,26:INK 2,24:INK 3,6:BO
RDER 0
310 RETURN
```

Italic Alphabet

Here's a complete set of numbers and upper and lower case letters, so your Amstrad output can be in italics. If you just want italic numbers, use the following in your program:

```
10 REM ITALIC NUMBERS
20 SYMBOL AFTER 48
30 SYMBOL 48,0,60,70,74,148,164,120,0
40 SYMBOL 49,0,48,80,16,32,32,248,0
50 SYMBOL 50,0,28,34,4,56,64,124,0
60 SYMBOL 51,0,30,4,24,4,72,56,0
70 SYMBOL 52,0,6,10,20,36,126,8,0
```

```
80 SYMBOL 53,0,30,16,60,2,68,56,0
90 SYMBOL 54,0,30,32,60,66,68,56,0
100 SYMBOL 55,0,30,2,4,8,16,32,0
110 SYMBOL 56,0,28,36,56,68,68,56,0
120 SYMBOL 57,0,28,34,34,28,4,56,0
```



If you want all the letters redefined, as well as the numbers, use this:

```
10 REM ITALIC NUMBERS
20 SYMBOL AFTER 48
30 SYMBOL 48,0,60,70,74,148,164,120,0
40 SYMBOL 49,0,48,80,16,32,32,248,0
50 SYMBOL 50,0,28,34,4,56,64,124,0
60 SYMBOL 51,0,30,4,24,4,72,56,0
70 SYMBOL 52,0,6,10,20,36,126,8,0
80 SYMBOL 53,0,30,16,60,2,68,56,0
90 SYMBOL 54,0,30,32,60,66,68,56,0
100 SYMBOL 55,0,30,2,4,8,16,32,0
110 SYMBOL 56,0,28,36,56,68,68,56,0
120 SYMBOL 57,0,28,34,34,28,4,56,0
130 REM
140 REM ITALIC CAPITAL LETTERS
150 REM
```

```
160 SYMBOL 65,0,12,18,34,62,66,66,0
170 SYMBOL 66,0,28,18,60,34,66,124,0
180 SYMBOL 67,0,28,34,32,64,68,56,0
190 SYMBOL 68,0,24,20,34,34,68,120,0
200 SYMBOL 69,0,30,16,60,32,64,120,0
210 SYMBOL 70,0,30,16,60,32,64,64,0
220 SYMBOL 71,0,28,34,32,76,68,56,0
230 SYMBOL 72,0,18,18,60,36,72,72,0
240 SYMBOL 73,0,62,8,16,16,32,248,0
250 SYMBOL 74,0,2,2,4,68,72,56,0
260 SYMBOL 75,0,18,20,56,40,68,66,0
270 SYMBOL 76,0,16,16,32,32,64,124,0
280 SYMBOL 77,0,34,54,42,66,68,68,0
290 SYMBOL 78,0,18,26,42,44,68,68,0
300 SYMBOL 79,0,28,34,34,68,68,56,0
310 SYMBOL 80,0,28,18,34,60,64,64,0
320 SYMBOL 81,0,60,66,66,164,148,120,0
330 SYMBOL 82,0,28,18,34,60,68,66,0
340 SYMBOL 83,0,28,34,24,4,68,56,0
350 SYMBOL 84,0,62,8,8,16,16,32,0
360 SYMBOL 85,0,17,34,34,68,68,56,0
370 SYMBOL 86,0,34,34,36,36,40,16,0
380 SYMBOL 87,0,33,33,66,66,90,36,0
390 SYMBOL 88,0,34,20,24,56,68,130,0
400 SYMBOL 89,0,34,20,8,16,32,64,0
410 SYMBOL 90,0,62,4,8,16,32,124,0
420 REM
430 REM ITALIC SMALL LETTERS
440 REM
450 SYMBOL 97,0,0,12,2,60,68,56,0
460 SYMBOL 98,0,16,16,60,32,66,124,0
470 SYMBOL 99,0,0,28,32,32,64,56,0
480 SYMBOL 100,0,2,2,28,36,68,56,0
490 SYMBOL 101,0,0,28,34,124,64,56,0
500 SYMBOL 102,0,6,8,12,16,16,32,0
510 SYMBOL 103,0,0,14,18,34,60,4,120
520 SYMBOL 104,0,16,16,62,34,68,68,0
530 SYMBOL 105,0,4,0,24,8,16,120,0
540 SYMBOL 106,0,2,0,4,4,8,72,48
550 SYMBOL 107,0,16,20,56,48,72,68,0
560 SYMBOL 108,0,8,16,16,32,32,24,0
570 SYMBOL 109,0,0,54,73,73,146,146,0
580 SYMBOL 110,0,0,60,34,34,68,68,0
```

```
590 SYMBOL 111,0,0,28,34,36,68,56,0
600 SYMBOL 112,0,0,28,18,34,60,64,64
610 SYMBOL 113,0,0,30,18,36,60,8,30
620 SYMBOL 114,0,0,14,16,16,32,32,0
630 SYMBOL 115,0,0,30,32,24,4,120,0
640 SYMBOL 116,0,4,30,8,16,18,12,0
650 SYMBOL 117,0,0,34,34,68,68,56,0
660 SYMBOL 118,0,0,34,34,36,40,16,0
670 SYMBOL 119,0,0,65,65,146,146,108,0
680 SYMBOL 120,0,34,20,24,40,68,0
690 SYMBOL 121,0,0,18,36,60,8,16,96
700 SYMBOL 122,0,0,60,8,16,32,120,0
```

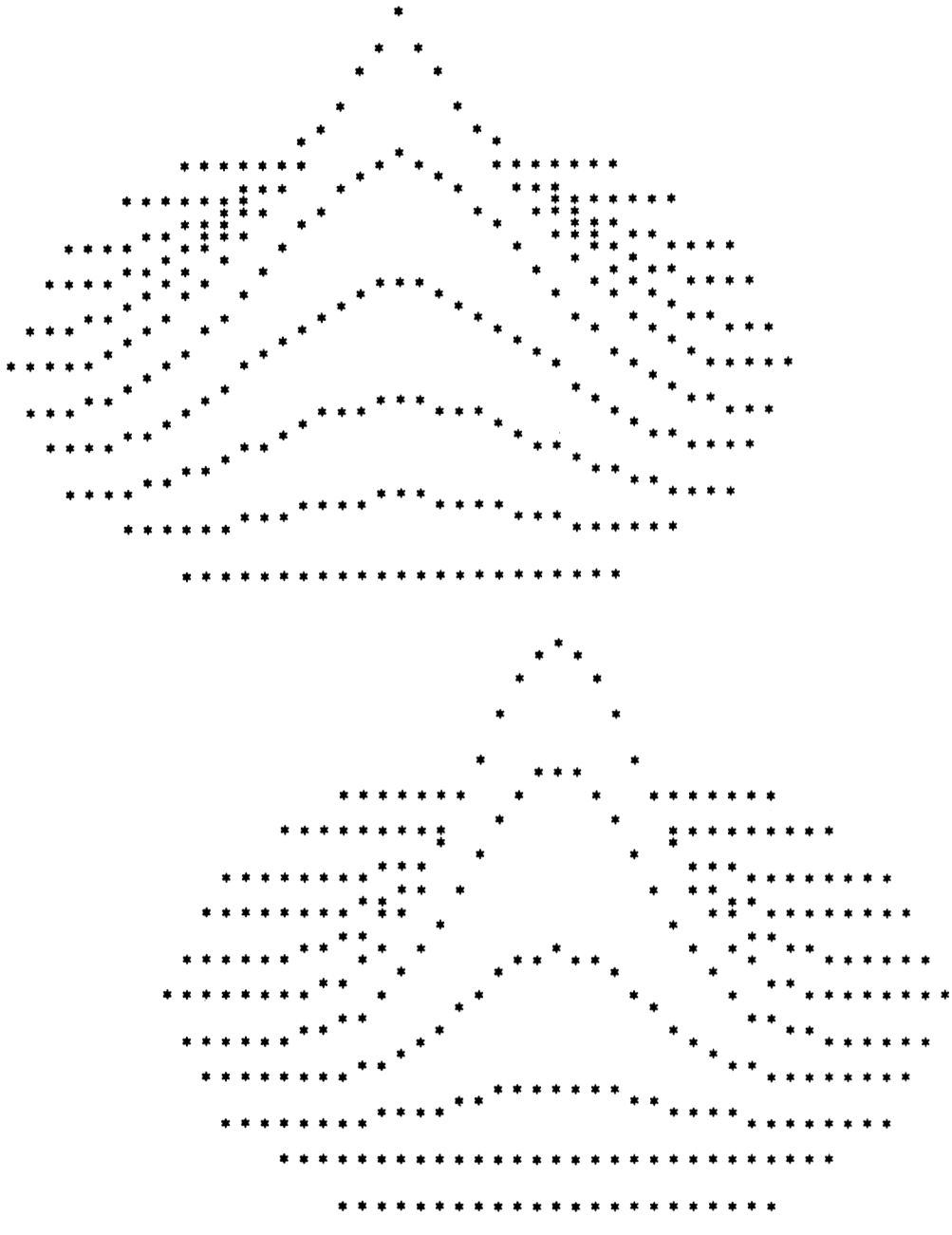
Computer Script

You can use this next set of data to convert the numbers into 'computer script', so they resemble the machine-readable numbers on cheques.

```
10 REM 'COMPUTER-LIKE' NUMBERS
20 SYMBOL AFTER 48
30 SYMBOL 48,0,60,66,66,98,98,98,60
40 SYMBOL 49,0,8,8,8,12,12,12,12
50 SYMBOL 50,0,124,2,2,60,96,96,62
60 SYMBOL 51,0,124,2,2,60,6,6,126
70 SYMBOL 52,0,64,96,100,100,126,4,4
80 SYMBOL 53,0,62,64,64,60,6,6,126
90 SYMBOL 54,0,62,64,64,124,70,70,126
100 SYMBOL 55,0,120,8,8,12,12,12,12
110 SYMBOL 56,0,60,66,66,60,70,70,60
120 SYMBOL 57,0,60,66,66,60,6,6,126
```

3D Printer Plotter

This next routine will produce, on a printer, 'three-dimensional' images from equations. There are four sample equations given in the lines 200 through to 230. Once you've seen these in action, you can try substituting some of your own. Here's what the sample ones look like:



And here's the listing, so you can see it in action on your own printer:

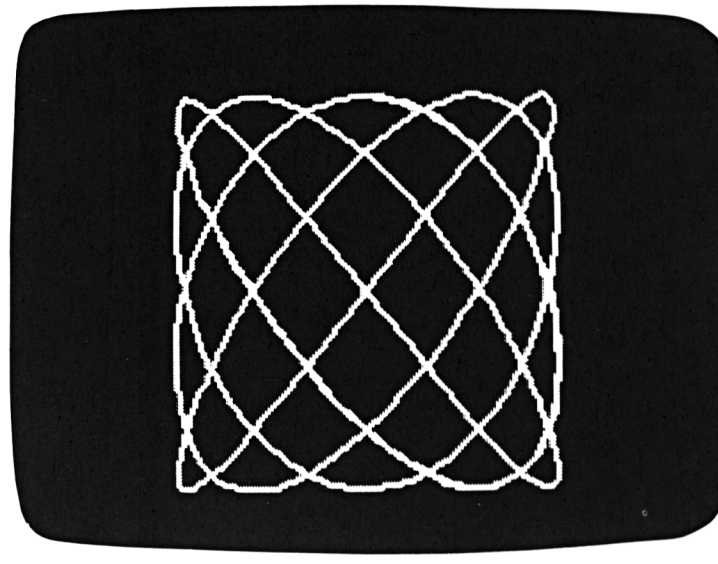
```
10 REM 3D PRINTER PLOTTER
20 MODE 2
30 FOR DEMO=1 TO 4:GOSUB 180
40 FOR CPC=-BIT TO BIT STEP BYTE
50 ALAN=0
60 SUGAR=VARS*INT(SQR(900-CPC*CPC)/VARS)
70 FOR TIM=SUGAR TO -SUGAR STEP -VARS
80 STEVE=INT(VARS*VARS+FNA(SQR(CPC*CPC+T
IM*TIM))-CHIP*TIM)
90 IF STEVE<=ALAN THEN 120
100 ALAN=STEVE
110 PRINT #8,TAB(STEVE);"*";
120 NEXT TIM
130 PRINT #8
140 NEXT CPC
150 NEXT DEMO
160 END
170 REM ------------------------
180 PRINT #8:PRINT #8
190 VARS=5:BYTE=1.5:BIT=6*VARS:CHIP=0.7
200 IF DEMO=1 THEN DEF FNA(Q)=BIT*EXP(-Q
*Q/100)
210 IF DEMO=2 THEN DEF FNA(Q)=BIT*EXP(-C
OS(Q/16))-BIT
220 IF DEMO=3 THEN DEF FNA(Q)=BIT-BIT*SI
N(Q/18)
230 IF DEMO=4 THEN DEF FNA(Q)=SQR(BIT*BI
T+BYTE/100-Q*Q)
240 RETURN
```

Lissajoux Figures

Jules Antoin Lissajous, a French physicist who lived from 1822 to 1880, made a study of the movement of particles under the action of periodic motion, acting at right angles to each other. He discovered that bodies moving in this way trace intricate patterns as they dance around each other.

The plotted point in the next two programs in this section traces out the

path of the sum of these period motions. The figure can be used in a number of ways. One way is to compare two frequencies. If they are the



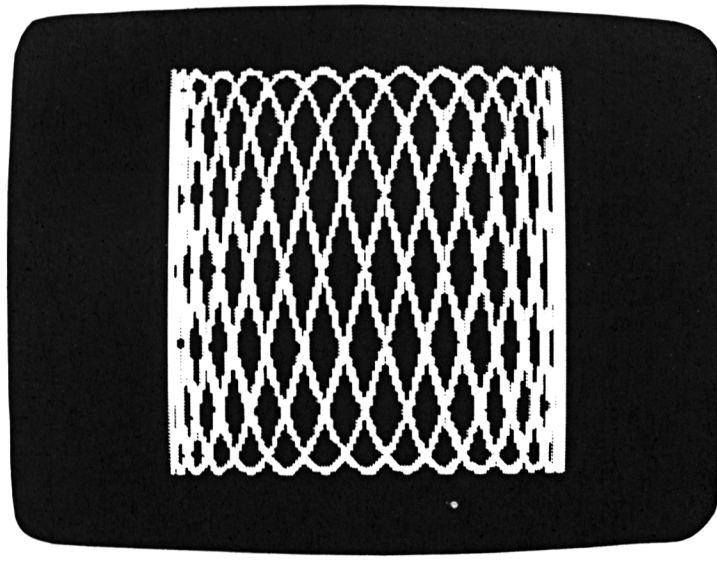
same, the program will draw a circle. If they are different, the number of points where the curve touches either the vertical or horizontal edge is the ratio of the two frequencies.

There are two versions of the program. In this first one, the frequencies are generated at random (lines 120 and 130), starting a new figure each time you press a key:

```
10 REM RANDOM LISSAJOUX FIGURES
20 REM
30 REM CREATES RANDOM LISSAJOUX
40 REM  FIGURES. PRESS ANY KEY
50 REM      FOR A NEW DESIGN
60 REM
70 RANDOMIZE TIME
80 MODE 1:INK 0,0:BORDER 0:INK 1,2:INK 2
,6:INK 3,24
90 IF INKEY$<>"" THEN 90
100 REM --------------------
110 WHILE NOT FINISHED
120 P=INT(RND*3)+1
```

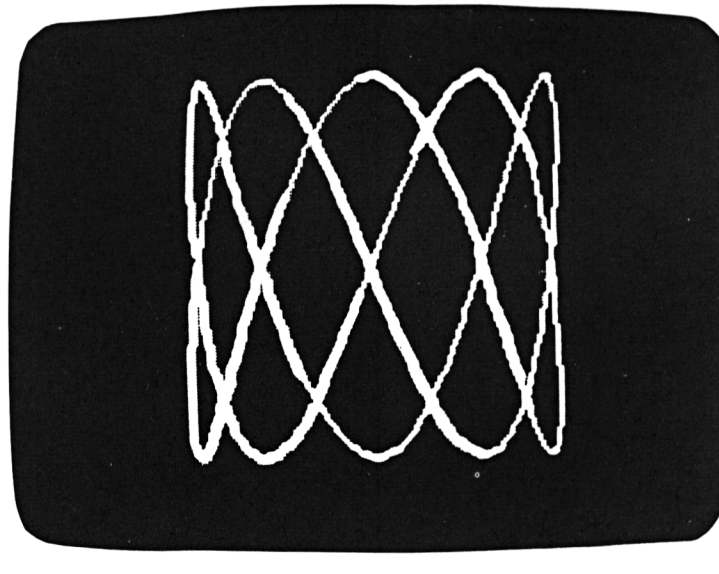
```
130 LOZ=(INT(RND*3)+1)/0.8
140 GF=0
150 PLOT 300+120*SIN(GF),200+120*COS(GF*
LOZ),P
160 REM --------------------
170 WHILE INKEY$=""
180 GF=GF+0.1
190 DRAW 300+120*SIN(GF),200+120*COS(GF*
LOZ),P
200 IF RND>0.9 THEN P=P+1:IF P=4 THEN P=
1
210 WEND
220 CLS
230 WEND
```

In the second version, you'll see colours chosen from a palette of thirteen (see the DATA in line 280) are used to create 'solid' Lissajoux figures:



```
10 REM 'SOLID' LISSAJOUX
20 REM
30 DIM PP(13)
40 RANDOMIZE TIME
50 WHILE NOT FINISHED
60 RESTORE:FOR J=1 TO 13:READ PP(J):INK
J,PP(J):NEXT J
70 MODE 0:INK 0,0:BORDER 0
```

```
80 IF INKEY$<>"" THEN 80
90 REM --------------------
100 P=1
110 LOZ=(INT(RND*3)+1)/0.8
120 GF=0
130 PLOT 300+140*SIN(GF),200+140*COS(GF*
LOZ),P
140 COUNT=0
150 REM --------------------
160 WHILE GF<6.2
170 GF=GF+0.1
180 COUNT=COUNT+1
190 DRAW 300+120*SIN(GF),200+120*COS(GF*
LOZ),P
200 DRAW 200+50*COS(GF),200+137.5*SIN(GF
)
210 IF 10*(INT(COUNT)/10)=COUNT THEN P=P
+1:IF P=14 THEN P=1
220 WEND
230 FOR J=1 TO 13
240 FOR K=13 TO 1 STEP -1
250 INK K,PP(J):FOR Q=1 TO 30:NEXT Q
260 NEXT:NEXT
270 WEND
280 DATA 24,2,6,8,18,25,1,3,15,17,5,4,3
```



Shadow-Blasters

Shadow-Blasters are highly effective demonstration programs, which show just how much your Amstrad can do when it is given a chance.

The first one is *Meta-Sphere:*

```
10 REM META-SPHERE
20 RANDOMIZE TIME
30 MODE 2:INK 0,0:INK 1,26:BORDER 0
40 TRY%=150
50 ORIGIN 320,200
60 FOR X=0 TO 130 STEP 0.4
70 DRAW TRY%*SIN(X),TRY%*COS(X)*SIN(X*0.
95)
80 IF RND>0.95 THEN INK 1,INT(RND*26+1)
90 NEXT X
100 WHILE INKEY$=""
110 FOR J=1 TO 16
120 INK 1,J
130 FOR DELAY=1 TO 100
140 NEXT DELAY
150 NEXT J
160 WEND
```

From that, we move to the most impressive *Dionysian Spiral:*

```
10 REM    DIONYSIAN SPIRAL
20 REM * HIGHLY EFFECTIVE *
30 RANDOMIZE TIME
40 INK 0,6:BORDER 6,2:MODE 0:CLG 0:PAPER
 0:SPEED INK 3,7
50 VARS=200:VIM=60
60 DIM Z(14)
70 FOR K=0 TO 14
80 INK K+1,K
90 NEXT K
100 INK 7,6,2:INK 2,24,0:INK 12,16,5
110 ORIGIN (VARS+2*VIM),VARS
120 FOR ALAN=103 TO 0 STEP -1
```

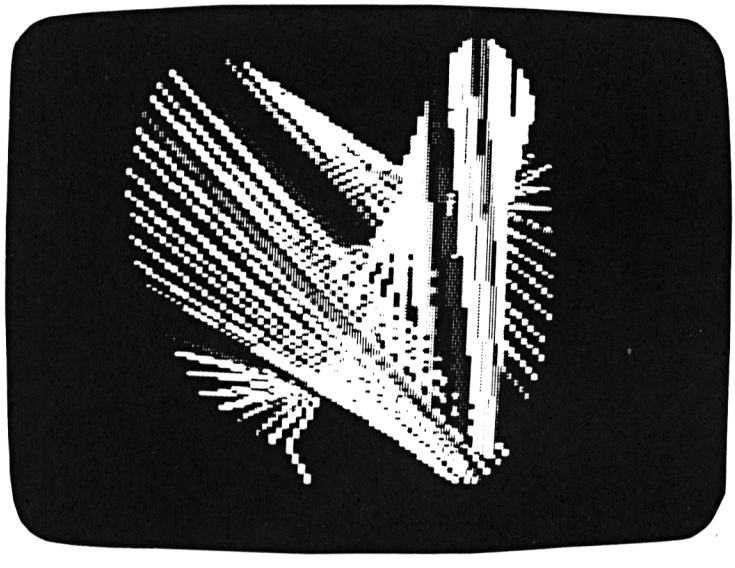
```
130 SUGAR=ALAN*PI/52
140 MOVE COS(SUGAR)*-VIM,SIN(SUGAR)*-VAR
S
150 DRAW COS(SUGAR)*318,SIN(SUGAR)*VIM,A
LAN MOD 13+1
160 NEXT ALAN
170 BORDER 6
180 REM ------------------------
190 WHILE NOT FINISHED
200 JUMP=0.3+RND
210 FOR T=1 TO 16+JUMP STEP JUMP
220 FOR K=0 TO 14
230 INK K+1,K+T
240 IF INKEY$<>"" THEN 350
250 NEXT K
260 NEXT T
270 FOR T=16 TO 2-JUMP STEP -JUMP
280 FOR K=0 TO 14
290 INK K+1,K+T
300 IF INKEY$<>"" THEN 350
310 NEXT K
320 NEXT T
330 IF RND>0.55 THEN H=INT(RND*15):INK 0
,H:BORDER H
340 WEND
350 MODE 1
```

You can terminate this program by pressing any key.

A change of pace comes with *Towering Walls:*

```
10 REM TOWERING WALLS
20 MODE 1:INK 0,0:INK 1,26:INK 2,0:INK 3
,6:BORDER 0
30 RANDOMIZE TIME
40 K=INT(RND*3+2)
50 FOR L=1 TO 3
60 ORIGIN 320,40
70 FOR Z=0 TO 320 STEP 1+RND
80 MOVE K*Z,0:DRAW Z,320,L
90 MOVE -K*Z,0:DRAW -Z,320,L
100 P=P+P/(100+3*L)
110 NEXT Z
```

```
120 NEXT L
130 WHILE INKEY$=""
140 INK INT(RND*3)+1,RND*26
150 FOR J=1 TO 400:NEXT J
160 WEND
```



As an exercise, try and predict what these programs will produce on the screen, *before* you run them:

```
10 ORIGIN 320,200
20 CLS
30 FOR J=200 TO 10 STEP -5
40 Z=COS(J*PI/180)
50 DRAWR Z,J:DRAWR J,Z
60 DRAWR Z,-J
70 DRAWR -J,Z
80 NEXT J
```

```
10 ORIGIN 320,200
20 CLS
30 FOR J=200 TO 10 STEP -15
40 Z=COS(J*PI/180)
50 DRAW Z,J:DRAW J,Z
60 DRAW Z,-J
70 DRAW -J,Z
80 NEXT J
```

```
10 ORIGIN 320,200
20 CLS
30 FOR J=200 TO 10 STEP -15
40 Z=COS(J*PI/180)
50 DRAW Z,J/2:DRAW J/2,Z
60 DRAW Z,J:DRAW J,Z
70 DRAW Z,-J
80 DRAW Z,-J/2
90 DRAW -J,Z
100 DRAW -J/2,Z
110 NEXT J
```

```
10 ORIGIN 320,200
20 CLS
30 K=0
40 FOR J=200 TO 10 STEP -15
50 Z=COS(J*PI/180)+K
60 DRAW Z,J/2:DRAW J/2,Z
70 DRAW Z,J:DRAW J,Z
80 DRAW Z,-J
90 DRAW Z,-J/2
100 DRAW -J,Z
110 DRAW -J/2,Z
120 K=K+11
130 NEXT J
```

Next, fasten your seatbelt for *Warp Factor V*:

```
10 REM WARP FACTOR V
20 RANDOMIZE TIME
30 MODE 0
40 FOR J=1 TO 15:INK J,J:NEXT J
50 ORIGIN 320,200
60 FOR Z=320 TO 10 STEP -(25+RND*20)
70 FOR X=-Z TO Z STEP (RND*8+S/5)
80 P=(Z+X) MOD 15
90 MOVE -Z,X:DRAW Z,-X,P
100 MOVE X,-Z:DRAW -X,Z,P
110 NEXT X
120 NEXT Z
130 WHILE INKEY$=""
140 INK (INT(RND*15)+1),(INT(RND*15)+1)
150 FOR H=1 TO 100:NEXT H
160 WEND
```

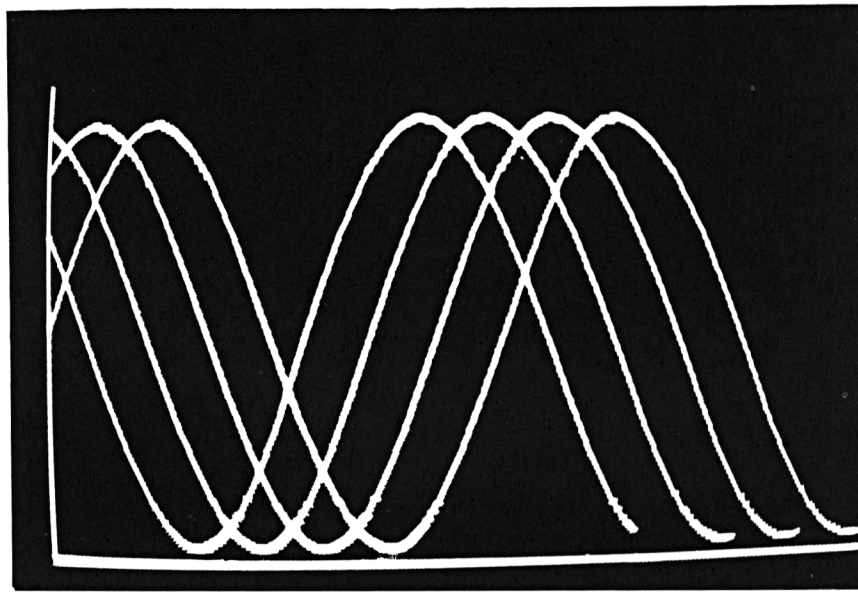
Frequency Waves

The next seven program show how you can plot a variety of sine, cosine and tangent functions with the Amstrad:

```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
50 FOR J=1 TO 350 STEP 50
60 MOVE 0,0
70 FOR CPC=0 TO 640-J STEP 4
80 Y=170+158*SIN((CPC+J)*PI/180)
90 DRAW CPC,Y
100 NEXT
110 NEXT
```

```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
```

```
50 MOVE 0,0
60 FOR CPC=0 TO 640 STEP 4
70 Y=170+158*COS(CPC*PI/180)
80 DRAW CPC,Y
90 NEXT
```



```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
50 MOVE 0,0
60 FOR CPC=0 TO 640 STEP 4
70 Y=100+28*TAN(CPC*PI/180)
80 DRAW CPC,Y
90 NEXT
```

```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
```

```
50 MOVE 0,0
60 FOR CPC=0 TO 640 STEP 4
70 Y=100+28*TAN(CPC*PI/180)/COS(CPC*PI/1
80)
80 DRAW CPC,Y
90 NEXT
```

```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
50 MOVE 0,0
60 FOR CPC=0 TO 640 STEP 4
70 Y=100+28*TAN(CPC*PI/180)/EXP((CPC/3)*
PI/180)
80 DRAW CPC,Y
90 NEXT
```

```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
50 MOVE 0,0
60 FOR CPC=0 TO 640 STEP 4
70 Y=200+40*EXP(CPC/7*PI/240)*SIN(CPC*PI
/70)
80 DRAW CPC,Y
90 NEXT
```

```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
50 MOVE 0,0
60 FOR CPC=0 TO 640 STEP 4
70 Y=1+EXP(CPC/2*PI/150)+EXP(CPC/2*PI/15
0)*SIN(CPC*PI/55)
80 DRAW CPC,Y
90 NEXT
```

```
10 REM FREQUENCY WAVES
20 MODE 2
30 MOVE 0,0:DRAW 640,0
40 MOVE 0,-360:DRAW 0,360
50 MOVE 0,0
60 FOR CPC=0 TO 640 STEP 4
70 Y=1+EXP(CPC/2*PI/150)+EXP(CPC/2*PI/15
0)
80 DRAW CPC,Y
90 NEXT
```

User-Defined Graphics Library

Finally in this section, we have a large number of pre-designed graphics symbols for you to use in your own programs. Designers include Dilwyn Jones, David Perry and Daniel Haywood. As you can see, the DATA has been provided to redefine character 240. Simply change the 240 which follows the word SYMBOL to whichever number character you wish to change.

You are welcome to include these graphics in your own programs. However, if you submit the programs for publication in any form, I'd appreciate an acknowledgement along the following lines in REM statements near the start of the program:

```
10 REM FANTASTIC PROGRAM
20 REM  USER-DEFINED GRAPHICS USED
30 REM  BY PERMISSION FROM
40 REM 'THE AMAZING AMSTRAD OMNIBUS'
50 REM  BY MARTIN FAIRBANKS
60 REM  INTERFACE PUBLICATIONS 1985
```

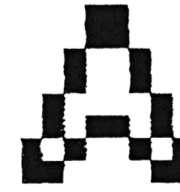
Space Ships

Zenon Ship Upwards

```
SYMBOL 240,24,24,189,189,255,255,189,24
```

Zenon Saucer

```
SYMBOL 240,0,0,0,16,254,124,124,0
```



'Asteroid' Ship — Up

```
SYMBOL 240,24,24,36,36,66,90,165,195
```

'Asteroid' Ship — North East

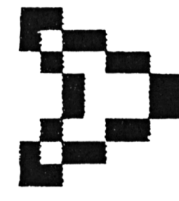
```
SYMBOL 240,3,13,50,194,52,20,8,8
```

'Asteroid' Ship — Right

```
SYMBOL 240,192,176,76,35,35,76,176,192
```

'Asteroid' Ship — South East

```
SYMBOL 240,8,8,20,52,194,50,13,3
```



'Asteroid' Ship — Down

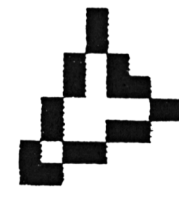
```
SYMBOL 240,195,165,90,66,36,36,24,24
```

'Asteroid' Ship — South West

```
SYMBOL 240,16,16,40,44,67,76,176,192
```

'Asteroid' Ship — Left

```
SYMBOL 240,3,13,50,196,196,50,13,3
```



'Asteroid' Ship — North West

```
SYMBOL 240,192,176,76,67,44,40,16,16
```

Alien Invaders

Static Invader — One

```
SYMBOL 240,129,90,60,90,60,102,153,195
```

Static Invader — Two

```
SYMBOL 240,129,66,60,219,165,24,36,195
```

Static Invader — Three

```
SYMBOL 240,195,165,24,60,90,36,24,231
```

Static Invader — Four

```
SYMBOL 240,129,102,60,255,90,255,129,231
```

Static Invader — Five

```
SYMBOL 240,60,126,219,219,255,126,66,102
```

Squishy Walking Intruder — Frame One

```
SYMBOL 240,60,126,235,235,255,126,36,231
```

Squishy Walking Intruder — Frame Two

```
SYMBOL 240,60,126,215,215,255,126,66,102
```

Fred Astaire Alien — Frame One

```
SYMBOL 240,66,36,189,172,255,60,36,231
```

Fred Astaire Alien — Frame Two

```
SYMBOL 240,36,36,60,247,189,189,36,36
```

Royal Wave Alien — Frame One

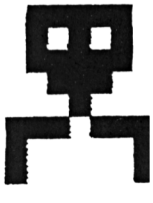
```
SYMBOL 240,129,189,189,173,239,60,26,231
```

Royal Wave Alien — Frame Two

```
SYMBOL 240,231,165,189,181,52,255,129,231
```

Magic Mushroom Galaxian — Frame One

```
SYMBOL 240,0,24,60,90,126,24,36,90
```



Magic Mushroom Galaxian — Frame Two

```
SYMBOL 240,60,90,126,60,24,25,90,0
```

Hokey Kokey Alient — Frame One

```
SYMBOL 240,198,34,20,42,28,34,187,230
```

Hokey Kokey Alien — Frame Two

```
SYMBOL 240,99,68,40,84,56,68,136,103
```

The Dreaded Feetoid

```
SYMBOL 240,124,84,124,0,0,0,0,198
```

Chrisso Alien

```
SYMBOL 240,36,24,27,254,190,36,102,0
```

Johnboy Alien

```
SYMBOL 240,60,126,219,231,255,146,219
```

Spartacus

```
SYMBOL 240,66,60,90,126,60,24,36,66
```

Drodericticus

```
SYMBOL 240,66,36,189,165,255,60,36,231
```

Tautorialian

```
SYMBOL 240,62,42,62,28,8,119,65,65
```

Instruments of War

Missile Base One

```
SYMBOL 240,24,24,24,60,126,255,219,219
```

Missile Base Two

```
SYMBOL 240,0,24,24,24,255,255,255,255
```

Missile Base Three

```
SYMBOL 240,24,24,24,60,255,255,255,0
```

Tank — Facing Right

```
SYMBOL 240,0,0,64,63,112,126,255,126
```

Tank — Facing Left

```
SYMBOL 240,0,0,20,252,14,126,255,126
```

Falling Bomb — at angle

```
SYMBOL 240,32,16,160,92,30,31,15,7
```

Falling Bomb — straight down

```
SYMBOL 240,54,28,8,28,62,62,28,8
```

Little People

Head of The Stranger

```
SYMBOL 240,56,84,108,124,56,16,124,254
```

Legs of The Stranger

```
SYMBOL 240,189,189,189,36,36,36,36,102
```

Bradley K Beings

```
SYMBOL 240,56,56,16,124,16,56,68,68
```

Robbo W. Running One Way

```
SYMBOL 240,60,126,90,255,61,24,38,96
```

Robbo W. Running the Other Way

```
SYMBOL 240,60,126,90,255,188,24,100,6
```

Robbo W. Standing Still

```
SYMBOL 240,60,126,90,255,60,24,102,0
```

Stevey Lad — one

```
SYMBOL 240,25,26,252,24,60,44,100,6
```

Stevey Lad — Two

```
SYMBOL 240,152,88,63,24,60,52,38,96
```

Stevey Lad — Three

```
SYMBOL 240,24,56,25,63,216,60,39,225
```

Stevey Lad — Four

```
SYMBOL 240,12,156,68,62,29,54,34,99
```

Shinto — Running Right

```
SYMBOL 240,24,24,16,254,16,28,116,70
```

Shinto — Running Left

```
SYMBOL 240,24,24,8,127,8,56,46,98
```

Shinto — Jumping

```
SYMBOL 240,24,24,16,254,16,208,112,24
```

Shinto — Climbing

```
SYMBOL 240,90,90,126,24,24,126,66,66
```

Fruit

Strawberry

```
SYMBOL 240,24,82,247,255,255,126,60,24
```

Raspberry

```
SYMBOL 240,4,8,8,86,171,213,106,60
```

Cherry

```
SYMBOL 240,8,8,20,20,34,99,243,96
```

Peach

```
SYMBOL 240,44,110,231,247,247,247,102,44
```

Apple

```
SYMBOL 240,24,82,255,255,255,255,126,36
```

Aeroplanes

Plane — Up

```
SYMBOL 240,24,24,60,126,255,24,24,60
```

Plane — Right

```
SYMBOL 240,16,24,156,255,255,156,24,16
```

Plane — Down

```
SYMBOL 240,60,24,24,255,126,60,24,24
```

Plane — Left

```
SYMBOL 240,8,24,57,255,255,57,24,8
```

The Zoo

The Owl from Antares

```
SYMBOL 240,255,153,231,231,231,126,42
```

Curly Snake

```
SYMBOL 240,0,224,228,74,74,74,49,0
```

Guppy

```
SYMBOL 240,137,64,20,42,21,64,21,40
```

Elephant

```
SYMBOL 240,32,127,95,127,127,91,219,27
```

Dog — Mk 1

```
SYMBOL 240,130,135,254,254,254,68,68,102
```

Dog — Mk 11

```
SYMBOL 240,35,226,162,254,190,62,18,54
```

Frog

```
SYMBOL 240,0,153,189,90,126,126,189,129
```

Left Half of Crocodile Head

```
240,0,0,3,2,15,63,255,0
```

Right Half of Crocodile Head

```
SYMBOL 240,6,12,152,240,224,85,255,0
```

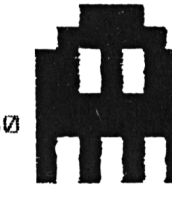
'Pacman'

Right Pacperson

```
SYMBOL 240,60,127,252,240,240,252,127,60
```

Left Pacperson

```
SYMBOL 240,60,254,63,15,15,63,254,60
```



Up Pacperson

```
SYMBOL 240,66,66,231,231,255,255,126,60
```

Down Pacperson

```
SYMBOL 240,60,126,255,255,231,231,66,66
```

Ghost Person

```
SYMBOL 240,56,124,214,214,254,254,170,170
```

Energy Tablet

```
SYMBOL 240,0,24,60,126,126,60,24,0
```

Food Dot

```
SYMBOL 240,0,0,0,24,24,0,0,0
```

Curved Corners

Upper Left

```
SYMBOL 240,255,252,240,224,192,192,128,128
```

Upper Right

```
SYMBOL 240,255,63,15,7,3,3,1,1
```

Lower Left

```
SYMBOL 240,128,128,192,192,224,240,252,255
```

Lower Right

```
SYMBOL 240,1,1,3,3,7,15,63,255
```

Timber!

Section of Floating Log

```
SYMBOL 240,0,255,170,85,85,170,255,0
```

Small Log — Left End

```
SYMBOL 240,127,182,189,175,127,0,0,0
```

Small Log — Centre A

```
SYMBOL 240,255,237,238,121,255,0,0,0
```

Small Log — Centre B

```
SYMBOL 240,255,121,207,125,255,0,0,0
```

Small Log — Right End

```
SYMBOL 240,252,62,230,190,252,0,0,0
```

Alternative Card Suits

Club

```
SYMBOL 240,0,8,28,42,127,42,8,28
```

Spade

```
SYMBOL 240,0,16,56,124,254,254,84,16
```

Heart

```
SYMBOL 240,0,108,254,254,124,56,16,0
```

Diamond

```
SYMBOL 240,0,16,56,124,254,124,56,16
```

Rolling Ball

Frame One

```
SYMBOL 240,60,66,135,143,143,159,126,60
```

Frame Two

```
SYMBOL 240,60,126,249,241,241,225,66,60
```

Fractions

Quarter

```
SYMBOL 240,68,72,82,38,74,146,31,2
```

Half

```
SYMBOL 240,68,72,80,44,66,132,8,15
```

Three-Quarters

```
SYMBOL 240,225,34,68,42,246,42,79,130
```

Board Games Pieces

Star

```
SYMBOL 240,16,56,254,124,56,108,130,0
```

Spinning Wheel

```
SYMBOL 240,24,102,126,165,165,126,102,24
```

Asmovian Cross

```
SYMBOL 240,153,90,60,255,255,60,90,153
```

Octogonal

```
SYMBOL 240,60,126,255,255,255,255,126,60
```

Circular

```
SYMBOL 240,60,126,255,255,255,255,126,60
```

King of the Road

BJ Car — Right

```
SYMBOL 240,0,238,68,255,255,68,238,0
```

BJ Car — Down

```
SYMBOL 240,90,126,90,24,90,126,90,24
```

BJ Car — Left

```
SYMBOL 240,0,119,34,255,255,34,119,0
```

BJ Car — UP

```
SYMBOL 240,24,90,126,90,24,90,126,90
```

P76 Car — Left Half

```
SYMBOL 240,15,18,34,127,255,255,40,16
```

P76 Car — Right Half

```
SYMBOL 240,128,64,32,254,254,255,40,16
```

Section A of Truck

```
SYMBOL 240,127,99,123,119,99,255,21,8
```

Section B of Truck

```
SYMBOL 240,254,182,206,206,183,255,64,128
```

Front of Truck

```
SYMBOL 240,0,248,196,196,254,254,40,16
```

Telunk Car Front — Facing Left

```
SYMBOL 240,1,2,4,127,127,255,20,8
```

Telunk Car Back

```
SYMBOL 240,240,72,68,254,255,255,20,8
```

Front of Left-Facing Truck

```
SYMBOL 240,0,31,35,35,127,127,20,8
```

Section A of Left-Facing Truck

```
SYMBOL 240,127,99,123,119,227,255,2,1
```

Section B of Left-Facing Truck

```
SYMBOL 240,254,182,206,206,182,255,168,16
```

Assorted Shapes and Designs

Barrel — End View

```
SYMBOL 240,60,102,255,165,165,255,102,60
```

Part of Ladder

```
SYMBOL 240,66,126,66,66,66,126,66,66
```

Hour-Glass

```
SYMBOL 240,255,126,60,24,24,60,126,255
```

Scatter Pattern

```
SYMBOL 240,85,34,85,136,85,34,85,136
```

Solid Line

```
SYMBOL 240,0,0,0,255,255,0,0,0
```

Checkerboard Pattern

```
SYMBOL 240,170,85,170,85,170,85,170,85
```

Small Central Dot

```
SYMBOL 240,0,0,0,0,24,24,0,0
```

Parallel Vertical Lines

```
SYMBOL 240,129,129,129,129,129,129,129,129
```

Acorn

```
SYMBOL 240,14,25,60,126,126,60,60,24
```

Decaying Acorn

```
SYMBOL 240,4,9,36,58,102,12,32,24
```

Bricks

```
SYMBOL 240,254,254,254,0,239,239,239,0
```

End of Bolt

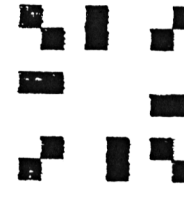
```
SYMBOL 240,60,114,225,193,131,135,78,60
```

Female Symbol

```
SYMBOL 240,16,40,68,40,16,16,56,16
```

Male Symbol

```
SYMBOL 240,15,3,53,73,132,132,72,48
```



Yacht

```
SYMBOL 240,8,24,60,126,8,255,126,60
```

Up Arrow

```
SYMBOL 240,0,24,126,24,24,24,24,0
```

Explosion

```
SYMBOL 240,145,82,0,192,3,0,74,137
```

Raindrop

```
SYMBOL 240,0,16,16,56,56,124,56,0
```

Section Two
The Amstrad Disc Companion

Once you've used disc drives, you'll never want to go back to using cassettes. If you have a 464, the disc drive unit (the DDI-1) is a peripheral which is well worth acquiring.

Back in the dim, dark past (some 20 years, or so, ago), computers used rotating drums, covered with a layer of ferric oxide which could be written to in much the same way as magnetic tape is written to. The amount of memory provided by these drums was, of course, related to the surface area of the drums.

Drums however, took up an awful lot of space, so an IBM engineer decided they should be 'squashed flat', and so the first disc memory was developed. Early discs were very, very inefficient compared to the three-inch discs provided for the Amstrad. It took 50 discs to provide five megabytes of memory. The Amstrad discs provide 360K each. They are double-sided, with 180K on each side.

Getting Started with Discs

If you've using a CPC-464, or an additional drive with your 664, always turn the disc drive(s) on *before* turning on the computer. Never turn the drive(s) on or off with a disc in them. I ignored this advice, and managed to 'scramble' one of my favourite games. It was unrecoverable. You have been warned.

When you turn your computer on, it checks to see if there is a disc drive in the system. If there is, all LOAD, SAVE and CAT commands are directed to the disc. If there are no disc drives, or you have turned them on, these commands go to the datacorder on the 464. If you do have a disc connected on the 464 and you want to use the datacorder instead, or you want to direct LOAD/SAVE operations through the 664's cassette port, use the

command |TAPE. To get back to disc, use |DISC. (Note that both TAPE and DISC, along with all commands for using the disc drives, and for naming files, can be in upper or lower case; your Amstrad computer does not differentiate between them for disc handling.)

To see what is on any particular disc, type in CAT (for catalogue) and, after a few moments' whirring, the computer will print up (in alphabetical order) the names of all the files on the disc. To run any one of them, just enter run "filename."

A file name is made up of two parts, or 'fields', separated by a full stop. The first part can be up to, but not more than, eight letters. The second part (which is optional) is up to three letters long. If you don't add a second part, the computer will add one of its own (using BAS for a BASIC file, BAK for a backup file, or BIN for a machine code/binary file). You only need to specify the second file, when loading or running a program, if the second field has been added by you, or there are two files with the same first field name, and you wish to specify which of the files you wish to run.

If you're an experienced user of the datacorder, you'll be pleased to discover that with one exception (the command 'cat'), all the cassette read/write commands work with the disc unit.

File reading commands:

LOAD, RUN, CHAIN, MERGE, CHAIN MERGE, OPENIN, EOF, INPUT #9, LINE INPUT #9, CLOSEIN

File writing commands:

SAVE, OPENOUT, PRINT #9, WRITE #9, CLOSEOUT

The cataloguing command, CAT, causes the disc directory to be sorted into alpha-numeric order, and displayed in columns (together with the individual file sizes, and the amount of free disc space which is still left).

If there is no disc in the drive, or it is incorrectly inserted, you'll get the following message when you try any disc operation:

```
Drive A: disc missing
  Retry, Ignore or Cancel?
```

The best thing to do, assuming there is a disc in the drive when you get that message, is to take the disc out, and re-insert it, very carefully, then hit the 'R' key (for Retry).

If the file you've specified is not on the upward-facing side of the disc you've inserted into the drive (because you've got the wrong disc in the drive; the wrong side of the disc is facing up; or you have made a mistake in the spelling of the specified file name), you'll get the following report from your computer:

```
FILENAME.      not found
```

If you've put more than eight letters before the full stop in the specified name, or more than three in the second field (or you've included something odd in the name, like a punctuation mark or a spurious space), your Amstrad will report:

```
Bad command
```

Leaving out the quote marks before the name of the file will give you the message:

```
Type mismatch
```

There are two other commands which are common at this point. If you get *Syntax error* on the screen, you've probably spelt *run* incorrectly, while *Press PLAY then any key:* means either your disc drive is not plugged in, is not turned on, or you've executed a |TAPE which has not been undone.

Formatting Blank Discs

Before you can use a disc, it must be formatted, a process which prepares a 'superstructure' onto which the data you wish to record can be placed. The computer system is operating under *Amsdos* (Amstrad Disc Operating System) when you first turn it on. You need to have the disc under control of CP/M (which we'll be looking at shortly) in order to format a disc (and so you can do many other things, such as copying disc, or copying particular files). To get into CP/M, type in |cpm. The drive will make a few whirring noises, and after a second or two you'll see this on the screen:

```
CP/M 2.2 - Amstrad Consumer Electronics plc.

A>
```

The A> is the world's least user-friendly prompt. The 'A' refers to the disc drive, with a second drive (if one is present) being referred to as 'B'.

To format a disc, type in FORMAT, and the disc drive will make its customary noises as it loads into computer memory the information it needs to format a disc. The computer will tell you to insert a disc and then press any key; will make lots and lots of noise as it formats the disc; and then will ask you if you want to format another disc. If you do not, you'll be asked to re-insert the master disc into the computer.

There are two things to note about formatting. Firstly, anything on the disc you are formatting will be wiped off by the FORMAT command, so be careful you don't insert a disc with something important on it into the machine for wiping, Secondly, if you want to copy a whole disc, you can copy it onto an unformatted disc, and the blank disc will be automatically formatted as the copying is done.

Backing up the System Disc

The 'master control' or 'system disc' is very valuable. You should make a copy of it immediately, and then put the original away somewhere safe, just in case you damage it in some way. To copy the system disc, when you have one disc drive, put your system disc in the drive, enter CP/M with |cpm, and then type in DISCCOPY. Now *leave* the system disc in the drive when you are asked to *Please insert source disc into drive A then press any key,* and press any key. Follow the instructions regarding taking the source and destination discs (where the source is the system disc, and the destination is a blank disc) in and out, and — after a rather tiresome number of disc swaps — you'll have a copy of one side of your system disc.

Answer 'yes' (by pressing the 'Y' key) when asked if you want to copy another disc, and turn the system disc over before putting it into the drive when asked to insert the source disc. Make sure you turn the destination disc over as well. Once you go through the whole disc-swapping process again, you'll have a back-up copy of the system disc. Label it as such, and 'open' both the write-protect holes on the copy you've made, to ensure it is not accidentally overwritten. Now, put the original disc safely away, where it will be safe from magnets and liquids.

To check if a particular file is present on the disc currently in drive A, enter:

```
A>dir ed.com
```

This checks to see if ED.COM is present. If it is, your computer will respond with:

```
A: ED       COM
```

If it is not present, you'll get this message:

```
NO FILE
```

To check *all* available command files, enter:

```
A>dir *.com
```

Copying Programs

Cassette to Disc

The cassette containing the program is placed in the datacorder, and the command |tape.in is entered, followed by load "program name" or simply load"". The Amstrad responds with the message Press PLAY then any key:, and once the program has loaded, the cassette will stop. The program is now in memory, and must be transferred to the disc, so you enter save "filename". This does not work, of course, with protected cassette programs.

Disc to Cassette

The command |disc.in must be entered first, if you have been using the cassette recorder. Then, you enter load "filename", and wait until the Ready prompt appears. Now, to save the program on cassette, enter |tape.out followed by save "program name" or just save"". Your Amstrad will give you the message Press REC and PLAY then any key.

Control Program for Microcomputer

The most important track on your system disc is CP/M. CP/M was one of the first operating systems written to enable microcomputers to handle discs, and version 2.2 — the one provided with your disc unit — has become the accepted operating system for business, and is now provided on hundreds of machines.

CP/M was invented by a (then) young computer whiz called Gary Kildall, while he was working for the semiconductor manufacturer, Intel. Gary wanted to hook floppy discs to his home-built microcomputer. And there was no way to do this. So, hunched over his primitive computer in the garden shed at the back of his parents' house (and using 'stolen' time at work), Gary wrote the first, relatively primitive, version of CP/M.

Once it was up and running, Gary thought there might be a few other microcomputer enthusiasts who would be interested in his operating system, and offered it to Intel to market. But, as *Your Computer* pointed out in its March, 1985 article *What does CP/M do?,* Intel "scoffed at the preposterous suggestion of bothering to write an operating system for a mere micro. After all, IBM itself had let it be known that it thought micros were merely a passing fad . . ."

Freed of any obligation to Intel, Gary set to work to market it by mail through computer magazines under the trading name of 'Digital Research'. Intel was massively wrong. From mail order ads, Digital Research grew and prospered. Gary is now massively rich, and Digital produce a wide range of software products (including the Logo, *Dr. Logo,* provided on the system disc).

Although CP/M was not the world's best-written program, it fulfilled a need, and it was the first program to do so. The more widely it was used, the more material became available for it. And, of course, the growing body of CP/M-compatible software encouraged more and more people to use it. In the April, 1985 issue of *Acorn User* magazine, Edward Brown's article *Making the Most of CP/M* points out that the wide acceptance of CP/M has meant that "thousands of software packages are available, covering almost every conceivable application . . . Many packages are also available in the public domain and can be obtained for little or no cost . . .". Unfortunately for us as Amstrad owners, the majority of the material is available only on eight, or five and a quarter, inch discs, and quite a bit of it needs a fair amount of juggling to get it to run. However, the growing number of interested Amstrad owners means that a lot of the more

interesting material is currently being converted. The CP/M User Group UK, at 72 Mill Road, Hawley, Dartford, Kent, DA2 7RZ, has a large amount of public domain CP/M material available at a nominal cost.

CP/M is made up of four main components: (1) the Basic Input/Output System (BIOS); (2) the Basic Disc Operating System (BDOS); (3) the Console Command Process (CCP); and (4) the Transient Program Area (TPA).

The BIOS provides the primitive input/output interface to the disc drives and input/output devices, while the BDOS controls the disc drives and file system. CCP reads the keyboard and processes commands, and TPA holds programs loaded in from disc.

To see what is on side A of your system disc, enter the command DIR (for directory, when in CP/M, as opposed to being in Amsdos, when you need to use CAT, for catalogue) and you'll see the following:

```
MOVCPM.COM

ED.COM

STAT.COM

FILECOPY.COM

CHKDISC.COM

FORMAT.COM

EX2.BAS

SUBMIT.COM

DDI.COM

DUMP.ASM

BOOTGEN.COM

DISCHE.COM

CLOAD.COM
```

```
XSUB.COM
LOAD.COM
AMSDOS.COM
COPYDISC.COM
SETUP.COM
EXI.BAS
PIP.COM
ASM.COM
DUMP.COM
SYSGEN.COM
DISCCOPY.COM
CSAVE.COME
POINTIME.COM
```

This extensive range of functions indicates just how flexible and useful CP/M can be. The main tasks you can carry out using CP/M are as follows:

— format discs for use on the Amstrad

— copy the contents of one disc to another

— check the size and read/write status of disc files

— display a disc directory or selected sections of the directory

— display the contents of a disc file on the screen

— erase and rename files

— copy or move files from one disc to another

— output files to a peripheral device, such as a printer

— execute a series of commands with one command

— create and edit files

Here are the utility CP/M programs on your system disc:

AMSDOS.COM — returns to AMSDOS and BASIC from CP/M

ASM.COM —8080 assembler

BOOTGEN.COM — copies boot and configuration sectors from one disc to another

CLOAD.COM — copies file from cassette to disc

CSAVE.COM — copies file from disc to cassette

CHKDISC.COM — compares two discs (two drives needed)

COPYDISC.COM — copies one disc to another (two drives needed)

DDT.COM — dynamic debugging tool; 8080 debugger

DISCCHK.COM — compares two discs (using a single drive)

DISCCOPY.COM — copies one disc to another (using a single drive)

DUMP.COM — displays a file on the screen in hex

ED.COM — text editor

FILECOPY.COM — copies files from one disc to another (single drive)

FORMAT.COM — formats a disc

LOAD.COM — reads file in Intel HEX format; produces COM file

MOVECPM.COM — contracts CP/M system of any given size

PIP.COM — peripheral interchange program; copies files on disc and between other peripherals

SETUP.COM — changes parameters in configuration sector

STAT.COM — gives details on files, discs, users and IOBYTE

SUBMIT.COM — takes CP/M commands from file instead of from keyboard

SYSGEN.COM — writes CP/M system onto system tracks

XSUB.COM — used with SUBMIT.COM for buffered program input

BIOS Jumpblocks

There are two jumpblocks, the standard CP/M one and the extended jumpblock, which starts at #BE80. Details of this jumpblock follow:

```
0: SET MESSAGE       #BE80 - enables/disables
                             disc error messages

1: SETUP DISC        #BE83 - resets various
                             disc parameters

2: SELECT FORMAT     #BE86 - selects AMSTRAD
                             format

3: READ SECTOR       #BE89 - reads physical
                             sector from disc

4: WRITE SECTOR      #BE8C - writes physical
                             sector to disc

5: FORMAT TRACK      #BE8F - formats entire
                             track

6: MOVE TRACK        #BE92 - unverified move to
                             specified track

7: GET DR STATUS     #BE95 - return status for
                             specified drive

8: SET RETRY COUNT   #BE98 - sets number of
                             retries on error
```

```
9: ENTER FIRMWARE    #BE9B - calls firmware
                             routine

10: SET REG SAVE     #BE9E - sets/clears saving
                             of alternate and
                             IY registers

11: SET SIO          #BEA1 - resets and initializes
                             serial interface

12: SET CMND BUFFER #BEA4 - initializes the initial
                             command buffer

13: D0 IN STATUS     #BEA7 - tests if special
                             character I/O device 0
                             has character available
14: D0 IN            #BEAA - fetches character

15: D0 OUT STATUS    #BEAD - tests if output ready

16: D0 OUT           #BEB0 - outputs character

17: D1 IN STATUS     #BEB3 - tests if special
                             character I/O device 1
                             has character available

18: D1 IN            #BEB6 - fetches character

19: D1 OUT STATUS    #BEB9 - tests if output ready

20: D1 OUT           #BEBC - outputs character
```

External Commands

```
|CPM       - shuts down BASIC and AMSDOS and
             cold boots CP/M

|DISC.IN   - redirects tape input firmware
             entries to disc counterparts

|DISC.OUT - redirects tape output firmware
             entries to disc counterparts
```

```
|DISC      - command redirects tape input and
             output firmware entries to disc
             counterparts

|TAPE.IN   - restores tape input firmware
             entries to pre-AMSDOS
             initialization state

|TAPE.OUT  - restores tape output firmware
             entries

|TAPE      - restores tape firmware entries

|A         - sets default drive to drive A

|B         - sets default drive to drive B

|DRIVE     - sets current default drive

|USER      - sets default user number

|DIR       - displays disc directory

|ERA       - erases files

|REN       - renames a file
```

Amsdos Error Messages

Unknown command — command spelt wrongly, or disc interface not connected correctly (or not connected at all)

Bad command — syntax error or hardware failure prevents command being carried out

filename already exists — you cannot use the same name for two files on the same disc

filename not found — file is not on the disc in the drive

Drive A: directory full — no room for new entry in directory

Drive A: disc full — no room on disc

Drive A: disc changed, closing filename — disc has been with files still open on it

filename is read only — attempt to write to a read-only file

Drive A: disc missing — no disc in drive, or disc not seated properly

Drive A: disc is write protected — the Write Protect hole is open, preventing further material being saved

Drive A: read fail — hardware error

Drive A: write fail — hardware error

Failed to load CP/M — no CP/M on disc, or read error following |cpm command

Further Reading

If you want to know more about CP/M, the following books may be of interest:

CP/M and the Personal Computer — Thomas A Dwyer & Margot Crithfield, Addison-Wesley, 1983

The Best of CP/M Software — John D Halamka, Sybex, 1984

The CP/M Handbook with MP/M — Rodnay Zaks, Sybex, 1980

CP/M Primer — Stephen M Murtha & Mitchell Waite, Sams, 1980

CP/M Revealed — Jack D Dennon, Hayden, 1982

CP/M Simplified — Jeffrey R Weber, Weber Systems, 1982

Mastering CP/M — Alan R Miller, Sybex, 1983

Programmer's Guide to CP/M — Sol Libes, Microsystems Press, 1982

Section Three
Exploring
Artificial Intelligence

Part One —
Learning and Reasoning

There is a continuing debate as to whether producing a machine which can behave in a manner which appears intelligent is actually taking us any closer to really producing intelligence. A related question, inextricably bound up in the debate, concerns the nature of intelligence.

The programs in this part of the book certainly allow your computer to exhibit intelligent responses to situations, making decisions and acting on them. However, there is no suggestion that your computer has awareness of its actions. It does not laugh at the nonsequiturs produced in ELIZA and cannot admire — or even recognize — a particularly effective poem produced by HANSHAN.

Is there, then, any justification for claiming that we are producing 'artificial intelligence'? It seems to me that without the kind of perception which recognizes such things as the 'effectiveness' of a poem, or the incongruity of a response, we cannot really suggest that intelligence is present.

Al is in its infancy, and to expect to elicit real awareness and perception from a short BASIC program on an Amstrad, when the largest mainframe machines have not even scratched the surface of this area, is unrealistic.

However, there are two areas of behaviour which are both reasonable candidates for classing behaviour as intelligent, and which can be elicited from your own computer. These are the fields of *learning* and *reasoning.*

TICTAC, a program which plays Tic-Tac-Toe (nor Noughts and Crosses) starts its life with just a knowledge of how to win a game, and how to block. It does not have any knowledge as to the early moves it should make

in a game in order to increase its chance of winning. In fact, its initial knowledge base is such that it plays as badly as it can.

But, put it up against an opponent playing totally at random (an opponent who does not even have the rudimentary knowledge that one wins the game by getting three noughts or crosses in a row) and within ten games or so TICTAC will have learnt the value of moving into the central square on the grid if it is available, and will have ordered its other moves into a sequence which — although it differs from the sequence you or I might create in similar circumstances — allows it to win an increasing proportion of its games, even against an intelligent opponent such as yourself. TICTAC has been written to show you the state of its present learning after each game. This makes it a fascinating program to run, and there are many ways you can extend the program to investigate its ability to learn.

SYLLOGY is our reasoning program. It aims to solve syllogisms, such as this early one:

```
SOCRATES IS A MAN
ALL MEN ARE MORTAL
THEREFORE, SOCRATES IS MORTAL
```

From the two initial premises, SYLLOGY draws a reasonable conclusion. The important thing to note is that SYLLOGY can reach conclusions about information which has not been explicitly fed into it.

I'll explain that. Look at these two premises:

```
A NOVEL IS A BOOK
A BOOK IS PRINTED ON PAPER
```

Although the program has not been told explicitly that a novel is printed on paper, it will answer YES when presented with this question:

```
IS A NOVEL PRINTED ON PAPER?
```

You can have a great deal of fun feeding in a long range of premises, then asking a variety of questions on them, to see what conclusions SYLLOGY can form. I HAVE NO DATA ON THAT, NO and I DON'T KNOW are all possible responses from SYLLOGY.

In the early stages of the 'could a machine really become intelligent?' debate, it became obvious that the fundamental terms under discussion needed looking at very carefully. What did we actually mean by thought and thinking? If we did not know really know what we meant when using the terms to refer to ourselves, how could we make judgements on the performance of machines in this field?

This sort of thinking is one of the many effects that studying Al has had. Man has been forced to look closely at himself, and to examine areas of human behaviour in a way which very few men had ever bothered to do.

I suggested a short while ago that while machines were not even approaching the kind of awareness which appears vital as a prerequisite for claiming that intelligence actually exists in a system, some aspects of intelligence — reasoning and the ability to learn — were within our present capabilities.

There are different kinds of learning. We can learn by watching others, by reading, by being told (which is a kind of 'verbal reading' so the two are very closely related) and by 'trial and error'. Computers can learn in all these ways. TICTAC learns largely from trial and error, although it has some preprogrammed knowledge (which it gained by 'being told').

Feedback

Of course, TICTAC's trials and errors would be meaningless unless it received feedback as to the success or otherwise of its efforts. Feedback is a vital element of learning.

An early 'machine which would learn' was the turtle, a forerunner of a swarm of such robotic terrapins, built in 1948 by Grey Walter, a physiologist who specialised in the brain. He built his turtle — a half-globe that trundled around the floor, working its way around obstacles, and going home to bed when its batteries were getting low — to demonstrate a thesis that complex behaviour, no matter how involved it looked to an outside observer, was based on interactions between only a few basic ideas.

The turtle learned its way around by utilising negative feedback, that is it would tend not to repeat behaviour which was not productive. A turtle which did not learn that rolling repeatedly into a wall was not a way to move around would cover very little ground.

How Do Machines Think?

Present-day computers are serial processors. That is, they proceed from point to point, one step at a time, with their future steps determined by the results of their previous ones. The human brain, by contrast, uses not only serial processing, but also parallel processing, in which a number of trains of thought — some conscious, others not — are underway at once.

A computer's thought and decision-making process is essentially a path through a maze of IF/THEN constructions:

IF this is true AND this is true
AND this not true THEN do this

The computer, of course, can make OR decisions as well as AND ones:

IF this is true OR this is true
THEN do this

They can be combined:

IF this is true AND that is true OR
something else is true THEN do this

How does it do this? The very first electronic calculating device was built (in his kitchen) by George Stibitz who worked for Bell Telephone Laboratories in the 1940s. He wired up batteries, bulbs and some telephone relay switches, to calculate in binary. (This is the numbering system which has only 0 and 1 as its digits. A switch turned on could be considered set to equal 1, while when off it was regarded as 0.) Stibitz realized that his crude device, if sufficiently expanded, could work on any kind of mathematical problems. (What he apparently did not realise was — as you will learn in a moment — that the same circuits he was using to add binary numbers could be used to reach decisions.)

However, a few years before, in 1937, Claude Shannon (who later also worked for Bell), had gained his master's at MIT with a thesis on the relationship between Boolean Algebra and the flow of power through switched circuits.

Boolean Algebra — which is where the 'thinking' part of machines really begins — is based on the work of George Boole, a lecturer at Queens

College, Cork, in the middle of the nineteenth century. His book *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities* (published in 1854) laid down the foundations of modern symbolic logic. Boolean Algebra is based on the rules he laid out, and is the pivot round which your computer's ability to reason rotates.

Boole wrote in the preface to his work:

> **The laws we have to examine are the laws of one of the most important of our mental faculties. The mathematics we have to construct are the mathematics of the human intellect.**

Until Boole's discoveries, it had been assumed that logic was a branch of philosophy. Boole showed clearly that, instead, it belonged without doubt within the province of mathematics.

Part Two — A Program Which Learns

Many Al programs do not spring into the Amstrad fully formed. Even when they are debugged, and operating, they are far from finished. The program we'll look at in this section, TICTAC (which is a version of TIC-TAC-TOE or NOUGHTS AND CROSSES) is one such 'unformed' program. TICTAC learns as it plays, modifying its rules in light of the success or otherwise of its current behavior.

A program which is going to learn as it runs needs its working rules in a 'soft' form which can be changed as it evolves. In this program, the computer knows the rules of the game, and has a section specifically to block rows of three being formed by its opponent, and to complete a row of three for itself if it gets the opportunity, but it has no strategy at all at the beginning.

Here's the board layout for TICTAC:

1	2	3
4	5	6
7	8	9

The program plays by selecting squares in line with a sequence which it evolves as the games go on. If the game is a success, it moves the positions chosen closer to the front of the sequence. It makes no change if the game is drawn. A loss shuffles the sequence so the moves are less likely to be chosen next time.

You and I know that the centre square (five in the diagram above) is the one to take if it is vacant. Initially, TICTAC does not know this. In fact, it has been deliverately given a very bad opening 'book' — with position two as its first choice — so that it is easier to see the effect its learning has on its play.

Eventually, if the learning mechanism is working, TICTAC should realise that position five is a very good one to possess if it is available. In fact, as we shall see, TICTAC does eventually come to this conclusion, even though it is playing against a totally random opponent which has no strategic knowledge whatsoever. It is reasonable to assume that if TICTAC was playing against an intelligent opponent — such as yourself — the program would improve more rapidly.

Donald Michie, a pioneer in artificial intelligence research at Edinburgh University and still very prominent in the field, investigated 'automatic learning' in the game of noughts and crosses. He used a mechanism called 'boxes' in which a goal is split into several sub-goals. A 'box' is formed to hold the information of each sub-goal.

The goal of noughts and crosses is to win. Each sub-goal is to make at first (a) a legal move and eventually (b) the best move given each game position.

Michie worked out that there are 28 fundamentally different positions which face a player if he or she starts in a game of noughts and crosses. He proceeded to build his mechanical opponent as follows (an experiment you may well want to duplicate). Michie took 288 matchboxes, and painted on the top of each a board position, with the vacant squares numbered in sequence. Next he wrote down, on tiny bits of paper, the numbers which were written on the vacant squares. Each number was duplicated several times, with the same number of each number per box. That is, if squares three and four were vacant in one board position, the matchbox contained, say, five scraps of paper with the number three written on them, and five bearing the number four.

He played the game as follows. The first move was made by opening the box with a blank grid on its top. Inside the box, of course, were five pieces of paper for each of the numbers one to nine. A piece of paper was chosen at random, and the move made there. Michie made a note of which number was selected, and of the box from which the number was chosen.

At the end of the game, Michie returned to his list of moves and boxes. If the 'matchbox computer' had won the game, an additional piece of paper bearing each number played was placed in the relevant matchbox. That is, if the first matchbox used, the one bearing the blank grid, had yielded the number five, an additional piece of paper with the number five on it was placed in that matchbox. Naturally enough, this increased the chance that five would be selected next time the box was opened.

The process was continued for every box used in that game. If the game was drawn, the contents of the boxes were left unchanged. If the 'computer' lost the game, the pieces of paper which triggered the moves in that losing game were withdrawn from the boxes, thus reducing the chance that such numbers would be drawn next time the computer came up against the same board configuration.

In the 1968 paper, *Boxes: An Experiment in Adaptive Control* [Chambers,

R. A. and Michie, D., Machine Intelligence 2 (Ed. Dale, E. and Michie, D.), Oliver & Boyd, 1968, pp. 137-152], Michie explains that the boxes 'learned' so well that after 1000 games against an opponent which played totally at random, the program was consistently winning between 75% and 87% of all games played. A similar success rate is not expected for TICTAC (even if you have the patience to play 1000 games) but it will still perform extremely well if draws as well as wins are counted, and the program is given a proper chance to learn.

Samuel and the Checkerboard

Michie's 'intelligent matchboxes' were but a toy compared to a checkers (or 'draughts') program created in the late sixties by Arthur Samuel of IBM. We are discussing here one of his later programs, as outlined in the paper *Some Studies in Machine Learning Using the Game of Checkers — II — Recent Progress* [Samuel, A., IBM Journal of Research and Development, vol. 11 (November 1967), pp. 601-617]. However, it is interesting to note that the final, acclaimed program did not spring out of his brain in all its majesty.

Samuel had, in fact, began programming checkers games in 1952 working on the (for then) powerful IBM 701 computer. Two years later he transferred the program to an IBM 704, and in 1955 began to develop the program's ability to learn. The program took note of some 40 factors when determining a move, although less than half of these were in use for working out a particular move. The program knew when a particular factor was not contributing towards choosing a move, and ignored that one for the time being.

The number of pieces each player had was an important consideration, and Samuel's program (like the majority of such programs which followed) was quite happy to trade off pieces when it had more than its opponent, but became very conservative in this regard when it was losing, from the material point of view. Other factors the program considered when evaluating its strength included control of the centre of the board and the number of pieces which could be brought under attack by a single move.

We will look more closely a little later at the Al aspects of board games (with the game SNICKERS, invented just for this book) but for now the main interest in Samuel's program lies in its ability to learn. CHECKERS had two ways of learning, rote and self-modification.

In the rote learning mode, the program stored the results of investigations

into possible moves radiating out from a current board position. This meant that next time the position was encountered, the program did not have to actually go through the process of working out its implications. The result was already there. This method, of course, is very memory-hungry, although highly effective. Eventually, the program played close to championship level, and had 'remembered' practically every worthwhile board position.

Samuel's evaluation function, which made use of around 40 factors, was mentioned a short while ago. The self-modification process worked as follows. Samuel allowed the program to search ahead from its present position, and to reach a conclusion as to the value of certain moves and positions. The program also used its evaluation function to reach a conclusion from the same board position.

Samuel reasoned that, if the evaluation function was perfect, it would generate the same advice as the look-ahead mechanism. The factors within the evaluation function were modified after each move, in light of the difference between the finding of the forward search, and the information given by the evaluation function. Working in this way removed the reliance on vast memory backup demanded by the rote-learning process. Our TICTAC program does not learn as did CHECKERS, but its method does involve self-modification, rather than depending upon rote accumulation of information.

Tictac — The Program

The program begins with an initialisation sub-routine. Four arrays are dimensioned. The A array holds the current game board, M holds the 'knowledge base' of moves (this is updated after each winning or losing game), W holds the data from which the program can recognise a potential win by itself or an opponent, and D holds the moves in the current game, so these can be used to modify the knowledge base at the end of a game.

As you can see from line 1350, it starts off with a knowledge base consisting of the numbers 2, 6, 8, 4, 7, 3, 1, 9, 5 and 2. This is as I pointed out earlier, a particularly bad sequence of moves, which practically ensures that it will lose a significant proportion of its early games. If you doubt that, mentally put those moves onto the board we're using in this game:

1	2	3
4	5	6
7	8	9

Note that the program does not necessarily make the moves in the order shown. It attempts to, but may find the relevant square already taken. As well, it does not use its sequence until the pre-programmed knowledge regarding blocking possible completed rows of threes by the opponent, and trying to complete its own, has been tested.

Watching the program learn is particularly fascinating. Therefore, part of the program reports to you at the end of game, showing you the current sequence it is storing. The update of the knowledge base, and its reporting to you, is carried by the section of the program from lines 300 to 480.

Here is the evolving knowledge base of a 'self-playing' version, whose opponent was my computer's unintelligent random number generator. Despite the lack of concentrated opposition, the program managed to learn very rapidly. You can see how quickly TICTAC discovers the value of moving into the centre position (number five on our board):

2	8	6	4	7	3	1	5	9
2	6	4	8	7	3	5	1	9
6	2	4	8	3	7	1	5	9
2	4	6	8	3	7	5	1	9
4	6	2	8	7	3	1	5	9
6	4	2	8	7	3	5	1	9
4	6	2	8	7	5	3	1	9
6	4	2	8	5	7	3	1	9
6	2	4	5	8	7	3	9	1
2	6	5	8	4	7	3	9	1
2	5	6	4	8	7	3	9	2
5	2	6	8	4	7	3	2	9
2	6	5	4	8	7	3	2	9
2	5	6	8	4	7	2	3	9
5	6	2	4	8	2	7	3	9
6	5	4	2	8	2	7	3	9
5	4	6	2	8	2	7	3	9
4	5	2	6	8	7	3	2	9
5	4	6	2	8	7	3	2	9
4	5	2	8	6	7	3	2	9
5	4	2	6	8	7	3	2	9
4	5	6	2	8	7	3	2	9
5	6	4	2	8	7	3	2	9

Next, I used the final sequence obtained from the automatic run (except for changing the duplicated two into a one) in place of the starting

sequence given in the complete program listing, and started to play against the program myself, trying to defeat it in every game. You can see that it continued to learn:

4	5	6	2	8	7	3	1	2
4	5	6	2	8	7	3	1	2
4	6	5	2	8	3	7	1	2
6	5	4	2	8	3	7	1	2
5	4	6	2	8	3	7	1	2
5	4	6	2	8	3	7	1	2
5	4	6	2	8	3	7	1	2
4	5	6	2	8	3	7	1	2
5	4	2	6	8	3	7	1	2

The program was modified slightly, and a new starting sequence, which I judged as the best I could give it, was entered. The computer played first against a human, with the following development (or lack thereof) of its knowledge base:

5	1	3	7	9	2	4	6	8
1	3	7	5	9	2	4	6	8
3	7	5	1	9	2	4	6	8
7	5	3	1	9	2	4	6	8
5	3	7	1	9	2	4	6	8
5	3	7	1	9	2	4	6	8

It was then set to work against the random opponent. You can see that it has little learning to do, and appears simply to be shuffling a few numbers around fairly aimlessly:

1	5	3	9	7	2	4	6	8
5	1	9	3	7	2	4	6	8
1	9	5	3	7	2	4	6	8
1	9	5	3	7	2	4	6	8
9	5	1	3	7	2	4	6	8

Finally, I returned to the poor starting sequence, and let the computer have its head against the random number generator. After 90 games, the sequence was as follows:

4	7	5	3	8	6	9	2	2
7	4	5	3	8	6	9	2	2
4	7	5	3	8	6	9	2	2
7	4	5	3	8	6	9	2	2
7	4	5	3	8	6	9	2	2

You can see one weakness of this program. Although it does learn, after a fashion, it appears to be too easily persuaded to swap numbers, even though this may not necessarily help it play better. You may well want to work on the way the Amstrad makes use of the lessons it gains from each game.

I said earlier that TICTAC's playing strategy does not come solely from its knowledge base. It also has information on the rows which it is trying to build (and which it is trying to prevent its opponent from completing). The section of code which looks for a move, before using the knowledge base, is from 540 to 620. It looks first for a winning move for itself (when P equals the ASCII code of the letter "O") and then tries for a blocking move (with P set equal to the code of the opponent's piece, the "X"). If it fails to find a move here, it brings in the data from the knowledge base.

If this fails to give it a move, it tries numbers at random; using the routine from 680. Having found a move, it makes it, then acts to ensure that, if all positions are filled and R$ (which stands for 'result string' with it being set to "W" for a win. "L" for a loss and "D" for a draw) not assigned, the game must be a draw. After each move, human or machine, the WIN CHECK routine from 870 to 960 is visited.

Here is the TICTAC program, so you can do some investigating of your own into machine education:

```
10 REM TICTAC
20 GOSUB 1180:REM INITIALISE
30 REM -- PREGAME SETTINGS --
40 FOR J=1 TO 9
50 A(J)=32
60 NEXT J
70 FOR J=1 TO 5
80 D(J)=0
90 NEXT J
100 COUNT=0
110 R$=""
120 GOSUB 1070:REM PRINT BOARD
130 REM -- MAIN CYCLE --
140 WHILE R$="":GOSUB 540:REM AMSTRAD MO
VE
150 GOSUB 1070:REM PRINT BOARD
160 GOSUB 870:REM WIN CHECK
```

```
170 IF R$<>"" THEN 240
180 GOSUB 980:REM ACCEPT HUMAN MOVE
190 GOSUB 1070:REM PRINT BOARD
200 GOSUB 870:REM WIN CHECK
210 WEND
220 REM -- REM END MAIN CYCLE --
230 REM ------------------------
240 REM -- END OF GAME --
260 PRINT:PRINT
270 IF R$="W" THEN PRINT TAB(8);"I WIN":
FLAG=-1
280 IF R$="L" THEN PRINT TAB(8);"YOU WIN
":FLAG=1
290 IF R$="D" THEN PRINT TAB(6);"IT'S A
DRAW":GOTO 430
300 REM -- UPDATE KNOWLEDGE BASE --
310 FOR B=1 TO 5
320 FOR J=2 TO 9
330 IF (J)=D(B) THEN GOSUB 370
340 NEXT J
350 NEXT B
360 GOTO 430
370 REM -- RE-ORDER M ARRAY ELEMENTS --
380 TEMP=M(J+FLAG)
390 M(J+FLAG)=M(J)
400 M(J)=TEMP
410 J=9
420 RETURN
430 PRINT:PRINT
440 PRINT "THIS IS MY UPDATED PRIORITY"
450 PRINT:PRINT
460 FOR J=1 TO 9
470 PRINT M(J);" ";
480 NEXT J
490 PRINT:PRINT
500 PRINT "PRESS <ENTER> TO CONTINUE"
510 WHILE INKEY$="":WEND
520 SOUND 5,100:SOUND 5,150:SOUND 5,100:
GOTO 30
530 REM --------------------
540 REM AMSTRAD MOVE
550 P=ASC("O")
560 X=0
```

```
570 J=1
580 IF A(W(J))=A(W(J+1)) AND A(W(J+2))=3
2 AND A(W(J))=P THEN X=W(J+2):GOTO 750
590 IF A(W(J))=A(W(J+2)) AND A(W(J+1))=3
2 AND A(W(J))=P THEN X=W(J+1):GOTO 750
600 IF A(W(J+1))=A(W(J+2)) AND A(W(J))=3
2 AND A(W(J+1))=P THEN X=W(J):GOTO 750
610 IF J<21 THEN J=J+3:GOTO 580
620 IF P=ASC("O") THEN P=ASC("X"):GOTO 5
70
630 REM - IF NO WIN/BLOCK MOVE FOUND -
640 REM - THEN THIS SECTION IS USED --
650 J=1
660 IF A(M(J))=32 THEN X=M(J):GOTO 750
670 IF J<9 THEN J=J+1:GOTO 660
680 H=0
690 H=H+1
700 X=INT(RND*9)+1:IF A(X)=32 THEN 750
710 IF H<100 THEN 690
720 R$="D":REM IT IS A DRAW
730 SOUND 5,260:RETURN
740 REM ----------
750 REM MAKE MOVE
760 A(X)=ASC("O")
770 COUNT=COUNT+1
780 D(COUNT)=X
790 FLAG=0
800 FOR J=1 TO 9
810 IF A(J)=32 THEN FLAG=1
820 NEXT J
830 IF FLAG=0 AND R$="" THEN R$="D"
840 REM IF ALL POSITIONS FULL, AND R$ IS
 NOT ASSIGNED, IT IS A DRAW
850 RETURN
860 REM ------------
870 REM WIN CHECK
880 J=1
890 IF A(W(J))=32 THEN J=J+3
900 IF J>23 THEN RETURN
910 IF A(W(J))=A(W(J+1)) AND A(W(J))=A(W
(J+2)) THEN 940
920 IF J<22 THEN J=J+3:GOTO 890
930 RETURN
```

```
940 IF A(W(J))=ASC("O") THEN R$="W":REM
 AMSTRAD WINS
950 IF A(W(J))=ASC("X") THEN R$="L":REM
 AMSTRAD LOSES
960 RETURN
970 REM -------------
980 REM HUMAN MOVE
990 PRINT:PRINT:SOUND 5,190
1000 PRINT TAB(5);"ENTER YOUR MOVE"
1010 MVE$="":MVE=0:WHILE MVE$<"1" OR MVE
$>"9":MVE$=INKEY$:WEND
1020 MVE=VAL(MVE$):PRINT TAB(10);MVE
1030 IF A(MVE)<>32 THEN 1010
1040 A(MVE)=ASC("X")
1050 RETURN
1060 REM ---------------
1070 REM PRINT BOARD
1080 SOUND 5,89:LOCATE 1,4
1090 PEN 3
1100 PRINT TAB(6);"1 : 2 : 3     ";:PEN 1
:PRINT CHR$(A(1));" : ";CHR$(A(2));" : "
;CHR$(A(3))
1110 PEN 3:PRINT TAB(6);"---------    ";:
PEN 1:PRINT " ----------"
1120 PEN 3:PRINT TAB(6);"4 : 5 : 6     ";
:PEN 1:PRINT CHR$(A(4));" : ";CHR$(A(5))
;" : ";CHR$(A(6))
1130 PEN 3:PRINT TAB(6);"---------    ";:
PEN 1:PRINT " ----------"
1140 PEN 3:PRINT TAB(6);"7 : 8 : 9     ";
:PEN 1:PRINT CHR$(A(7));" : ";CHR$(A(8))
;" : ";CHR$(A(9))
1150 CLS #1
1160 LOCATE 1,10:PEN 2:RETURN
1170 REM ----------------
1180 REM INITIALISATION
1190 MODE 1:BORDER 6:INK 0,6:INK 1,26:IN
K 2,24:INK 3,0:PAPER 0:PEN 1
1195 WINDOW #1,1,40,10,22
1200 DIM A(9):REM TO HOLD BOARD
1210 DIM M(10):REM FOR KNOWLEDGE BASE
1220 DIM W(24):REM WIN/BLOCK DATA
1230 DIM D(5):REM TO HOLD MOVES IN
                    CURRENT GAME
```

```
1240 REM WIN/BLOCK DATA
1250 FOR J=1 TO 24
1260 READ W(J):CHECKSUM=CHECKSUM+W(J)
1270 NEXT J:IF CHECKSUM<>120 THEN PRINT
"ERROR IN WIN/BLOCK DATA":END ELSE PRINT
 "WIN/BLOCK DATA OK"
1280 DATA 1,2,3,4,5,6,7,8,9
1290 DATA 1,4,7,2,5,8,3,6,9
1300 DATA 1,5,9,3,5,7
1310 REM INITIAL KNOWLEDGE BASE
1320 FOR J=1 TO 10
1330 READ M(J):CHECKSUM=CHECKSUM+M(J)
1340 NEXT J:IF CHECKSUM<>167 THEN PRINT
"INITIAL KNOWLEDGE BASE DATA ERROR":END
ELSE PRINT "INITIAL KNOWLEDGE BASE DATA
OK"
1350 DATA 2,6,8,4,7,3,1,9,5,2
1360 FOR J=1 TO 1000:NEXT J:CLS
1370 RETURN
```

If you wish to experiment with an automatic, random opponent, you might want to use the following one, which I used for this section of the book:

```
4500 REM RANDOM OPPONENT
4510 H=0
4520 H=H+1
4530 MVE=INT(RND*9)+1
4540 IF A(MVE)=32 THEN A(MVE)=ASC("X"):R
ETURN
4550 IF H<100 THEN 4520
4560 R$="D"
4570 RETURN
```

To trigger this unintelligent, tireless opponent, simply replace line 140 with GOSUB 4500.

Part Three — A Program Which Reasons

From a program which learns, we move to SYLLOGY, a program which reasons. Given two related statements, SYLLOGY is capable of deducing a third statement which contains information which was not explicitly stated.

The program works with syllogisms. A syllogism is a form of deductive argument. Aristotle worked out the rules which determine the validity of a syllogism. It generally takes the following form:

```
A is a B
C is an A
Therefore C is a B
```

The first two lines of a syllogism are propositions, while the third line is a conclusion.

```
A dog is an animal
An animal is furry
Therefore, a dog is furry
```

Before we discuss the program, and the background to it, in detail, we will show it at work. Ignore the material in parentheses before the conclusion, as this is included so that you can see the program actually working. You'll understand what this material is once you have followed through the explanation of the program.

The '?' prompt appears when SYLLOGY is waiting for an input. '> OK' appears when the program has accepted and understood your input.

```
? AN EAGLE IS A BIRD
     > OK

? A BIRD IS A WINGED CREATURE
     > OK

? IS AN EAGLE A WINGED CREATURE
        (LOOKING FOR EAGLE)
         ( FOUND AT 1 1 )
     > YES
```

As the program runs, it builds up a database of propositions, which it can refer to any time within that run. Here is the next pair of propositions we tried:

```
? A BIRD IS A FLYER
     > OK

? IS AN EAGLE A FLYER
        (LOOKING FOR EAGLE)
         ( FOUND AT 1 1 )
     > YES

? IS A FLYER A WINGED CREATURE
        (LOOKING FOR FLYER)
         ( FOUND AT 1 4 )
     > YES
```

SYLLOGY will accept, to add to its database, any statement of the following form:

```
A ... is a ...
```

This statement can include 'an' or 'the', as the language parsing is programmed to cope with them. Therefore, the following are valid, although the program cannot cope with a 'the' after the 'is' in the middle of the sentence:

```
An ... is a ...
The ... is an ...
```

The program goes into its 'deductive mode' if you start a sentence with 'is':

```
Is ... a ...
Is an .... a ....
```

If you simply press the ENTER key, without entering any input, the program will terminate (although it may be restarted, without loss of data, by GOTO 30).

Entering a question mark when the prompt appears will allow you to discover what SYLLOGY is holding in its memory, under each category heading it has created. After you enter the questionmark, the program will ask "SUBJECT TO CHECK?". At this point you enter the category heading you wish the program to investigate:

```
? ?
SUBJECT TO CHECK? BIRD
 2  2 EAGLE
 3  2 WINGED CREATURE
 4  2 FLYER

? ?
SUBJECT TO CHECK? EAGLE
 2  1 BIRD

? ?
SUBJECT TO CHECK? WINGED CREATURE
 2  3 BIRD

? ?
SUBJECT TO CHECK? FLYER
 2  4 BIRD
```

SYLLOGY will often produce surprising conclusions, which fly in the face of all the evidence we can bring to bear:

```
? TIM IS A FOOL
     > OK

? A FOOL IS AN IDIOT
     > OK

? IS TIM AN IDIOT
        (LOOKING FOR TIM)
         ( FOUND AT 1 1 )
     > YES

? ?
SUBJECT TO CHECK? TIM
 2  1 FOOL

? ?
SUBJECT TO CHECK? FOOL
 2  2 TIM
 3  2 IDIOT

? ?
SUBJECT TO CHECK? IDIOT
 2  3 FOOL
```

Although SYLLOGY can be tricked into some absurd conclusions, it generally is fairly robust:

```
? A CROW IS AN IDIOT
     > OK

? IS TIM A CROW
        (LOOKING FOR TIM)
         ( FOUND AT 1 1 )
     > NO

? IS A CROW A FOOL
        (LOOKING FOR CROW)
         ( FOUND AT 1 6 )
     > YES
```

SYLLOGY works with a two-dimensional string array, Z$, cross-referencing the propositions entered into it, and from this cross reference producing conclusions.

This is fairly easy to understand if you visualise what is happening as you enter statements. If we type in TIM IS A FOOL the program ignores the IS A and uses TIM as a file heading, and puts FOOL underneath that. A second statement of the type A FOOL IS AN IDIOT allows the program to open up a new file headed FOOL which has IDIOT underneath it. When the program is asked IS TIM AN IDIOT it first looks to see if it has a category called TIM. On finiding it has, it looks under that for the first subject filed. It comes across FOOL.

Now it looks to see if it has a category headed FOOL. On finding it has, it follows down through the subjects filed under this heading, and discovers the subject TIM. Because of this cross-referencing, it knows that the answer to the question IS TIM AN IDIOT is yes.

The same procedure, of course, occurs no matter which series of statements you feed into SYLLOGY. There is a lot of room in a 25 x 25 array such as we have with this program, and you may well wish to save your databases on some subjects.

The TIM IS AN IDIOT series was, of course, handled quite separately from THE EAGLE IS A BIRD series. To make it easy to understand how SYLLOGY files, and then accesses, the propositions upon which it reaches

conclusions, this is the internal storage arrangement for THE EAGLE IS A BIRD:

	1	2	3	4
1	EAGL	BIRD	WING.	FLYER
2	BIRD	EAGL	BIRD	BIRD
3		WING.		
4		FLYER		
5				

When the program encounters a new subject (the subject being the first noun in the proposition), it goes across the 'top' of the array, looking in turn at 1,1 then 1,2 then 1,3 and so on, for an unused space. So, you enter THE EAGLE IS A BIRD at the start of a run. 1,1 is vacant, so it stores EAGLE in 1,1 and BIRD under that in 2,1.

It then swaps the two nouns, and opens a category called BIRD which it places at 2,1 and underneath that files EAGLE (at 2,2). When it gets another statement which calls on a subject for which it has already set up a category, such as A BIRD IS A WINGED CREATURE, it stores the information WINGED CREATURE at 3,2 then opens a WINGED CREATURE file at 3,1 and stores BIRD underneath that.

And so it goes, cross-filing all the information it receives so that it can access it later. The final statement we entered for this run was A BIRD IS A FLYER, so SYLLOGY filed FLYER in the first available blank spot under BIRD (at 4,2) and opened a new category FLYER at 1,4 and stored BIRD underneath that at 2,4.

When you enter a question mark, to check the contents of a file, the computer simply goes to across the subject heading row (that is from 1,1 to 1,2 to 1,3 and so on) until it finds the subject. If it gets to the end (that is to 1,25) and does not find the subject, it will tell you it has no data stored on that subject. Having found the subject (such as BIRD at 1,2) it then works down the file, printing out the contents of each file. In this case, then, it would print out EAGLE, WINGED CREATURE and FLYER.

When it comes time to make a decision, on whether IS AN EAGLE A FLYER (decisions are triggered by the fact that the user input starts with the word OR) the program first looks across the top row to check whether or not it has any information stored on the first noun in the question. If it finds it has, SYLLOGY reports this to you (LOOKING FOR EAGLE FOUND AT 1,1) then looks down that row for the words stored under it. It finds BIRD (at 2,1) and then returns to the first row to find FLYER. It discovers it at 1,4 and scans down that row to find BIRD (at 2,4). It has now found a common link (BIRD) between the two words it is thinking about (EAGLE and FLYER) and can therefore conclude that the answer to the question IS AN EAGLE A FLYER is, in fact, YES. SYLLOGY then tells you what it has concluded.

The Program

In SYLLOGY, line 40 sends action to 910 if a question mark has been entered. Line 80 detects the 'IS' at the start of the input, indicating that the user is asking SYLLOGY to try and reach a conclusion. This sends action to 480, where the conclusion routine begins.

Lines 90, 100 and 110 strip THE, AN or A from the front of the input, so that A$ now begins with the noun which will be used to head a file.

The next routine, from 120 to 230, splits the input up into two words, with lines 120 to 160 getting the first noun, and triggering I DON'T UNDERSTAND (from line 170) if the input is not in accord with the specified format. Lines 180 through to 230 extract the second word. Line 190 checks to see if the phrase which is left after the first noun has been stripped starts with "W" and, if it does, assumes the centre word is 'WAS' ". This allows it to accept phrases such as:

```
THE DODO WAS A BIG BIRD
```

and

```
TIM IS AN IDIOT
```

Having extracted the important words (and having set B$ to the first one and C$ to the second) the program proceeds to store them in its database. Remember, this section of code is only used for 'laying down' information. Taking it up again is looked after by the 'reach a conclusion' section of the program.

The program next looks across the top of its file table, to see if (a) it already has a file on that subject, and if not (b) it has a space left in which to start such a file. If there is no space left, the message in line 310 I HAVE NO MORE SUBJECT STORAGE ROOM is triggered.

The next routine, from 320, is reached once the program has either discovered it already has a file (line 280) or has found room to create a file and has, in fact, done so (line 290).

There is no need for SYLLOGY to store EAGLE under BIRD more than once, even if the line AN EAGLE IS A BIRD is fed to the program more than once. Line 360 ensures that duplication definitions are not saved. Once the 'object' has been saved, the computer swaps subject and object (lines 420 through to 450) and then saves them the other way around. That is, if it saved EAGLE as a subject heading before, with BIRD underneath it, this time it saves BIRD with EAGLE as one of the file contents.

Now we come to the really interesting part (at least in terms of performance when SYLLOGY is running), the section which reaches conclusions. Firstly the leading IS is stripped from the input, along with A (line 510) or AN (line 520) if these are present (this means it can deal with IS AN EAGLE A BIRD as well IS TIM AN IDIOT). This section of code gets the first word, and sets it equal to F$. The next section extracts the second word, to set it equal to S$.

The program lets you know what it is looking for (printing up LOOKING FOR 'first word' in line 630) and if it finds it, tells you where in the table it was located (FOUND AT . . . in line 660). If it cannot find the second word it informs you of this (line 680) then returns to the main program. This line is triggered if, for example, you asked it IS TIM A GENIUS and it had not previously encountered the word GENIUS.

Our next section of code reaches conclusions. The first bit, from 700 to 730, says YES if the question you asked was exactly in the form you originally gave it some information. That is, if you had asked IS AN EAGLE A BIRD and earlier you had told it explicitly AN EAGLE IS A BIRD, this first part would discover this, and tell you YES.

The next section, from 710 right through to 800, searches to find the word using the method outlined earlier, reaching either a YES (line 850) or a NO (line 820) conclusion.

This final section is the one which lets you know what the program has stored under particular subject headings.

Now, here is the listing of SYLLOGY, so you can reach a few conclusions of your own.

```
10 REM SYLLOGY
20 GOSUB 1050:REM INITIALISE
30 PRINT:PEN 2:INPUT A$:A$=UPPER$(A$)
40 IF A$="?" THEN PEN 1:GOTO 910
50 IF A$="" THEN END
60 FLAG=0
70 REM NOTE THERE IS A SPACE BEFORE THE
      CLOSE QUOTE IN NEXT FOUR LINES
80 IF LEFT$(A$,3)="IS " THEN PEN 3:GOTO
480:REM CONCLUSIONS
90 IF LEFT$(A$,4)="THE " THEN A$=MID$(A$
,5)
100 IF LEFT$(A$,3)="AN " THEN A$=MID$(A$
,4)
110 IF LEFT$(A$,2)="A " THEN A$=MID$(A$,
3)
120 X=LEN(A$)
130 N=0
140 N=N+1
150 IF MID$(A$,N,1)=" " THEN B$=LEFT$(A$
,N-1):GOTO 180:REM EXTRACTS FIRST NOUN
160 IF N<X THEN 140
170 PEN 3:PRINT "I DON'T UNDERSTAND":GOT
O 30
180 K=4
190 IF MID$(A$,N+1,1)="W" THEN K=5
200 C$=MID$(A$,N+K):REM QUALIFYING
                          PHRASE
210 IF LEFT$(C$,2)="A " THEN C$=MID$(C$,
3):REM REMOVES ARTICLE
220 IF LEFT$(C$,3)="AN " THEN C$=MID$(C$
,4)
230 IF LEFT$(C$,4)="THE " THEN C$=MID$(C
$,5)
240 REM -- STORE INFORMATION --
250 REM -- FIRST CHECK TO SEE IF CAN
   FIND SUBJECT BEFORE FINDING BLANK --
```

```
260 N=0
270 N=N+1
280 IF Z$(1,N)=B$ THEN 320:REM SUBJECT
                          HEADING EXISTS
290 IF Z$(1,N)="" THEN Z$(1,N)=B$:GOTO 3
20
300 IF N<25 THEN 270
310 PEN 3:PRINT "I HAVE NO MORE SUBJECT
STORAGE ROOM"
320 REM - PROGRAM REACHES HERE WITH
          SUBJECT STORED AS HEADING -
330 REM - NOW PUT SUBJECT UNDER THIS -
340 K=0
350 K=K+1
360 IF Z$(K,N)=C$ THEN 400:REM INFO
    ALREADY STORED UNDER THAT HEADING

370 IF Z$(K,N)="" THEN Z$(K,N)=C$:GOTO 4
00
380 IF K<25 THEN 350
390 PEN 3:PRINT "I HAVE NO MORE OBJECT S
TORAGE SPACE":GOTO 30
400 IF FLAG=1 THEN PEN 3:PRINT TAB(6);">
 OK":GOTO 30:REM SWAP HAS BEEN DONE
410 REM -- NOW SWAP OBJECT AND SUBJECT
                  AND SAVE AGAIN --
420 FLAG=1
430 M$=B$
440 B$=C$
450 C$=M$
460 GOTO 250
470 REM ------------------------
480 REM     -- CONCLUSIONS --
490 REM -- FIRST SPLIT INPUT --
500 A$=MID$(A$,4):REM STRIP "IS"
510 IF LEFT$(A$,2)="A " THEN A$=MID$(A$,
3):REM STRIP "A" IF PRESENT
520 IF LEFT$(A$,3)="AN " THEN A$=MID$(A$
,4):REM STRIP "AN" IF PRESENT
530 REM -- GET FIRST WORD - F$ -
540 X=LEN(A$)
550 N=0
560 N=N+1
```

```
570 IF MID$(A$,N,1)=" " THEN F$=LEFT$(A$
,N-1):GOTO 600
580 IF N<X THEN 560
590 PEN 3:PRINT TAB(6);"I DON'T UNDERSTA
ND":GOTO 30
600 REM -- NOW GET SECOND WORD - S$ -
610 S$=MID$(A$,N+3)
620 IF LEFT$(S$,1)=" " THEN S$=MID$(S$,2
):REM STRIPS SPACE IF ARTICLE WAS "AN"
630 PEN 3:PRINT TAB(9);"(LOOKING FOR ";F
$;")"
640 X=0
650 X=X+1
660 IF Z$(1,X)=F$ THEN PEN 2:PRINT TAB(1
0);"( FOUND AT 1";X;")":GOTO 700
670 IF X<25 THEN 650
680 PEN 3:PRINT TAB(6);"> I CANNOT FIND
THE SUBJECT":PRINT TAB(8);F$
690 GOTO 30
700 Y=1
710 Y=Y+1
720 IF Z$(Y,X)=S$ THEN PEN 3:PRINT TAB(6
);"> YES":GOTO 30
730 IF Y<25 THEN 710
740 Y=1
750 Y=Y+1
760 P$=Z$(Y,X)
770 M=0
780 M=M+1
790 IF Z$(1,M)=P$ THEN 830
800 IF M<25 THEN 780
810 IF Y<25 THEN 750
820 PEN 1:PRINT TAB(6);"> NO":GOTO 30
830 Q=1
840 Q=Q+1
850 IF Z$(Q,M)=S$ THEN PEN 3:PRINT TAB(6
);"> YES":GOTO 30
860 IF Q<25 THEN 840
870 IF M<25 THEN 780
880 GOTO 820
890 REM ------------------------
900 REM -- CHECK CONTENTS
```

```
910 REM     OF PARTICULAR FILE --
920 INPUT "SUBJECT TO CHECK";H$:H$=UPPER
$(H$)
930 T=0
940 T=T+1
950 IF Z$(1,T)=H$ THEN 990
960 IF T<25 THEN 940
970 PEN 3:PRINT "I HAVE NO DATA STORED O
N ";:PEN 1:PRINT H$
980 GOTO 30
990 K=1
1000 K=K+1
1010 IF Z$(K,T)<>"" THEN PRINT K;T;Z$(K,
T)
1020 IF K<25 THEN 1000
1030 GOTO 30
1040 REM ------------------------
1050 REM INITIALISE
1060 MODE 1:BORDER 1:INK 0,1:INK 2,26:IN
K 1,24:INK 3,18:PAPER 0
1070 DEFINT A-Z
1080 DIM Z$(25,25)
1090 RETURN
```

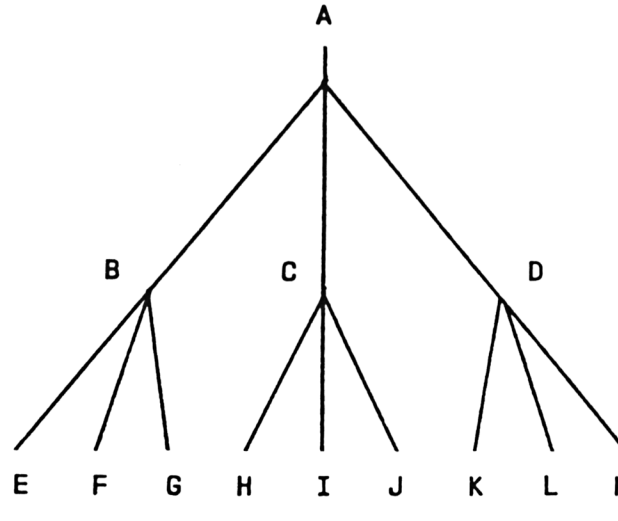
Part Four — Search Trees and Snickers

In this section of the book, we will develop a Draughts-like program called SNICKERS. We will use it to discuss some ideas of tree-searching, in which the computer behaves with a degree of intelligence by searching along lines of related options, and then from these chooses that which it judges to be the best action.

Searching through trees of options in this way is common to most problem-solving programs. Modifications, many of them most important ones, such as 'pruning' the tree to save following worthless branches at all, or to follow other branches to an unnecessary depth, are nearly always used in tree-searching to stop the process from taking an inordinate amount of time, but the basic idea of the tree search is still fundamental to problem-solving.

Why is it Called a Tree?

A search tree grows like any other tree, apart from being upside down. Take A in the following diagram as the starting point for the search. The 'branches' (labelled B, C and D) going off it represent valid decisions (or legal moves, if the program is tackling a game). The smaller branches radiating from these (E, F and so on) are implications of following that branch.



If the tree represents a move-finding mechanism in a chess game, for example, the A may represent the movement of a particular knight. The

program then follows through the implications of that move. B assumes, let us say, that moving this knight puts one of the opponent's pieces under attack. Response E is the opponent simply backing this piece away, F may be supporting the threatened piece with another one, and G may be capturing the offending knight. E, F and G would further split, into N, O . . . and so on, which would cover the possible responses to each action.

You can see that the search would rapidly escalate, and the options being considered would reach astronomical proportions, unless there was some means of guiding the search. Only in a very simple program, such as one which played Noughts and Crosses, could a program examine every branch of every tree, before choosing the best move.

For other programs, a branch can be examined to a pre-determined depth (and we'll be discussing depth shortly) instead of to the end, and the result of that examination stored.

'Parallel processing'

Another approach would be to examine a short distance down one branch, then back up and start another branch, and so on, and then examine the more promising branches to a greater depth. A branch, for example, which assumed the opponent in a chess game would sacrifice the queen to capture a pawn, would not merit further examination. Any branch which led the opponent — in the opinion of the program's evaluation mechanism — to weaken his or her position could be abandoned the moment this discovery was made, and processing time and effort put into following more promising leads.

When developing your own Al programs, it is worth starting to think about them in terms of search trees, as it is likely that they will involve this in some way. The tree may grow quite frighteningly, especially if you are not working in a tightly-restricted domain (such as we do in BLOCK-WORLD), or you are not too clear as to the criteria by which your program could be making choices.

We have developed SNICKERS for this section of the book, in order to demonstrate some aspects of primitive tree-searching. Naturally enough, you need to know how to play the game in order to understand the discussion about it. Each piece moves like a checkers' piece diagonally. Captures in SNICKERS are carried out in a familiar way, by leaping over an enemy piece into a vacant square beyond. However, in contrast to checkers, there are no multiple jumps in this game.

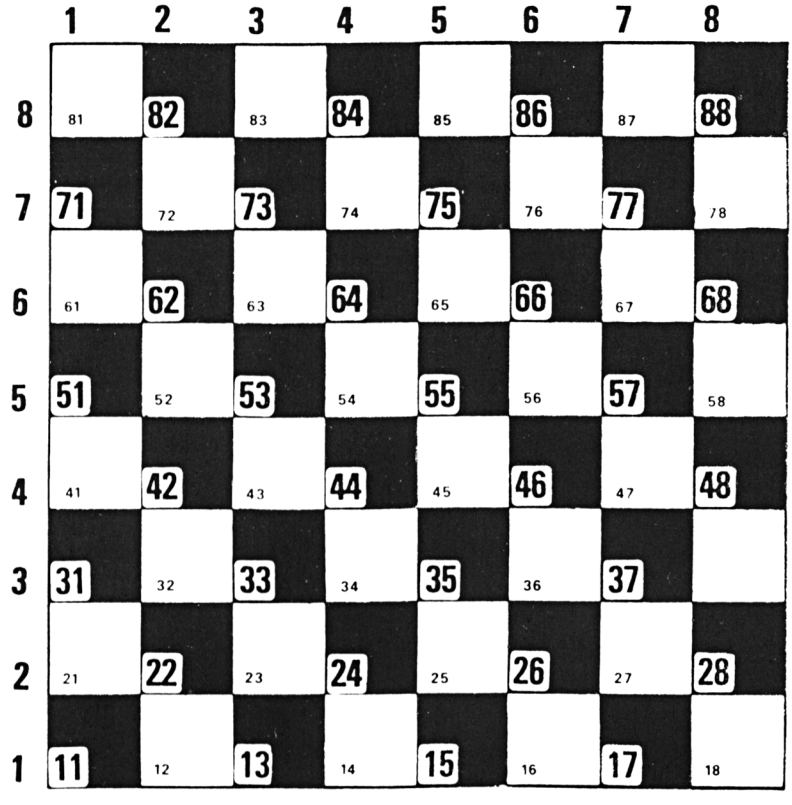
Vanishing Acts

The aim of the game is to get a score of five before the opponent does so. There are two ways to score a point. One way, predictably enough, is to capture an enemy piece. The other way is to reach the back row on the opposite side of the board. In checkers, this would result in the piece being 'crowned', or turned into a king with the ability to move backwards and forwards at will. In SNICKERS, the piece vanishes on reaching the opposite back row (which means, among other things, that you cannot have either kings in SNICKERS, nor pieces moving 'backwards' on the board).

If you leap over an enemy piece, and end up after that capture on the opposite back row, you'll get two points, rather than one. You'll see this occurring several times in most games. Your Amstrad will tell you moves it is considering at each point in the game, so you can see its machine intelligence at work. At the beginning of the game, as a moments' thought will show, there are just seven possible opening moves. The computer finds each legal move, then prints up the moves on the top of the screen, before making the move, as follows (with the numbers themselves being worked out by specifying the number down the edges of the board first, followed by the number across the top or bottom):

```
I AM CONSIDERING 71 TO 62
I AM CONSIDERING 73 TO 64
I AM CONSIDERING 73 TO 62
I AM CONSIDERING 75 TO 66
I AM CONSIDERING 75 TO 64
I AM CONSIDERING 77 TO 68
I AM CONSIDERING 77 TO 66
```

The numbers printed here by the computer refer to those within a master array which holds the board inside your Amstrad. At the top of the facing page you'll see a diagram of the board which the computer uses in SNICKERS.



You'll see that the numbering is not consecutive, and does not even start from one. However, this board is much easier to use, in computer terms, that is one in which only the black squares are numbered from one to thirty-two.

The Amstrad needs to know where the edges of the board are, and the 'missing' numbers supply it with that information. For example, if it tries to move from 48 to 59, the value held by element 59 in the array (zero, in the case of SNICKERS) will warn it that such a move is 'off the board'.

The second, and much more important, advantage lies in the consistency with which moves can be specified, no matter where on the board they occur. I'll explain what I mean by that. Look at the list of moves which the computer is considering to begin with, and notice the simple mathematical relationships connecting the square moved from, to that moved to:

71 to 62 -9	**73 to 62 -11**	**75 to 64 -11**
73 to 64 -9	**75 to 66 -9**	**77 to 68 -9**
		77 to 66 -11

The difference between the starting square, and the ending square, is either minus nine or minus eleven. And if you compare the numbers given above with the board, you'll see that moves downward and to the left are always minus eleven, and those downward and to the right are always minus nine.

This is true all over the board. Any non-capture move made by the computer must be minus nine or minus eleven from the starting square. This is, I'm sure you can appreciate, most convenient from the computer's point of view. (If you care to try the experiment using a board which simply has the black squares numbered from one to thirty-two you'll soon appreciate the grave problems this can cause.)

Furthermore, the Amstrad can make decisions fairly easily on this board. Assume the square the computer is on, is numbered X. If there is a human piece on X-9, and X-18 is empty, it knows it can capture by leaping into X-18. Its score can then be incremented, and X-9 turned into a blank square.

Furthermore, and this is where the 'intelligence' really comes in, the computer can look beyond that move, to see which one the human is likely to make next. If there is a human piece in X-27, the computer can assume — possibly rightly — that the human's next move will be to capture the computer piece now sitting on X-18, by moving into X-9.

The position after the capture (remembered, you as Amstrad are now on X-18) is also potentially under threat from X-25, if X-7 is vacant. This explanation is probably becoming a little bewildering at this point, so I suggest you try and follow it through on the board which was printed on page 73, or on a game board you have numbered in the same way.

The Amstrad can also sense when a piece of its own is under threat. Imagine, once again, that you are the computer on square X. The human moves into square X-9. You know that X+9 is vacant, so the human may well move into X+9 on his or her next move, capturing you on X. You could counter this by either moving a piece of your own into X+9, or — if this is not possible — moving a piece so that it threatens X+9. This may persuade the human player not to make the capture.

There is no equivalent of checkers 'huffing' in SNICKERS. You are under no obligation to capture a piece if you do not want to. You may prefer not to capture a threatened piece, knowing that you may have a chance of scoring two points with the threatening piece a little later, on another capture which would end on the back row.

Digging Deep

The SNICKERS tree search does not proceed very deeply, although it manages to play reasonably well, winning its fair share of games. You may well be tempted to think that, if a tree was set up and searched completely, the program would play perfectly.

SNICKERS is a less complex game than checkers, with no multiple jumps and no kings, so it is not too unreasonable to assume that a perfect system might be evolved. At the very least, it should be possible to create a path which the computer can follow to play the game extremely well.

We could do this by a method somewhat similar to the 'matchbox tic-tac-toe computer' discussed in the section on the program TIC-TAC. That is, we could examine every possible move, of every possible game, and analyse them in depth. After all, we have tireless computers at our disposal, and they could do the donkey work.

Think about it a little. You know (because the computer told you so a few pages back) that it had seven moves it could make at the start of the game. So our tree, with A at the top, starts with branches B, C, D, E, F, G and H, at the very first level. The human player similarly has seven moves from which to choose at the start of the game. Each of our initial branches now needs seven sub-branches (or 'nodes' as the branching points are called). After each player has had one move, and even before the program starts to look at possible responses to the human's first move, we have stacked up forty-nine divergent streams to follow.

The position gets worse. Now that one piece has moved out of the front row of each player's rank, two possible moves (one in some cases, if the initial move was at either end) are now available, plus another six (the first move has possibly blocked a move by a piece which is still on the front row). That means we have another 49 times 8 branches to consider, even before the human has had a second move.

A similar search tree for checkers would contain around 10 raised to the 40th power nodes. Considered at the rate of three million nodes per second (which would take a pretty nifty computer), this tree would take around 10 raised to the 21st power years to consider.

We suggested earlier that one way of pruning the tree would be to abandon unprofitable branches (such as any that imagines the opponent would

deliberately move into danger needlessly), to leave time and effort to examine more worthwhile branches. It was also suggested that the computer could check a certain distance into a branch, take note of what it had concluded, then swap to another branch, then another and another, with the option of abandoning branches which were becoming weaker, and concentrating on the more promising ones.

To do this, we have to be able to assign a value to the position found. This can be a number (based on something like the one for Samuel's checkers program — discussed in the TIC-TAC section of the book) or can be based on an hierarchical scheme to order moves chosen, and decide not to follow the majority of move branches which could be generated. As you'll see shortly, this is how we do it in the SNICKERS program.

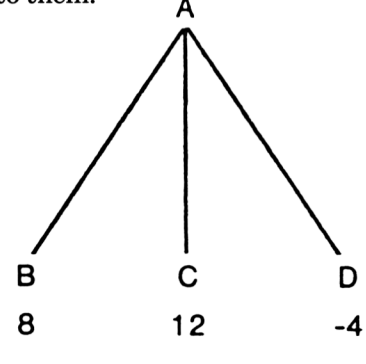
Mini-Maxing

However, we must first look a little further into search trees, in our quest for the perfect game-playing computer. SNICKERS uses a crude form of the technique known as 'mini-maxing' with which we can prune our relentlessly multiplying branches.

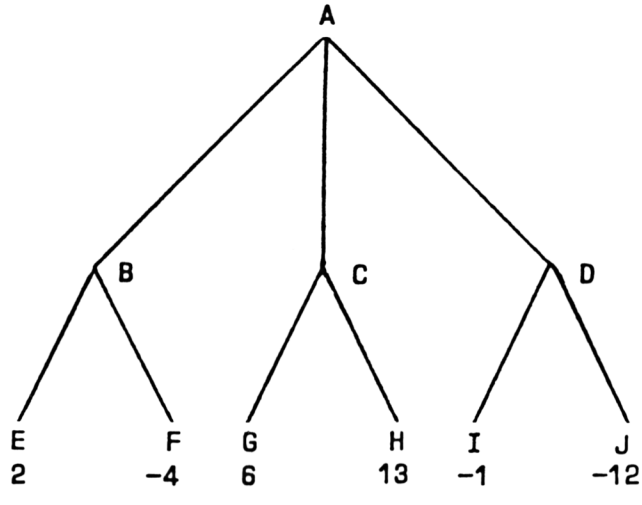
To use this, however, the computer should be able to assign numerical values to the positions it discovers.

Imagine that it has three options it is considering, and each option consists of a move by a different piece. The value given to the move could consist, in part, of how close to the centre the piece will be after the move, if it threatens (immediately, or could do so after another move) an enemy piece, if the square it is considering moving to is under threat, if the move actually makes a capture, or achieves some other goal (such as reaching the opposite back row).

Here is our tree, with moves B, C and D at the ends of the first three branches, with their scores next to them:



You can see that C has the highest value, so this node would seem the obvious choice. Remember, this little tree is based on the situation after the computer has moved. However, if the machine looks at the next series of branches, when the possible responses by the human player are considered and evaluated, it could see this:



The values given here for nodes E to J are assessed in terms of the *player's* evaluation of the board positions. The best move to be made by the computer could be the one which gives the human choices that will leave him or her in the weakest possible position. The choice then must be the one which gives the computer the maximum possible score while leaving the human choices which minimise his or her strength. This is where the term *mini-maxing* comes from.

Assuming the computer was not going to look further, to assess its own position after each of the moves which the player could make (and possibly assess player responses to that response), it may well be advised to choose move B. This leaves it in a fairly strong position (rating 8) although it does not leave it in the same position as move C would have done (rating 12).

The computer assumes the player will make the best move it can in the circumstances. Had the computer played C, to get a maximum rating immediately after the move, it would have left the human to play H, ending up with a rating of 13. Instead, by playing move B, the human can — at best — respond for a rating of 2, from node E.

I said earlier that SNICKERS works by assigning a value to each possible

move, in a hierarchy. It chooses its moves in reference to this hierarchy, which puts a value on the possible moves in the following order. It will always make a move which is higher up the tree if it can.

A degree of mini-maxing is present. The program thinks solely in terms of material advantage, that is, it seeks at all times to minimise the number of pieces the opponent has, and to preserve its own lives.

For example, the program may see two possible captures, one of which will subsequently expose it to capture and one which will not. Naturally enough, it will make the move which leaves it in the strongest position after the move (with the piece which has done the capturing still on the board) and ignore the move which enables the opponent to strengthen his or her position (by scoring a capture in return).

The hierarchy of moves used by SNICKERS to prune the 'possible moves' tree, and to save searching down branches which represent moves it is most unlikely to make, is as follows. Any moves found that fit the description are stored:

— Safe captures which further threaten human pieces, and do not expose another piece to capture.

— Captures which leave the pieces making the capture in complete safety.

— Other captures.

— Moves to protect pieces under threat.

— Random rejection of above moves, if the making of the move will expose a subsequent piece to capture.

— Non-capture moves onto the back row.

— Non-capture moves which do not expose the computer to danger.

— Any legal move.

If it finds any capture moves, it will not bother looking further down the tree. In effect, it automatically prunes branches with 'lower' nodes, by not even considering them. This may seem rash, and certainly means the program is unable to play with any kind of overall strategy, but it works

surprisingly well (aided, of course, by the simple nature of the game) in practice, and manages to play with an appearance of skill.

So you can appreciate, I hope, that this hierarchical ordering of moves, minimises the number of possibilities which must be explored. When examining the program, you'll see it first sweeps the board, square by square, looking for captures, which are subsequently stored as 'good, safe', 'safe' or 'captures'.

If the storage areas (dedicated areas) are empty at the end of this sweep, the computer sweeps the board again, looking to see if any of its pieces are under threat by human pieces.

If this search fails to find a move, our Amstrad looks to see if it has any pieces on the second back row which could be moved onto the back row, adding to its score. It has a predetermined order for doing this, ensuring that if two pieces are on the back row and can be moved, the one closest to the centre will be moved first, on the assumption that it is more likely to be under threat than a piece at the end. This is a rough-and-ready assumption which ensures the computer does not simply move the first piece it finds onto the back row.

If a move has not yet been made, the board is swept yet again, and any safe moves (that is, moves which do not expose the piece moved to capture) discovered are stored. If any have been found, the Amstrad chooses at random from these.

If this search has failed to find a move, the Amstrad picks locations on the board at random, looking for any legal move. If no move has been found in the 200 stabs it allows for finding one, the computer will concede the game. We will go through the main parts of the listing shortly, and identify the subroutines which carry out each of the tasks specified.

Incidentally, you may feel the multiple sweeps of the board are somewhat wasteful. Could the program not do all of its looking in a single sweep? The answer, of course, is 'yes', except it would mean a considerable waste of effort in many cases, when it would be looking for, and storing moves, which it had no intention of even considering. You may well, however, like to modify the program, or write one of your own, to do all the checks in a single sweep, and see what effect this has on its reaction time.

It is pretty obvious that the hierarchical system for determining the relative value of moves could be combined, for greater flexibility, with an

evaluation function. This could bring in things like Samuel did, such as the number of pieces on the board held by one player as opposed to the other, the number of pieces under direct threat, and 'control of the centre', however that may be defined.

One alternative to using an evaluation function would be for the computer to store all the possible board positions, and have each of these assigned a predetermined value. But this 'solution', like the idea of making complete trees covering all the possible outcomes of the game, runs up against the barrier of astronomically large numbers. There are around 10 raised to the 40th power possible board positions with checkers, and those of SNICKERS would approach the same order of magnitude.

There are no simple rules to apply when developing your own evaluation functions for board game programs you write. 'Informed guesswork' should guide your initial function, and then trying out the function in practice should allow you to modify it so that it performs well in practice. The advantage of a simple game, like noughts and crosses, is that the program can be set up to play repeatedly against an opponent playing randomly, or an intelligent one. The results of the games can be used to automatically modify the evaluation function, or a large number of games can be played with one version of the function, and compared with a similar number of challenges with modified functions. It is not so simple to program an opponent to play repeatedly against a program in a more complex game such as chess, checkers or even SNICKERS.

Weighted Elements

The task is made a little easier by the fact that the elements which make up an evaluation function are generally weighted with some factors within the function being multiplied by a higher value than others. Modification of the evaluation function may well then be a matter of modifying the weighting factors, rather than having to add or discard whole new elements.

I'll try to explain that last paragraph with a concrete example. Experience has shown chess players that the relative value of pieces can be expressed, in a rough and ready manner, as follows:

PAWN	**— 1**	**ROOK**	**— 5**
BISHOP	**— 3**	**QUEEN**	**— 9**
KNIGHT	**— 3.5**	**KING**	**— 128 ('infinite')**

You could create your first evaluation simply by adding up the pieces you

have, and subtracting the pieces your opponent has, to give a measure of your relative 'strength' as follows:

```
Strength  =  n*wP + 3*n*wB +  3.5*n*wK  +
5*n*wR  +  9*n*wQ  - (N*bP  +  3*N*bB  +
      3.5*N*bK + 5*N*bR + 9*N*bQ)
```

With this as a starting evaluation, you could possible write a rough chess program, which was willing to consider sacrificing pieces, or trading them, when your 'strength' was positive, and which would be most conservative in this regard when the 'strength' was negative. Playing against this program, and using this function in order to help decide which branches should be searched (using mini-maxing) could indicate that — in fact — the value of the rook has been underestimated, leading to unnecessary errors of judgement. You could then increase the value of a rook to say 6.5 or 7.

The worth of your evaluation function could be increased if mobility (possibly expressed as the number of moves each piece has) could be incorporated. The value of the rook, for example, could be expressed (with rm equal to the moves it could make) as 5*n*wr+3*rm. The function could be further elaborated by adding a number to the value of the piece which reflected the 'value' of the square it was occupying (with the central four squares worth, say, 8 each, the squares surrounding the central four worth 6.5 and the next set worth 4). And so on. Thinking about the problems inherent in creating an evaluation function for a game as complex as chess indicates clearly that such a task is not a trivial one.

If you are interested in developing evaluation functions, you might like to start with one for SNICKERS, and use it to modify the way moves are chosen. You should find that even a crude function — if you can get the computer to apply it in practice — should improve the computer's play to a noticeable extent.

It would be possible, given nearly infinite time and computing power, to search each branch of the tree until the end of the game was reached. This would mean investigating an enormous number of possibilities, as you shall see in a moment. A more sensible approach, perhaps, would be to limit the depth of search. Let's assume, for now, that we have deliberately decided to follow the tree for just two steps, one move and the opponent's possible answers to that move.

A search of this kind is called '2-ply' because we are looking to a depth of one move and the immediate response to that move. In a rough way, SNICKERS uses a kind of 2-ply search (but without overall mini-maxing)

trying for the move which gives it the best material advantage, assuming the opponent plays his or her best move in material terms in response (that is, the opponent captures if this is possible). Assuming your evaluation function is realistic, the deeper the ply, the better the results your program should achieve.

However, astronomical numbers come into play again as we increase the depth of search. If we assume, in noughts and crosses, that there are three possible moves at the start of a game (that is, a move in one corner is equal to a move in any corner, as the first board can be transformed into the others by rotation), there are twelve positions at the 2-ply level, and a number approaching 12*7 ('approaching', because not all these games would be played out to completion, as a draw or win would be evident before all nine positions were filled) at the next level.

In other games, the possibilities increase even more dramatically. An average 4-ply search in chess, for example, has to cope with around a million possibilities.

The Alpha-Beta Algorithm

How can we possibly cope with all these numbers, in an attempt to write a program which plays reasonably well, but which does not take 10 raised to the 40th power years to make a move? It is time now to introduce the alpha-beta algorithm, a very useful aid in trimming branches in our search tree.

The alpha-beta idea is simple, but powerful. It says that — if you can choose from a set of possible moves — once you have found one move which suits your needs (and your needs could well be expressed in terms of improving the score produced by your evaluation function), there is no need to look for another move in that set.

The alpha-beta algorithm is so named because it operates simply by keeping track of two values, called alpha and beta. Our program is searching through a tree, looking for a good move. Alpha is the value of the best move it has so far discovered. As the search continues, the program finds a move which produces a lower value than alpha. It knows immediately it is not worth following that branch, because it would lead to a worse result than the best one found so far. This means the computer is free to continue searching, on a new branch.

Meanwhile, the program is also working out the possible responses to its

moves. If it finds a response which is bad from the opponent's point of view — so the opponent would be unlikely to make it — there is no point in following the situations which could arise from that response. Beta is the value which the opponent has when making his or her best response to a computer move. The search is discontinued if the branch leads to an opponent move which would diminish the value of beta, seen from the player's point of view.

The search cut-off caused by discovering the path being investigated, that lowers the computer's score is called an 'alpha-cutoff'. The other search terminator is called, naturally enough, a 'beta-cutoff'.

We can see a crude form of the alpha side of this algorithm in action in the following sequence of events:

— Measure the value of the current board.

— Find the first move.

— Measure the value of the board after that move.

— Find the best opponent response, and work out what the board would be worth after that move.

— Record both values.

— Find the next move, and follow the process.

— If the new move gives a better mini-max result, discard the first move, but store the second.

— Continue testing moves in this way, keeping a record only of the move found which gives the best mini-max so far.

Doing this would mean you would end up with a single move which — given the limited look ahead — would be the 'best' one to make.

Note that the alpha-beta algorithm can be applied in many decision-making areas, other than in board games. Many intelligent programs, faced with a choice between a number of options, follow an alpha-beta line in determining which is the best choice of action.

How the Program Works

Like the other programs in this book, SNICKERS is built around a major loop, which is recycled over and over again until a particular condition is satisfied. Within that loop is a number of sub-routine calls.

The action first goes to the INITIALISE routine, from line 2070. Here, several arrays are dimensioned. These are as follows:

A — to hold the board and the 'off the board' squares surrounding it.

G — to act as store for 'good, safe capture' moves found during a sweep.

S — as G, excpt the captures stored here are less desirable, being defined as 'safe'.

T — this holds captures which are not classed by the program as either 'good, safe' nor 'safe'.

The REM statements identify the variables that are assigned here, with E representing an empty white square, B the empty black square (shown on the display as a dot), C the computer piece and H the human piece. It makes sense to use variable names which will remind you of what the variable stands for, as we have in this case. HS holds the human score, and CS the computer score.

Lines 2210 to 2260 read the initial board configuration into the A array.

Our main cycle gives an indication of how the computer proceeds from this point. We will not look at how the board is printed, nor how human moves are accepted because these are trivial programming problems.

When the computer looks for its move, it follows — as we pointed out earlier — a strict hierarchy of moves. The program sets three variables, which are used each time the program cycles, to zero with lines 220, 230 and 240. The REM statements explain them.

Now the computer begins its first sweep of the board, jumping over the evaluation process (see line 320) if the square under consideration does not contain one of its own pieces. It may be worthwhile following the whole of this capture sequence through in detail. The REM statements explain the code fairly thoroughly.

Note how the proposed move is stored in line 820 as a single number. The

result of this manipulation is to produce a four-figure number, with the first two digits representing the 'from' square (or START as it is called in several places in the program) and the final two digits representing the 'to' square. The four-digit number is decoded, and the move made, by the routine from 1510.

If the program has found a 'good, safe' move (or more than one) it plays this move, and then allows the human to move. If it has not found a 'good, safe' move, but does have a 'safe' one, it plays that. Failing this, a 'capture' move will be played. If none of these are possible, the program then goes to the next element in its hierarchy, moving to protect a piece which is under threat from the human player.

If such a move is found by line 1040, the next line will check to see that this move does not expose another piece to danger. If it does, the proposed move will be rejected around 50% of the time. This is hardly a sophisticated mechanism for making a choice but it ensures the computer does not always blindly move to protect a piece (a blindness which could be discovered and exploited by a human player), and also tends to make each game played by the program different from other ones.

Moving a piece onto the back row carries the same reward as capturing a piece, so the next item in the hierarchy is to make a move onto the back row if that is possible. The routine from 1140 looks after this. The sequence of squares checked in this section means those in the middle squares will move into the back row sanctuary before those at the end:

If all these have failed, SNICKERS tries to find a legal move. It chooses up to 200 moves at random (counting them with variable L) and if it cannot find a move in this time, concedes the game with line 1710. If this is not possible, the program sweeps to find a legal move which will not place it in danger. The moves are counted by the variable CMOVE, and the actual move to be made is chosen by line 1500. If all these have failed, SNICKERS tries to find a legal move. It chooses up to 200 moves at random (counting them with variable L) and if it cannot find a move in this time, concedes the game with line 1710.

Here's the listing of SNICKERS:

```
10 REM SNICKERS
20 GOSUB 2070:REM INITIALISE
30 GOSUB 1760:REM PRINT BOARD
40 REM -- MAIN CYCLE STARTS --
```

```
50 GOSUB 190:REM AMSTRAD MOVES
60 GOSUB 1760:REM PRINT BOARD
70 IF CS>4 THEN 120
80 GOSUB 1950:REM ACCEPT HUMAN MOVE
90 GOSUB 1760:REM PRINT BOARD
100 GOSUB 2390:IF HS<5 THEN 50
110 REM ------------
120 REM -- END OF GAME --
130 PEN 4:LOCATE 1,1:PRINT "THE GAME IS
OVER"
140 REM DECLARE VICTORY
150 IF HS>CS THEN PRINT "YOU HAVE WON"
160 IF CS>HS THEN PRINT "I'M THE WINNER"
170 LOCATE 1,20:PEN 1:END
180 REM -----------------
190 REM   AMSTRAD MOVES
200 REM -----------------
210 REM SEARCH FOR CAPTURES
220 GSAFE=0:REM TO SEARCH FOR GOOD, SAFE
 CAPTURES WHICH THREATEN HUMAN PIECES
230 CSAFE=0:REM TO COUNT SAFE CAPTURES
 WHICH DO NOT PLACE AMS. UNDER THREAT
240 CCAPTURE=0:REM TO COUNT OTHER
                    CAPTURES FOUND
250 FOR J=1 TO 3
260 G(J)=0:REM EMPTY 'GOOD SAFE CAPTURE'
                  STORAGE
270 S(J)=0:REM EMPTY 'SAFE CAPTURE'
                       STORE
280 T(J)=0:REM EMPTY 'OTHER CAPTURES'
                       STORE
290 NEXT J
300 FOR J=80 TO 30 STEP -10
310 FOR K=1 TO 8
320 IF A(J+K)<>C THEN 390:REM SKIP
 EVALUATION IF NO AMSTRAD PIECE HERE
330 REM -- CAPTURE TO RIGHT --
340 X=J+K-9:Y=J+K-18:Z=J+K-27:M=-11
350 IF A(X)=H AND A(Y)=B THEN GOSUB 700:
REM CAPTURE FOUND
360 REM -- CAPTURE TO LEFT --
370 X=J+K-11:Y=J+K-22:Z=J+K-33:M=-9
380 IF A(X)=H AND A(Y)=B THEN GOSUB 700:
REM CAPTURE FOUND
```

```
390 NEXT K
400 NEXT J
410 IF GSAFE+CSAFE+CCAPTURE=0 THEN 980:R
EM NO CAPTURES FOUND
420 REM - NOW CHOOSE CAPTURE TO MAKE -
430 PEN 4:LOCATE 1,1:PRINT ">>CAPTURE FO
UND"
440 FOR TT=1 TO 1000:NEXT TT
450 IF GSAFE<>0 THEN 500
460 IF CSAFE<>0 THEN 670
470 REM - CHOOSE FROM GENERAL CAPTURES
480 MVE=T(INT(RND*CCAPTURE)+1)
490 GOTO 540
500 REM  -- CHOOSE FROM GOOD SAFE --
510 REM -- SELECT FROM STORED MOVES --
520 MVE=G(INT(RND*GSAFE)+1)
530 REM -- MAKE MOVE --
540 START=INT(MVE/100)
550 ED=MVE-100*START
560 A(START)=B
570 A(START-ED)=B
580 A(START-2*ED)=C
590 CS=CS+1
600 REM -- CHECK FOR ADDITIONAL SCORE
              IF LANDING ON BACK ROW --
610 IF START-2*ED>18 THEN RETURN
620 A(START-2*ED)=B
630 CS=CS+1:FOR JJ=1 TO 5:GOSUB 2390:NEX
T JJ
640 PEN 4:LOCATE 1,1:PRINT "CAPTURED AND
 LANDED":PRINT "ON"START-2*ED"ON BACK RO
W"
650 FOR TT=1 TO 2000:NEXT TT
660 RETURN
670 REM -- SAFE CAPTURE --
680 MVE=S(INT(RND*CSAFE)+1)
690 GOTO 540
700 REM -- CHECK PROPOSED CAPTURE
                    FOR SAFETY --
710 REM -- CHECK SQUARE BELOW IN
                    SAME DIRECTION --
720 PEN 4:LOCATE 1,1:PRINT J+K"TO"Y:PRIN
T "CAPTURING ON"X
```

```
730 FOR TT=1 TO 900:NEXT TT
740 IF A(Z)=H THEN 920:REM STORE AS A
                        NON-SAFE CAPTURE
750 REM -- CHECK SQUARE IN OTHER
       DIRECTION FROM INTENDED MOVE --
760 IF A(Y+M)=H AND A(Y-M)=B THEN 920
770 REM -- NOW CHECK TO SEE IF WILL
 LEAVE PIECE EXPOSED BY MAKING MOVE --
780 IF A(J+K+M)=C AND A(J+K+2*M)=H THEN
920
790 REM - IF REACHED THIS POINT, CAPTURE
             IS 'SAFE' --
800 REM -- STORE THIS MOVE --
810 CSAFE=CSAFE+1
820 S(CSAFE)=100*(J+K)+20+M:REM THIS
  ENCODES ENOUGH INFO TO RECREATE MOVE
830 REM -- NOW SEE IF THIS DESERVES TO
      BE CALLED A 'GOOD SAFE' MOVE --
840 CHECK=GSAFE
850 IF Y+2*M<1 THEN RETURN
860 IF A(Y+M)=H AND A(Y-(20+M))<>B AND A
(Y+2*M)=B THEN GSAFE=GSAFE+1
870 IF CHECK=GSAFE THEN RETURN:REM THIS
 MOVE FOUND NOT TO BE 'GOOD SAFE'
880 REM -- STORE GOOD SAFE MOVE --
890 PEN 4:GOSUB 2390:LOCATE 1,1:PRINT "C
ONSIDERING"J+K"TO"M+20+J+K:FOR TT=1 TO 4
00:NEXT TT
900 G(SAFE)=100*(J+K)+20+M
910 RETURN
920 REM -- STORE NON-SAFE CAPTURES --
930 CCAPTURE=CCAPTURE+1
940 PEN 4:LOCATE 1,1:PRINT "CONSIDERING"
J+K"TO"M+20+J+K:FOR TT=1 TO 400:NEXT TT
950 T(CCAPTURE)=100*(J*K)+20+M
960 RETURN
970 REM ----------------------
980 REM -- MOVE TO PROTECT PIECE
                        UNDER THREAT --
990 MVE=0
1000 J=80
1010 K=1
1020 Q=J+K
```

```
1030 IF A(Q)<>C THEN 1110:REM DO NOT
 CONSIDER THIS SQUARE, NO COMP. PIECE
1040 IF A(Q+9)=B AND A(Q-9)=H AND A(Q+18
)=C THEN MVE=100*(Q+18)+Q+9
1050 REM -- RANDOM REJECTION OF THIS
 MOVE IF IT EXPOSES A SUBSEQUENT ONE --
1060 IF MVE<>0 AND A(Q-2)=H AND A(Q+20)=
B AND RND>0.5 THEN 1510
1070 IF A(Q+9)=B AND A(Q-9)=H AND A(Q+20
)=C THEN MVE=100*(Q+22)+Q+11
1090 IF MVE<>0 AND A(Q+2)=H AND A(Q+22)=
B AND RND>0.5 THEN 1510
1100 IF A(Q+11)=B AND A(Q-11)=H AND A(Q+
20)=C THEN MVE=100*(Q+20)+Q+11:GOTO 1510
1110 IF K<8 THEN K=K+1:GOTO 1020
1120 IF J>10 THEN J=J-10:GOTO 1010
1130 REM ------------------------
1140 REM -- NO CAPTURE FOUND, SO LOOK
FOR MOVE TO 'DISAPPEAR' ON BACK ROW --
1150 MVE=0
1160 REM - LESS DESIRABLE MOVES CHECKED
FIRST SO CAN BE OVERWRITTEN BY BETTER -
1170 IF A(22)=C AND A(11)=B THEN MVE=22
1180 IF A(28)=C AND A(17)=B THEN MVE=28
1190 IF A(22)=C AND A(13)=B THEN MVE=22
1200 IF A(26)=C AND A(17)=B THEN MVE=26
1210 IF A(26)=C AND A(15)=B THEN MVE=26
1220 IF A(24)=C AND A(15)=B THEN MVE=24
1230 IF A(24)=C AND A(13)=B THEN MVE=24
1240 IF MVE=0 THEN 1310
1250 PEN 4:LOCATE 1,1:PRINT "ONTO BACK R
OW FROM"MVE
1260 FOR TT=1 TO 2000:NEXT TT
1270 A(MVE)=B
1280 CS=CS+1
1290 RETURN
1300 REM ----------------------------
1310 REM -- SAFE, NON-CAPTURE MOVES --
1320 CMVE=0:REM COUNT MOVES FOUND
1330 FOR J=80 TO 30 STEP -10
1340 FOR K=1 TO 8
1350 IF A(J+K)<>C THEN 1460
1360 X=J+K-9:Y=J+K-18:Z=J+K-20
```

```
1370 Q=J+K+2
1380 IF A(X)<>B THEN 1460
1390 IF A(Y)=H OR A(Z)=H OR A(Q)=B THEN
1460
1400 GOSUB 1560
1410 X=J+K-11:Y=J+K-22:Z=J+K-20
1420 Q=J+K-2
1430 IF A(X)<>B THEN 1460
1440 IF A(Y)=H OR A(Z)=H AND A(Q)=B THEN
 1460
1450 GOSUB 1560
1460 NEXT K
1470 NEXT J
1480 IF CMVE=0 THEN 1630
1490 REM -- MAKE MOVE --
1500 MVE=T(INT(RND*CMVE)+1)
1510 START=INT(MVE/100)
1520 ED=MVE-100*START
1530 A(START)=B
1540 A(ED)=C
1550 RETURN
1560 REM -- STORE MOVES --
1570 CMVE=CMVE+1
1580 PEN 4:LOCATE 1,1:PRINT "CONSIDERING
"J+K"TO"X
1590 FOR TT=1 TO 800:NEXT TT
1600 T(CMVE)=100*(J+K)+X
1610 RETURN
1620 REM --------------------
1630 REM RANDOM NON-CAPTURE MOVE
1640 PEN 4:LOCATE 1,1:PRINT "RANDOM, LEG
AL MOVE?"
1650 L=0
1660 L=L+1
1670 J=10*INT(RND*8+1)
1680 K=INT(RND*8+1)
1690 IF A(J+K)=C THEN 1720
1700 IF L<200 THEN 1660
1710 PEN 4:LOCATE 1,1:PRINT "I CONCEDE T
HE GAME":LOCATE 1,22:END
1720 IF A(J+K-9)=B THEN MVE=100*(J+K)+J+
K-9:GOTO 1510
1730 IF A(J+K-11)=B THEN MVE=100*(J+K)+J
+K-11:GOTO 1510
```

```
1740 GOTO 1700
1750 REM ---------------
1760 REM PRINT BOARD
1770 GOSUB 2390:LOCATE 1,1:PRINT "
                                  "
1780 PRINT "
          "
1790 PEN 1:PRINT "AMSTRAD:"MID$(STR$(CS)
,2);:PEN 2:PRINT " HUMAN:"MID$(STR$(HS),
2)
1800 PRINT
1810 PEN 3:PRINT TAB(6);"12345678"
1820 PRINT TAB(6);"--------"
1830 FOR J=80 TO 10 STEP -10
1840 PEN 3:PRINT TAB(3);J/10;
1850 FOR K=1 TO 8:F=A(J+K)
1860 PEN -(F=230)-2*(F=231)-3*(F=207)
1870 PRINT CHR$(F);
1880 NEXT K:PEN 3:PRINT J/10
1890 NEXT J
1900 PRINT TAB(6);"--------"
1910 PRINT TAB(6);"12345678"
1920 REM NOW BACK TO MAJOR LOOP
1930 RETURN
1940 REM ---------------------
1950 REM ACCEPT HUMAN MOVE
1960 LOCATE 1,1:PRINT "MOVE FROM";:PEN 4
:INPUT START:PEN 3:GOSUB 2390:IF START<1
1 OR START>88 THEN 1960
1970 IF A(START)<H THEN 1960
1980 LOCATE 1,1:PRINT "FROM"START" TO";:
PEN 4:INPUT ED:PEN 3:LOCATE 1,1:PRINT "
                    ":GOSUB 2390:IF ED<11 O
R ED>88 THEN 1980
1990 IF A(ED)<>B OR ABS(START-ED)>11 AND
 A((START+ED)/2)<>C THEN 1980
2000 A(START)=B
2010 A(ED)=H
2020 IF ABS(START-ED)>11 THEN A((START+E
D)/2)=B:HS=HS+1:LOCATE 1,1:PRINT "WELL D
ONE!":FOR TT=1 TO 800:NEXT TT
2030 PEN 4:IF ED>80 THEN A(ED)=B:HS=HS+1
:LOCATE 1,1:PRINT "ONE MORE FOR YOU!"
```

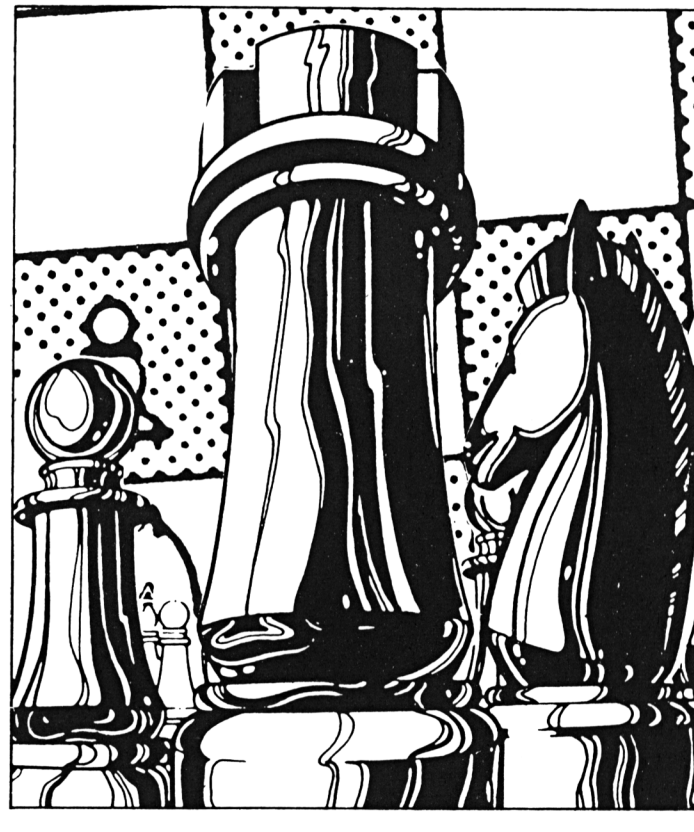
```
2040 FOR TT=1 TO 700:NEXT TT
2050 RETURN
2060 REM ---------------------
2070 REM INITIALISE
2080 DEFINT A-Z
2090 MODE 0:INK 0,17:BORDER 17:INK 1,20:
INK 2,24:INK 3,0:INK 4,18,14:SPEED INK 5
,5
2100 RANDOMIZE TIME:GOSUB 2390
2110 DIM A(110):' BOARD AND BLANK
      SPACES AROUND AND BEYOND IT
2120 DIM G(3):' GOOD SAFE CAPTURE STORE
2130 DIM S(3):' SAFE CAPTURE STORE
2140 DIM T(18):'OTHER CAPTURE STORE,
 ALSO FOR SAFE, NON-CAPTURE MOVES
2150 E=207:' BLACK SQUARES
2160 B=32:'  EMPTY
2170 C=230:' AMSTRAD PIECES
2180 H=231:' HUMAN PIECES
2190 HS=0:'  HUMAN SCORE
2200 CS=0:'  AMSTRAD SCORE
2210 REM -- STEP UP BOARD --
2220 FOR J=10 TO 80 STEP 10:GOSUB 2390
2230 FOR K=1 TO 8
2240 READ A(J+K):CHECKSUM=CHECKSUM+A(J+K
)
2250 NEXT K:NEXT J
2260 IF CHECKSUM<>10824 THEN PRINT "DATA
 ERROR":END
2270 RETURN
2280 REM ------------------------
2290 DATA 231,207,231,207,231,207,231,20
7
2300 DATA 207,231,207,231,207,231,207,23
1
2310 DATA 32,207,32,207,32,207,32,207
2320 DATA 207,32,207,32,207,32,207,32
2330 DATA 32,207,32,207,32,207,32,207
2340 DATA 207,32,207,32,207,32,207,32
2350 DATA 230,207,230,207,230,207,230,20
7
```

```
2360 DATA 207,230,207,230,207,230,207,23
0
2370 REM ----------------
2380 REM SOUND
2390 SOUND 5,RND*100+10
2400 RETURN
```

Part Five — The Wider Value of Games

It was argued, in the earliest days of Al research, that game-programming was not a worthy pursuit. It was suggested that the effort being put into chess-playing algorithms, for example, could better be spent on devices to prove mathematical theorems or on programs which modelled the way (to the extent it was understood at that time) the human brain operated.

But the means by which a brain arrives at a solution to a complex problem



— such as that presented by a chess board in mid-game — has been of continual fascination. Long before computers (as we understand them) existed, men were thinking aout how a chess program could be written.

Back in 1949, Claude Shannon (whose work with relays and logic is

discussed in the LEARNING AND REASONING section of this book), while working at Bell Telephone Laboratores, presented a very important paper at a New York convention. It was called *Programming a Computer for Playing Chess.* The value of this paper far transcends its historic importance as the first published work on the subject. A significant number of the concepts Shannon discussed in that paper are still used in present-day chess programs.

What was more, Shannon saw that if the problems of programming a computer to play chess could be solved, the insights gained could be of great value in helping machines develop expertise in other fields wher problems of similar complexity existed. He listed some of these: the design of electronic circuits, complicated telephone switching situations, language translation and problems of logical deducation.

Those who sneered at attention being put into making game-playing machines missed the point. Any advance in Al expertise is potentially a source of information which will assist in other areas of Al application. Earlier, we looked at the program TICTAC. It is not very significant, on its own, to have a program which teaches itself to play better Noughts and Crosses. But the actual **idea** of learning is very important.

Real-World Complexities

There are many situations in the world which are the product of a bewildering array of factors. Far too many factors have led to the present situation to enable it to be easily comprehended by man. And, if the situation is changing (as all real-world situations do) the ability of man to keep up with the present position, in order to make the most reasonable decisions as to what to do, iks almost impossible.

Here is where game-playing computers can help. The expertise gained from writing an evaluation function in chess (an evaluation function assesses the overall strength or weakness of one side of the game, in terms of a number of factors, including the number of pieces on the board, their nature and position, the other squares they attack, and so on) could well be applied in producing an evaluation function to suggest the best steps to overcome problems such as smog, or the disposal of nuclear waste.

Consider the situation when the Three Mile Island nuclear reactor malfunctioned. The number of variables to be considered was beyond the

ability of the human operators, as the Malone Committee Report on the accident pointed out:

> *... the operator was bombarded with displays, warning lights, print-outs and so on to the point where the detection of any error condition and the assessment of the right action to correct the condition was impossible...*

A computer expert which could cut through all the input to pinpoint what was important, and suggest a course of action, would have been invaluable in that situation.

It seems probable, then, that the expertise gained from working on such programs as one is to play chess, can produce payoffs in other areas of Al development.

The advances gained in this way are not always as might be predicted. For example, chess programs have been written which (a) try to emulate the way human beings play chess; and (b) simply try to play as well as possible. It has been found that programs which seek to act like human players do not, on the whole, play as well as machines acting in their own best interests.

There are two lessons from this. One is that attempting to model human thinking patterns onto a machine may not be the best routine to follow to elicit the highest possible levels of Al performance. The second is that, from attempting to produce a program which behaves like a human being, we can gain some genuine insights into the way human minds behave.

Other Games, Other Lessons

Of course, chess was not the only game in town in the early days of work on artificial intelligence. For example, checkers and tic-tac-toe were other, early candidates for attention.

Earlier, we discussed the work of Arthur Samuel on developing a checkers program which could learn as it played. Samuel had no appreciation of the problems involved in writing a checkers program when he first began. he told Pamela McCorduck (in *Machines Who Think,* San Francisco: W. H. Freeman and Co., 1979; pp. 148, 149) that his checkers program began in 1946 when — after working for Bell — he went to teach at the University of Illinois.

He decided the university needed a computer, but even the $110,000 the university's board of trustees came up with was not enough to buy a machine. Samuel concluded that the only way they could get a machine would be to use the money to build one themselves. He thought that, if he could do something spectacular with the first machine they planned to build, a small one, the exposure they got would enable them to attract government funds to add to those provided by the trustees. Samuel says he thought that checkers was a fairly trivial game, which would be easily programmed. Once the program was written, they would use it to defeat the current world checkers champion in a forthcoming championship in Kankakee, a nearby town, and from the publicity that would generate, they could get other funds.

The magnitude of the task soon became apparent. By championship time, not even the computer — much less the checkers program — was complete.

Samuel says he thought of checkers because he knew other groups were working on chess. In comparison with chess, he regarded checkers as a trivial game. But, as you can see from the LEARNING AND REASONING section of this book, even programming a computer to play Noughts and Crosses has its own difficulties.

If Noughts and Crosses is not trivial, think of a game such as Go. Much effort, throughout the history of artificial intelligence, has gone into designing chess programs, but relatively little into Go.

There are three reasons for this. One is purely cultural. Most of us in the West don't play Go, but nearly all of us have at lest a passing acquaintance with chess. The second reason is historical. The earliest workers in the field, such as Turing and Shannon, highlighted chess as an area worth exploring. And the third reason, pointed out forcefully by J. A. Campbell (in 'Go', his contribution to *Computer Game-Playing, Theory and Practice,* edited by M. A. Bramer, Chichester, West Sussex: Ellis Horwood Ltd., 1983; p. 136) is that it has proved extremely difficult to write a program which plays even as well as a raw recruit to the game.

While games such as Othello — where the relative values of various squares on the board can be fairly easily tabulated — may respond well to brute-force search techniques, it has been suggested that 'Go' will only respond to a less hamfisted approach. Indeed, 'Go' may well take the place of chess as the ultimate test for Al (see David Brown, *Seeing is Believing,* op. cit., p. 177).

Part Six — Understanding Natural Language

There is little doubt that the ability of computers to understand 'natural language' (that is, the ordinary language we use for human communication) is an ability upon which the intelligence or otherwise of computers can be, and will be, judged.

The inability of a computer to converse in our ordinary, everyday tongue at the very least sets up a barrier between the computer and ourselves. And such a barrier impedes our willingness to grant the computer a degree of intelligence.

There have been a couple of landmark programs in this field, and in this section of the book we will look at programs which will allow you to experience at least some of the excitement created by the original programs. The landmark-programs were SHRDLU (our version is called BLOCKWORLD) and ELIZA.

In the original SHRDLU, a 'robot' manipulated coloured blocks and other shapes, in response to natural language orders. It was able to carry out a superb conversation as to what is was currently doing, and why, and what it did in the past.

ELIZA, an imitation psychiatrist (after the style of Carl Rogers) was so effective and startling when it was first written that its creator reports receiving anguished telephone calls from people desperate for a little more access to the program to sort themselves out.

As well as BLOCKWORLD and ELIZA, we'll look at the problems and potential of machine translation. A fairly trivial program (TRANSLATE) is included in this section which generates sentences in 'Franglais' to illustrate the kind of solutions less-than-intelligent computers can reach when trying to handle not one, but two, natural languages.

HANSHAN, the final program in this section on language handling, creates random poems. This is a fairly low-level program compared to the others in this book, and one which — you may argue — hardly gives evidence of the brainpower of the computer which is running it. However, if you had read the preceding line some 30 years ago, with an author making an offhand remark about a low-cost machine being able to write poetry, followed by him or her dismissing this achievement as being fairly insignificant, you would have been amazed. Thirty years ago it may have

been an earth-shattering event. Proximity to wonder has blunted our perception and appreciation of it.

However, some of the results produced by the programs in this section should invoke at least an approximation to wonder. Before we get to the point of discussing and running the programs, we need to look a little at some of the problems which impede perfect communication between man and machine in natural language.

Language Parsing

Parsing is the word which describes the breaking up of sentences into elements which a computer can manipulate. The field of computational linguistics had traditionally researched ways of parsing sentences in order to reveal the role of various parts of the sentence in relation to their syntax. This is done, of course, in the hope that the machine doing the parsing can approximate an understanding of the sentence being processed.

However, there is now a growing interest in seeking meaning in terms of the sentence's role within a much wider frame of reference (such as we bring to bear, in terms of prior experience and knowledge of the environment, when attempting to understand a sentence). Of course, while research based on syntactic structure is continuing, the thrust towards 'world view environment' approaches is increasing.

It is pretty obvious why this is so. We want to be able to talk to computers on our own terms, rather than those dictated by metallic language limits. When we talk about a field which interests us, to friends with a similar interest, we can assume a great deal of commonly-shared background knowledge. In a similar way, we would like to be able to talk to computers when we can assume the existence of a particular knowledge base within which to communicate.

Assume you run a mining company. You have a computer program which will assist you in searching out precious minerals (at least one such program, PROSPECTOR, does exist). You would like to be able to talk to it in the words and phrases which are generally used by you when 'talking mining' with your colleagues.

It comes down to an effort to give a computer a 'world view' which will enable it to interpret natural language input, using the knowledge it has as a kind of template against which possible meanings can be checked.

You'll discover, in this section of the book, that the only convincing demonstrations of 'natural language communication' we can give are for extremely restricted 'world views'. In BLOCKWORLD, for example, the world consists of a two-dimensional space, within which your computer manipulates four coloured blocks. However, the computer's performance within that limited universe is fairly startling, even if it does not reach the dizzy heights of SHRDLU, the program which inspired it.

SHRDLU, for example could reply to sentences such as FIND A BLOCK THAT IS TALLER THAN THE ONE YOU ARE HOLDING AND PUT IT INTO THE BOX. You'll find BLOCKWORLD, even though it inhabits an even more restrictive universe than that of SHRDLU, unable to match it. However, as you'll see in due course, BLOCKWORLD can do pretty well on your computer.

The program has, as I said, four coloured blocks to manipulate. It can tell you where they are, by finding a specified block or by describing the whole scene, and can move them around. In the sample run which precedes the program listing, there is a green block on top of the yellow one. I asked the Amstrad to put the red block on top of the yellow one. This meant it first had to clear the top of the green one — in order to expose the yellow one — before locating the red block and putting it on top of the yellow.

Here's the program output (with the computer's speech in upper case):

```
. . . . . .
. . . . . .
. . . . . .
. . G . . .
. R Y B . .

Put the red block on the yellow one

          I UNDERSTAND
   NOW I'LL MOVE THE GREEN ONE

     I'M MOVING IT TO ROW 4
   I'M NOW MOVING THE RED ONE
      ONTO THE YELLOW BLOCK

. . . . . .
. . . . . .
. . . . . .
. . R G . .
. . Y B . .
```

As you'll discover when you run this program, there is powerful magic in communicating in English (a very limited subset, admittedly, but English nevertheless) with a computer, and having it both follow your instructions, and talk back to you in plain English as well.

In the early days of Al, much time was spent asking whether or not a program **really understood** what was going on. It was felt that even programs such as SHRDLU or ELIZA, while they gave convincing impressions of intelligent behaviour, didn't realy get us any closer to 'real' intelligence (whatever we assume that actually is).

This concern has lost much of its potency today. We do not spend time asking if a robot spot-welder working on a car assembly line can 'really see' what it is doing, or 'takes satisfaction' in a job well done. It is important that the thing works. If, as we will find to some extent in this section on language handling, the computer can handle language effectively, *as though* it 'really' understood what it was hearing and saying, this is more than enough in many situations.

The 'expert systems' programs (discussed in detail in the section of that name in this book) can make fresh discoveries, and can help human beings solve difficult problems. The pragmatic side of the Al world now tends to take an 'intelligence is as intelligence does' view of things. If it behaves intelligently — even within an extremely limited domain — let's assume the program does understand what is happening. Let's get on with the important questions, such as increasing the apparent intelligence of the beast.

Problems

There are a number of major problems which Al researchers are grappling with in an attempt to solve the mysteries of natural language processing. The enormous number of words in any human language, and the bewildering array of ways in which those words can be combined, is the major, and most obvious, stumbling block. Many phrases within a sentence are ambiguous. From prior knowledge, we can generally cut through the ambiguity to get at the meaning. Ambiguity is often inherent in speaking — perhaps more so than in written communication — and the spoken word is often incomplete and almost totally unstructured.

Each additional task a computer is given increases the processing time. A natural language system must not demand so much time that the process becomes useless in human terms. If it takes your computer a week to 'understand' a paragraph, you're not going to spend much time

investigating its ability to communicate with you.

Syntax and Semantics

These are the two approaches to the field of language parsing. They are not mutually exclusive. They are used to attack the problems which lie even within ordinary language use. Even working out which person 'he' refers to in the following sentence may take you a moment or two:

THE MAN WHO WAS WITH PETER SAID HE WAS TIRED

If this is read in a vacuum, as you have just done, there are no clues as to whom the 'he' refers.

Any natural language parsing system must be able to deal with problems like this. Margaret Boden (in *Artificial Intelligence and Natural Man*, Hassocks, Sussex: Harvester Press, 1977; p. 112) gives the delightful name of "The Archbishop's Problem" to the difficulty of automatically assigning such words. Her source for this name is *Alice in Wonderland:*

> **"Even Stigand, the partriotic Archbishop of Canterbury, found it advisable —"**
>
> **"Found what?" said the duck.**
>
> **"Found it," the mouse replied rather crossly. "Surely you know what 'it' means?"**
>
> **"I know what 'it' means well enough when I find a thing," said the duck. "It's generally a frog or a worm. The question is, what did the Archbishop find?"**

Let's have a look at the sentence now, and see how a parser might split it up, before putting each word through its processor in order to approximate an understanding of the writing analysed. (Then we'll examine the important question of how 'understanding' is defined).

Here is the sentence:

THE OLD THIN MAN IS UNDER THE OAK TREE

We can look at the sentence syntacticly (with each syntactic element of the structure bound within parentheses) as follows:

[[THE [[OLD][[THIN][MAN]]]]

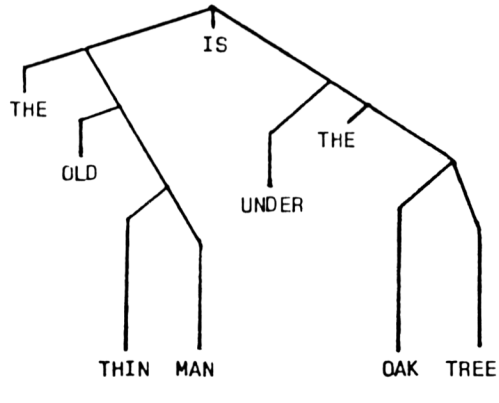
IS [[UNDER][THE[[OAK][TREE]]]]]

Look at this carefully, following the binding, and you may get a reasonable impression of the various elements which are thus bound together. For example, the words THIN MAN are individually bound, as [THIN][MAN] and also bound together in a larger group [[THIN][MAN]].

The adjective OLD modifies the noun, as well as THIN does, so it is bound in a similar way as [[OLD][THIN][MAN]]], except this binding sees a 'stronger' link between THIN MAN than between OLD and MAN. There is a further bond around the entire left hand side of the sentence [THE . . . MAN]]]] with the linking verb IS only bound by the parentheses which hold the entire sentence.

If we look at the right hand side, we can see that UNDER is held within the same bond as TREE, as a pair of parentheses bind the whole of this side. THE is not bound on both sides as are all the other words, in recognition of the fact that its only purpose is to modify the following noun (and 'the oak tree' is different, fairly obviously, from 'an oak tree').

We can express the syntactic structure of our sentence as a tree as follows:



If we could get a computer to break a sentence down like this, able to recognize the parts of speech on each branch of the tree, and/or within the

bonded pairs in our multi-parenthesised sentence, we would be well on the way to getting a degree of understanding.

This brings us back to the question I raised a short while ago. What do we mean, in the machine context, by "understanding"? J. Klir and M. Valach (in *Cybernetic Modelling,* London: Iliffe Books, 1965) suggest that understanding a spoken message is usually regarded to be a three-part thing:

1. **A way of 'hearing' the message.**

2. **A means of responding to that message.**

3. **A method for assessing whether or not the response (2) was such that it could be interpreted as showing understanding had taken place.**

There could be several ways of assessing the understanding of written text, claims Geoff Simons (in *Are Computers Alive?,* Brightong, Sussex: The Harvester Press, 1983; p. 129). These include supposing that understanding has taken place if the computer can answer questions correctly on it, or noting whether the machine can make intelligent connections between its own prior knowledge base, and the information it has picked up from its 'reading'.

Part Seven — Blockworld

Sometimes a computer 'conversing' in natural English (or an approximation to it) can produce a most unsettling effect. In BLOCKWORLD, a simplified version of a famous program called SHRDLU (which I'll discuss a little later), your computer manipulates a series of coloured toy blocks, following your instructions, and telling you — from time to time — how the blocks are arranged in relation to each other.

The blocks, of course, do not really exist, except as electronic figments of your computer's brain. However, you can see a representation of them on the screen, and this representation changes as the computer moves the blocks around.

As you've certainly gathered by now, it is generally easier to obtain a convincing demonstration of machine intelligence when the computer is operating within a limited domain. The domain of toy blocks is often used in Al experiments because it is clearly limited, yet allows a considerable degree of interaction and manipulation, as you shall see.

There are four blocks in the universe, your computer will be manipulating with this program. The blocks are red (shown as the letter 'R'), green ('G'), yellow ('Y') and blue ('B').

When the program begins, you see this on your screen:

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
.RYBG.
```

You are looking at the blocks from the 'front'. The BLOCKWORLD is essentially two-dimensional. Although you can move blocks around, and put them on top of each other, you cannot put blocks 'behind' or 'in front of' other blocks. The dots are invisible to the computer. They exist only for your benefit, and show a position which a block can occupy.

Although you have only a limited number of sentences you can use when communicating with the BLOCKWORLD, you'll be pleased to see how convincing that conversation can be. As well, it makes an easy-to-explain

(and impressive) demonstration of artificial intelligence in action to show your sceptical friends.

After the blocks have appeared on the screen, you'll see the '?' prompt, indicating that BLOCKWORLD is waiting to hear from you. We can ask the computer to describe its world:

```
? TELL ME WHAT YOU CAN SEE
  STARTING FROM THE RIGHT

     ...A SPACE THEN
MY SENSORS REGISTER THE GREEN BLOCK
IT'S THE BLUE BLOCK
MY SENSORS REGISTER THE YELLOW BLOCK
MY SENSORS REGISTER THE RED BLOCK
FINALLY   ...A SPACE
```

Once it has given you an overview in this way (reporting a dot as a 'space'), you can ask it to locate specific blocks within the world:

```
      .....
      .....
      .....
      .....
      .RYBG.

? WHERE IS THE YELLOW BLOCK

        > LET ME SEE NOW <

IT IS IN POSITION 3 FROM THE LEFT
THE RED BLOCK IS TO
  ITS IMMEDIATE LEFT

I CAN SEE THE BLUE BLOCK
  TO ITS RIGHT, TOUCHING IT


THERE IS NOTHING ABOVE THAT
```

Of course, straight reporting is not too big a deal, even though — as you can see — the program is responding to natural language questions, and

replying in a reasonable version of English. We now get to the meat of the matter, getting the Amstrad to manipulate the elements of its restricted domain:

```
......
......
......
......
.RYBG.

? PUT THE GREEN BLOCK ON THE YELLOW ONE

                OK

   I'M NOW MOVING THE GREEN ONE
     ONTO THE YELLOW BLOCK


                ......
                ..G...
                .RYB..

? WHERE IS THE YELLOW BLOCK

        > LET ME SEE NOW <

IT IS IN POSITION 3 FROM THE LEFT
THE RED BLOCK IS TO
  ITS IMMEDIATE LEFT
I CAN SEE THE BLUE BLOCK
  TO ITS RIGHT, TOUCHING IT


ABOVE IT IS THE GREEN BLOCK

THERE IS NOTHING ABOVE THAT
```

It is one thing to put one block (the 'object') on top of another block (the 'target') when the target is clear on top. However, it is another situation entirely, and one which requires a significant quantity of code, when the target must be cleared. The situation is made even more complicated when there is one block (or more) on top of the object block, which must be cleared before it can be moved. Here the program must clear the target. The object is unobstructed:

```
......
......
......
..G...
.RYB..

? PUT THE RED BLOCK ON THE YELLOW ONE

        I UNDERSTAND
NOW I'LL MOVE THE GREEN ONE

I'M MOVING IT TO ROW 4
I'M NOW MOVING THE RED ONE
  ONTO THE YELLOW BLOCK
```

A little later in the run, the blocks looked like this:

```
......
......
.R....
.B....
.GY...


? TELL ME WHAT YOU SEE


  STARTING FROM THE RIGHT
  ...A SPACE THEN
  ...A SPACE THEN
  ...A SPACE THEN
MY SENSORS REGISTER THE YELLOW BLOCK
  IT'S THE RED BLOCK
  ...AND BELOW IT...
  I SEE THE BLUE BLOCK
  ...AND BELOW IT...
  I SEE THE GREEN BLOCK
  FINALLY ...A SPACE
```

We now make the strongest demand to date:

```
      .....
      .....
      .R....
      .B....
      .GY...

PUT THE BLUE BLOCK ON THE YELLOW

          OK
I MUST MOVE THE RED BLOCK
I'M MOVING IT TO ROW 5
I'M NOW MOVING THE BLUE ONE
  ONTO THE YELLOW BLOCK
```

This program behaves in a manner which is almost uncanny. It appears to understand your commands, and acts in accordance with them. It seems to comprehend what it is doing, and can report the result of its actions.

How Does It Work?

How is it doing this? What is the secret of this impressive performance? The entire project boils down to sentence parsing to get the important parts of your commands, manipulation of elements within a two-dimensional array in response to instructions gleaned from the parsed input and, finally, printing out the array in a way which conveys the array contents so that they can be easily perceived.

'Explaining' it in this way does not, of course, really detract from the impression of intelligence this program conveys, although BLOCKWORLD is a primitive creature compared to its progenitor, SHRDLU.

This program, which was finished in 1971 by Terry Winograd, not only manipulated blocks, but showed them on a screen in colour, creating the world as well as interacting with it. Let's listen in as Terry speaks with SHRDLU (the computer output is in upper case letters, preceded by a > sign):

```
Pick up the big red block.
 >OK
Find a block which is taller than the one
   you  are holding,  and put it into the
   box.
 >OK
```

```
What does the box contain?
 >THE BLUE PYRAMID AND THE BLUE BOX
What is the pyramid supported by?
 >THE BOX
Is   there  a  large  block  behind   the
   pyramid?
 >YES, THREE OF THEM:  A LARGE RED ONE, A
   LARGE GREEN CUBE, AND THE BLUE ONE
```

As you can see, this is a remarkable effort and one which has led researchers to regard Winograd's work as one of the high spots in the development of artificial language programs. Our BLOCKWORLD pales somewhat by comparison.

However, BLOCKWORLD is still capable of fairly involved manipulation, even if its output is not as complex as that produced by SHRDLU. We will prove this claim by returning to our program, and asking it to jump through a few difficult hoops.

```
......
......
......
..B...
.GY.R.
```

The yellow block is, as you can see, underneath the blue one. We will now ask the program to reverse the position of those two blocks.

```
? PUT THE YELLOW BLOCK ON THE BLUE

      I UNDERSTAND
I MUST MOVE THE BLUE BLOCK
I'M MOVING IT TO ROW 4
I'M NOW MOVING THE YELLOW ONE
  ONTO THE BLUE BLOCK
          ......
          ......
          ......
          ...Y..
          .G.BR.
```

Now we will build a little tower of blocks to see how BLOCKWORLD can handle it:

```
? PUT THE RED BLOCK ON THE YELLOW ONE

          I UNDERSTAND

     I'M NOW MOVING THE RED ONE
       ONTO THE YELLOW BLOCK

                .....
                .....
                ...R..
                ...Y..
                .G.B..
```

The Amstrad is then told we want to get the bottom block from the tower of three and place it on the block which is currently standing alone. Note here that the program has been written to ensure that if the target is clear, it is not covered while the object is being uncovered:

```
? PUT THE BLUE BLOCK ON THE GREEN ONE
          OK
     I MUST MOVE THE RED BLOCK
     I'M MOVING IT TO ROW 6
     I MUST MOVE THE YELLOW BLOCK
     I'M MOVING IT TO ROW 6
     I'M NOW MOVING THE BLUE ONE
       ONTO THE GREEN BLOCK

                .....
                .....
                .....
                .B...Y
                .G...R
```

There are now two little towers, one with the blue block on top of the green one, and the other with the yellow block on top of the red one. The computer will be asked to manipulate both the blocks which are currently under other blocks. As you can see from the sample run at the top of page 112 we are putting BLOCKWORLD through a series of tests of increasing difficulty.

```
? PUT THE RED BLOCK ON THE GREEN ONE

          OK
I MUST MOVE THE YELLOW BLOCK
I'M MOVING IT TO ROW 5
NOW I'LL MOVE THE BLUE ONE

I'M MOVING IT TO ROW 5
I'M NOW MOVING THE RED ONE
  ONTO THE GREEN BLOCK

                .....
                .....
                .....
                .R..B.
                .G..Y.
```

It completed that without any problems. Let's use the green as our object (which is currently covered by the red) and the blue as the target to build up — in due course — a tower of all four blocks:

```
? PUT THE GREEN BLOCK ON THE BLUE ONE

     I UNDERSTAND
I MUST MOVE THE RED BLOCK
I'M MOVING IT TO ROW 3
I'M NOW MOVING THE GREEN ONE
  ONTO THE BLUE BLOCK


                .....
                .....
                ....G.
                ....B.
                ..R.Y.

? PUT THE RED BLOCK ON THE BLUE ONE

     I UNDERSTAND
NOW I'LL MOVE THE GREEN ONE

I'M MOVING IT TO ROW 2
I'M NOW MOVING THE RED ONE
  ONTO THE BLUE BLOCK
```

```
      .....
      .....
      ....R.
      ....B.
      .G..Y.

? PUT THE GREEN BLOCK ON THE RED ONE

          I UNDERSTAND

    I'M NOW MOVING THE GREEN ONE
      ONTO THE RED BLOCK

      .....
      ....G.
      ....R.
      ....B.
      ....Y.
```

Now we come to a very difficult test and one which — frankly — I did not think the program was going to be able to handle. Certainly, I had written a general-purpose target/object manipulation routine into the program, but was not confident it could handle a situation like the one it is about to tackle. It has to take a block (the blue one) from near the bottom of the tower by first moving the blocks from above it, then put this blue block on top of one of those (the red one) which it has taken from the initial stack:

```
? PUT THE BLUE BLOCK ON THE RED ONE

                OK

    I MUST MOVE THE GREEN BLOCK
    I'M MOVING IT TO ROW 6
    I MUST MOVE THE RED BLOCK
    I'M MOVING IT TO ROW 4
    I'M NOW MOVING THE BLUE ONE
      ONTO THE RED BLOCK

      .....
      .....
      .....
      ...B..
      ...RYG
```

```
? TELL ME WHAT YOU SEE

STARTING FROM THE RIGHT
IT'S THE GREEN BLOCK
MY SENSORS REGISTER THE YELLOW BLOCK
IT'S THE BLUE BLOCK
...AND BELOW IT...
I SEE THE RED BLOCK
...A SPACE THEN
...A SPACE THEN
FINALLY ...A SPACE
```

I was extremely pleased when BLOCKWORLD passed this test. I had not thought of testing in this difficult way. But when I tried it with a stack of four, and asked it to put the bottom block on top of the one which was currently on top of the stack of four, it bombed badly. It took me more than a week to work out how to solve it. At first I attempted a 'kludge' approach, with a separate routine for stacks of four. Then I realised that adding just seven characters to line 1790 (`TASK = 3 :`) enabled it to handle the situation.

BLOCKWORLD has been working hard, so I thought I should give it a little R & R:

```
......
......
......
...B..
...RYG

? SHUFFLE THE BLOCKS AROUND A BIT


IT'S GOOD TO BE GIVEN A CHANCE
   TO DO WHAT I WANT

......
......
......
......
GY..BR
```

We'll now look at the important elements of the program.

Modules of the Program

As with many other programs in this book, BLOCKWORLD starts off with a call to a subroutine at the end of the program which initialises the variables used.

Firstly, (line 2500) a five by six array is used to hold the 'world'. It is initially filled (lines 2510 through to 2550) with 46, the ASCII code of the dot which is used to indicate a blank space in the world. The starting position of the blocks is given by lines 2560 through to 2590. You can see here that the program assigns the initial letter of the colour ('R' for red, and so on) to the block of that colour. There is nothing very complicated in this first subroutine.

Although the initialisation subroutine is called just once per program, another subroutine, COLOUR NAME, is called every time the computer wishes to refer to a block. This subroutine changes the initial letter into the full name of the relevant colour. Both these subroutines are at the very end of the listing.

Back at the start of the program from line 30, we find a short section of code which prints out the view of the blocks. This could well have been a subroutine, but as it is needed everytime the program cycles through the main loop, it seemed sensible to have it here.

Line 50 shows that the view is printed 'upside down' with the '5 row' printed before the '4 row' and so on, with the '1 row' at the bottom of the scene. This was done to make it easier for the program to manipulate the blocks. It knows that it needs to look at a higher number to see if there is a block on top of the one it is considering. There would have been no real difficulty in doing it the other way (the lower the number, the higher the position of the block) but this seemed an unnecessary complication.

The next section of code, from line 130, accepts the user's input, and from it determines which subroutine should be called to act upon the input.

Constructing an AI program leads one very quickly to appreciate the complexities of intelligence in operation. BLOCKWORLD operates in a very restricted domain, and reacts only to those situations which have been specifically allowed for (although, as we saw in the sample run, it managed to grapple with a situation for which I did not realise I had

prepared it). Despite the limitations of domain and performance, BLOCKWORLD demands a lot of code, with a section to carry out each investigation, and to follow each command.

Look, for example, at the routine which determines the location of a specific block. First the program must check that you are asking for a block which is within its known universe. It does this by extracting — with line 260 — the initial letter of the block you are seeking, and checks — line 270 — that this is one of the four it recognizes. If you have asked it, for example, about a 'pink block', it uses the next routine (280 to 310) to randomly choose a reply, before returning to the main program.

If it bypasses that, it starts searching for the block. We do not use FOR/NEXT loops for this search, as we want the program to be able to exit the search at this point. A program which arbitrarily exits FOR/NEXT loops is not one which has been well constructed (many computers hang up fairly quickly if too many NEXT addresses accumulate on the stack). This part of the program gives it the first part of its required information regarding the block's location. The REM statements in the rest of the section explain what each one does.

The next subroutine, from line 1040, is called if you ask the program to TELL ME WHAT YOU SEE. This is a much simpler routine than the one which locates a specific block. The SHUFFLE BLOCKS routine is also relatively simple.

Finally, we come to the routine, starting at 1500, which produces the most impressive results, the 'PUT THE object on THE target' routine. As in the first major routine we examined in BLOCKWORLD, the REM statements explain what each section does.

Naturally enough, the program has to cater for each situation it is required to manage. After the complete program listing, we have a little more of Winograd's conversation with SHRDLU, to give you some ideas on how you can expand BLOCKWORLD. By keeping the program structured in a way similar to the present one, you'll find you can add complexity without getting lost in a maze of coding.

The only additional information you need is the input format demanded by the program. There are four questions you can ask, as follows (and this program expects them in upper case, although you can modify that to suit yourself):

WHERE IS THE colour BLOCK (or ONE or CUBE or whatever you like)?

TELL ME WHAT YOU SEE (or CAN SEE).
SHUFFLE THE BLOCKS.

PUT THE colour BLOCK ON THE colour ONE

You can quite the program at any time (as indicated by line 150) by simply pressing ENTER, when you are prompted for a question/command.

Here, now, is the listing of BLOCKWORLD for the Amstrad:

```
10 REM BLOCKWORLD
20 GOSUB 2470:REM INITIALISE
30 REM -- PRINT OUT VIEW --
40 PEN 1:CLS:PRINT:PRINT
50 FOR X=5 TO 1 STEP -1
60 PRINT TAB(8);
70 FOR Y=1 TO 6
80 PRINT CHR$(A(X,Y));
90 NEXT Y
100 PRINT
110 NEXT X
120 PRINT:PRINT
130 PEN 2:INPUT A$:A$=UPPER$(A$):PEN 3
140 PRINT
150 IF A$="" THEN END:REM TERMINATE RUN
   BY JUST PRESSING <ENTER>
160 IF LEFT$(A$,8)="WHERE IS" THEN GOSUB
 240
170 IF LEFT$(A$,12)="TELL ME WHAT" THEN
GOSUB 1050
180 IF LEFT$(A$,7)="SHUFFLE" THEN GOSUB
1280
190 IF LEFT$(A$,7)="PUT THE" THEN GOSUB
1500
195 WHILE INKEY$<>"":WEND
200 PRINT:PRINT:PEN 1:PRINT " >>>";:PEN
3:PRINT" PRESS ";:PEN 2:PRINT"<ENTER>";:
PEN 3:PRINT" TO CONTINUE ";:PEN 1:PRINT
"<<<"
205 SOUND 5,RND*100+100:WHILE INKEY$="":
WEND
210 GOTO 40
```

```
220 END
230 REM ---------------
240 REM "WHERE IS THE"
250 P=0
260 B$=MID$(A$,14,1)
270 IF B$="R" OR B$="Y" OR B$="B" OR B$=
"G" THEN 330
280 IF RND>0.7 THEN 300
290 PRINT "SORRY, I HAVE NO INFORMATION
ON THAT":GOTO 310
300 PRINT "I HAVE NO DATA WITH WHICH TO
ANSWER YOU";
310 RETURN
320 REM ----------------
330 M=ASC(B$)
340 PRINT TAB(8);"> LET ME SEE NOW <"
350 X=5
360 Y=1
370 IF A(X,Y)=M THEN 410
380 IF Y<6 THEN Y=Y+1:GOTO 370
390 IF X>1 THEN X=X-1:GOTO 360
400 GOTO 280
410 IF X>1 THEN 910:REM ON TOP OF
                          ANOTHER
420 IF Y>1 THEN 530:REM NOT ON LEFT
430 REM -------------------
440 REM -- ON LEFT --
450 PRINT "IT IS ON THE LEFT"
460 IF A(1,2)=46 THEN PRINT "THERE IS NO
THING TO ITS IMMEDIATE RIGHT";:GOTO 790
470 Q=A(1,2)
480 PRINT
490 PRINT "BESIDE IT I CAN SEE THE"
500 GOSUB 2400
510 PRINT "BLOCK"
520 GOTO 790
530 IF Y<6 THEN 650
540 REM --------------------
550 REM -- ON RIGHT --
560 PRINT
570 PRINT "IT IS ON THE RIGHT HAND SIDE"
580 IF A(1,5)=46 THEN PRINT "THERE IS NO
THING TO THE IMMEDIATE LEFT":GOTO 790
```

```
590 PRINT "TO ITS LEFT I SEE THE ";
600 Q=A(1,5)
610 GOSUB 2400
620 PRINT "ONE"
630 GOTO 790
640 REM ------------------
650 REM -- MIDDLE --
660 PRINT
670 PRINT "IT IS IN POSITION"Y"FROM THE
LEFT"
680 IF A(X,Y-1)=46 THEN PRINT "THERE IS
NOTHING ON ITS IMMEDIATE LEFT":GOTO 730
690 Q=A(X,Y-1)
700 PRINT "THE ";
710 GOSUB 2400
720 PRINT "BLOCK IS TO":PRINT "  ITS IMM
EDIATE LEFT"
730 IF A(X,Y+1)=46 THEN PRINT "NOTHING T
OUCHES IT ON THE RIGHT":GOTO 790
740 Q=A(X,Y+1)
750 PRINT:PRINT "I CAN SEE THE ";
760 GOSUB 2400
770 PRINT "BLOCK":PRINT"  TO ITS RIGHT,
TOUCHING IT"
780 REM ----------------------
790 REM -- ANYTHING ABOVE? --
800 PRINT
810 P=X
820 IF X=5 THEN 910
830 IF A(X+1,Y)=46 THEN PRINT:PRINT "THE
RE IS NOTHING ABOVE THAT":GOTO 310
840 PRINT:PRINT "ABOVE IT IS THE ";
850 Q=A(X+1,Y)
860 GOSUB 2400
870 PRINT "BLOCK"
880 X=X+1
890 GOTO 820
900 REM ------------------------
910 REM -- ON TOP OF ANOTHER? --
920 IF P<>0 THEN X=P
930 PRINT
940 IF X=1 THEN 310
950 PRINT "IT IS ";
```

```
960 PRINT "ON TOP OF THE ";
970 Q=A(X-1,Y)
980 GOSUB 2400
990 PRINT "BLOCK"
1000 X=X-1
1010 IF X<2 THEN 310
1020 GOTO 960
1030 RETURN
1040 REM ------------------
1050 REM "TELL ME WHAT"
1060 PRINT "STARTING FROM THE RIGHT"
1070 Y=6
1080 X=5
1090 IF A(X,Y)<>46 THEN 1150
1100 IF Y=1 AND X=1 THEN PRINT "FINALLY
";
1110 IF X=1 AND A(X,Y)=46 THEN PRINT ".
.A SPACE ";:IF Y>1 THEN PRINT "THEN"
1120 IF X>1 THEN X=X-1:GOTO 1090
1130 IF Y>1 THEN Y=Y-1:GOTO 1080
1140 RETURN
1150 L=INT(RND*3)
1160 IF L=0 THEN PRINT "IT'S THE ";:GOTO
 1190
1170 IF L=1 THEN PRINT "MY SENSORS REGIS
TER THE ";:GOTO 1190
1180 PRINT "I SEE THE ";
1190 Q=A(X,Y)
1200 GOSUB 2400
1210 PRINT "BLOCK"
1220 IF X=1 THEN 1130
1230 X=X-1
1240 PRINT "...AND BELOW IT ..."
1250 GOTO 1180
1260 RETURN
1270 REM --------------------
1280 REM SHUFFLE THE BLOCKS
1290 PRINT
1300 IF RND>0.5 THEN PRINT TAB(7);"IT'S
ABOUT TIME, TOO":GOTO 1320
1310 PRINT "IT'S GOOD TO BE GIVEN A CHAN
CE":PRINT TAB(4);"TO DO WHAT I WANT"
1320 FOR X=1 TO 5
```

```
1330 FOR Y=1 TO 6
1340 A(X,Y)=46
1350 NEXT Y
1360 NEXT X
1370 Y1=INT(RND*6)+1
1380 Y2=INT(RND*6)+1
1390 IF Y2=Y1 THEN 1380
1400 Y3=INT(RND*6)+1
1410 IF Y3=Y2 OR Y3=Y1 THEN 1400
1420 Y4=INT(RND*6)+1
1430 IF Y4=Y3 OR Y4=Y2 OR Y4=Y1 THEN 142
0
1440 A(1,Y1)=82
1450 A(1,Y2)=89
1460 A(1,Y3)=66
1470 A(1,Y4)=71
1480 RETURN
1490 REM -----------------
1500 REM "PUT THE..BLOCK..ON THE..ONE"
1510 IF RND>0.5 THEN PRINT TAB(5);"I UND
ERSTAND":GOTO 1530
1520 PRINT TAB(8);"OK"
1530 B$=MID$(A$,9,1):REM OBJECT BLOCK
1540 IF B$="R" THEN L=26
1550 IF B$="B" THEN L=27
1560 IF B$="G" THEN L=28
1570 IF B$="Y" THEN L=29
1580 C$=MID$(A$,L,1)
1590 B=ASC(B$)
1600 C=ASC(C$)
1610 FLAG=C
1620 REM -- FIND B$ BLOCK --
1630 X=5
1640 Y=1
1650 IF A(X,Y)=B THEN 1740
1660 IF Y<6 THEN Y=Y+1:GOTO 1650
1670 IF X>1 THEN X=X-1:GOTO 1640
1680 PRINT "I CAN'T FIND THE ";
1690 Q=B
1700 GOSUB 2400
1710 PRINT "ONE..."
1720 FOR TT=1 TO 2000:NEXT TT
1730 RETURN
1740 R=X:S=Y
```

```
1750 REM -- OBJECT BLOCK IS AT R,S --
1760 REM -- IS TARGET BLOCK CLEAR? --
1770 IF A(R+1,S)=46 THEN 1920:REM 'YES'
1780 IF A(R+2,S)=46 THEN TASK=1:GOTO 180
0
1790 TASK=3:IF A(R+3,S)=46 THEN TASK=2
1800 FOR W=TASK TO 1 STEP -1
1810 PRINT "I MUST MOVE THE ";
1820 Q=A(R+W,S)
1830 GOSUB 2400
1840 PRINT "BLOCK"
1850 DE=INT(RND*6)+1
1860 IF DE=S OR A(1,DE)=C OR A(2,DE)=C O
R A(3,DE)=C THEN 1850
1870 PRINT "I'M MOVING IT TO ROW"DE
1880 L=1
1890 IF A(L,DE)=46 THEN A(L,DE)=A(R+W,S)
:A(R+W,S)=46:GOTO 1910
1900 L=L+1:GOTO 1890
1910 NEXT W
1920 REM TARGET BLOCK AT R,S NOW CLEAR
1930 REM -- IS OBJECT BLOCK CLEAR? --
1940 REM --- FIND OBJECT BLOCK ---
1950 X=5
1960 Y=1
1970 IF A(X,Y)=C THEN 2070
1980 IF Y<6 THEN Y=Y+1:GOTO 1970
1990 IF X>1 THEN X=X-1:GOTO 1960
2000 PRINT "I CAN'T FIND THE ";
2010 Q=C
2020 GOSUB 2400
2030 PRINT "BLOCK"
2040 FOR TT=1 TO 2000:NEXT TT
2050 RETURN
2060 REM -- C HAS BEEN FOUND --
2070 T=X:U=Y:REM LOCATION OF C
2080 IF A(T+1,U)=46 THEN 2260
2090 IF A(T+2,U)=46 THEN TASK=1:GOTO 211
0
2100 TASK=3:IF A(T+3,U)=46 THEN TASK=2
2110 DE=INT(RND*6)+1
2120 IF DE=U OR DE=S THEN 2110
2130 FOR W=TASK TO 1 STEP -1
```

```
2140 PRINT "NOW I'LL MOVE THE ";
2150 Q=A(T+W,U)
2160 GOSUB 2400
2170 PRINT "ONE"
2180 PRINT
2190 PRINT "I'M MOVING IT TO ROW"DE
2200 L=1
2210 IF A(L,DE)=46 THEN A(L,DE)=A(T+W,U)
:A(T+W,U)=46:GOTO 2230
2220 L=L+1:GOTO 2210
2230 NEXT W
2240 REM -- OBJECT BLOCK NOW CLEAR --
2250 REM --- MAKE THE MOVE ---
2260 PRINT "I'M NOW MOVING THE ";
2270 Q=A(R,S):Z=A(R,S)
2280 GOSUB 2400
2290 PRINT "ONE"
2300 PRINT " ONTO THE ";
2310 IF A(T,U)=46 THEN A(T,U)=FLAG
2320 Q=A(T,U)
2330 GOSUB 2400
2340 PRINT "BLOCK"
2350 A(R,S)=46
2360 A(T+1,U)=Z
2370 FOR TT=1 TO 2000:NEXT TT
2380 RETURN
2390 REM ------------------
2400 REM COLOUR NAME
2410 IF Q=ASC("R") THEN PRINT "RED ";
2420 IF Q=ASC("Y") THEN PRINT "YELLOW ";

2430 IF Q=ASC("B") THEN PRINT "BLUE ";
2440 IF Q=ASC("G") THEN PRINT "GREEN ";
2450 RETURN
2460 REM ------------------
2470 REM INITIALISE
2480 MODE 1:INK 0,26:BORDER 26:INK 1,2:I
NK 2,6:INK 3,0:PEN 1
2490 RANDOMIZE TIME
2500 DIM A(5,6)
2510 FOR X=1 TO 5
2520 FOR Y=1 TO 6
2530 A(X,Y)=46
2540 NEXT Y
```

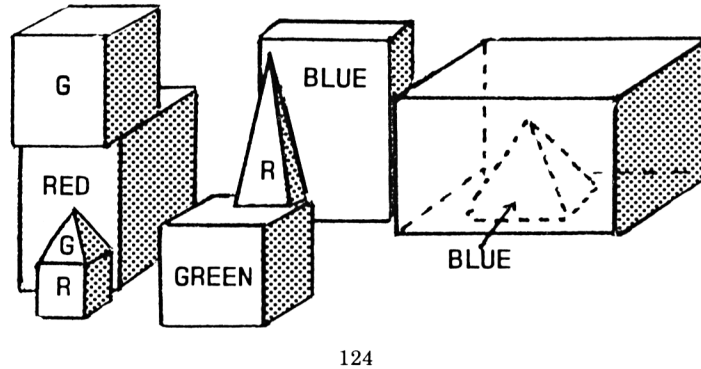
```
2550 NEXT X
2560 A(1,2)=ASC("R"):REM RED BLOCK
2570 A(1,3)=ASC("Y"):REM YELLOW
2580 A(1,4)=ASC("B"):REM BLUE
2590 A(1,5)=ASC("G"):REM GREEN
2600 RETURN
```

Improving the Program

Although BLOCKWORLD performs quite creditably in its present form, there is much that can be done to it to develop it. As you have seen, it has a simple modular structure, with a single subroutine to cope with each kind of command given to it. Therefore, to increase BLOCKWORLD's abilities, you can add to the simple vocabulary recognition section at the start of the program, and use this to send action to new subroutines.

You could start, perhaps, by giving BLOCKWORLD the ability to move a block to the right, or to the left, rather than just 'on top of' another block. Once you've managed to do this (and made sure it has a means of rejecting 'impossible' demands, such as trying to put a block to the right of one which is already at the very right-hand end of the frame), you could try adding a fifth block, or perhaps a shape — or more than one — which is not a block at all. Finally, you could create your own BLOCKWORLD program from scratch, in which the blocks and other elements can be moved in three dimensions, so 'in front of' and 'behind' are valid commands.

To give you some ideas of what can be done, we'll look at some more conversation with the landmark program, SHRDLU, written by Terry Winograd. The brief portion of conversation given earlier in this section was extracted from the longer conversation which follows. First of all, here's a sketch showing the situation under discussion:



Let the conversation continue:

```
Pick up the big red block.
  >OK
Grasp the pyramid.
  >I  DON'T UNDERSTAND WHICH PYRAMID  YOU
   MEAN
Find  the block which is taller than  the
   one  you are holding and put  it  into
   the box.
  >BY  "IT",  I ASSUME YOU MEAN THE BLOCK
   WHICH  IS  TALLER THAN THE ONE I HAVE
  >OK
What does the box contain?
  >THE BLUE PYRAMID AND THE BLUE BLOCK
What is the pyramid supported by?
  >THE BOX
How many blocks are not in the box?
  >FOUR OF THEM
Is at least one of them narrower than the
   one which I told you to pick up?
  >YES, THE RED CUBE
Is it supported?
  >YES, BY THE TABLE
Can the table pick up blocks?
  >NO
Can a pyramid be supported by a block?
  >YES
Can a pyramid support a pyramid?
  >I DON'T KNOW
Stack up two pyramids.
  >I CAN'T
The blue pyramid is mine.
  >I UNDERSTAND
```

Impressive as this is, it fades almost into insignificance when compared with the following exchange, which includes logic manipulation which even human beings would find difficult to follow. This next segment followed on directly from the one above:

```
I  own all the blocks which are not  red,
   but   I   don't  own  anything   which
   supports a pyramid.
  >I UNDERSTAND
```

```
Do I own the box?
  >NO
Do I own anything in the box?
  >YES,  TWO THINGS:  THE BLUE BLOCK  AND
   THE BLUE PYRAMID
Will  you please stack up both of the red
   blocks  and either a green cube  or  a
   pyramid?
  >OK
Which cube is sitting on the table?
  >THE LARGE GREEN ONE WHICH SUPPORTS THE
   RED PYRAMID
Is  there  a  large  block  behind  the
   pyramid?
  >YES, THREE OF THEM: A LARGE RED ONE, A
   LARGE GREEN CUBE AND THE BLUE ONE
Put  the  small one onto the  green  cube
   which supports a pyramid.
  >OK
Put the littlest pyramid on top of it.
  >OK
How  many  things  are on  top  of  green
   cubes?
  >I'M  NOT SURE WHAT YOU MEAN BY "ON TOP
   OF"  IN  THE PHRASE "ON TOP  OF  GREEN
   CUBES"
  >DO YOU MEAN:
   1 - DIRECTLY ON THE SURFACE
   2 - ANYWHERE ON TOP OF?
   2
Had  you touched any pyramid  before  you
   put the green one on the little cube?
  >YES, THE GREEN ONE
When did you pick it up?
   >WHEN I WAS STACKING UP THE RED CUBE, A
    LARGE RED BLOCK AND A LARGE GREEN CUBE
 Why?
   >TO GET RID  OF IT
 Why did you do that?
   >TO CLEAR OFF THE RED CUBE
 Why did you clear off that cube?
   >TO PUT IT ON A LARGE GREEN CUBE
```

```
Why did you do that?
  >TO STACK UP THE RED CUBE,  A LARGE RED
   BLOCK AND A LARGE GREEN CUBE
Why did you do that?
  >BECAUSE YOU ASKED ME TO
```

What could we add to that? Now that you've seen what SHRDLU and BLOCKWORLD can do, go and do likewise.

Part Eight — The Doctor is In

No study of 'natural language processing' would be complete without a discussion of ELIZA, the most famous program of this type (and, at one stage, the most famous computer program of all).

Joseph Weizenbaum, Professor of Computer Science at the Massachusetts Institute of Technology, created ELIZA in 1964-66. The program was designed to take on the role of a Rogerian psychotherapist (Weizenbaum insists it was intended to parody such a practitioner). A psychotherapist of Carl Rogers' school works by refusing to lead the conversation with a patient, preferring instead to reflect the patient's remarks back on themselves, so the conversation direction is constantly dependent upon the patient.

The reaction to the program — once it was made public — at first startled, and then seriously worried Weizenbaum. As a result of the train of thought engendered by the shock of its reception (which we'll discuss in a moment), Weizenbaum left MIT for two years to ponder the wider implications of the ELIZA experience, and during that time wrote the very important work *Computer Power and Human Reason* (San Francisco: W. H. Freeman and Co., 1976). There is quite a bit of material in this book that I don't agree with, but this in no way diminishes the importance of the work. Regardless of my views, it is a work I urge you to read if you are seriously interested in artificial intelligence, and in the relationships and dependencies which we see developing between man and machine.

You use Weizenbaum's program by typing in your side of a 'conversation', to which the program replies, using its inner stock of answers. Here's part of one 'conversation' I had with my Amstrad, running ELIZA:

```
? I HAVE A SERIOUS PROBLEM
CAN YOU ELABORATE ON THAT?

? I AM AFRAID OF MICE
HOW LONG HAVE YOU BEEN AFRAID OF MICE

? SINCE BEFORE I TOOK UP COMPUTERS
WHY DO YOU MENTION COMPUTERS?

? IT SEEMED RELEVANT AT THE TIME
CAN YOU ELABORATE ON THAT?
```

? IT SEEMED RELEVANT TO THE FACT THAT I AM AFRAID OF LITTLE SQUEAKY THINGS
HOW LONG HAVE YOU BEEN AFRAID OF LITTLE SQUEAKY THINGS

? I THINK WE HAVE BEEN THROUGH THIS BEFORE
MARTIN, DO YOU DOUBT WE HAVE BEEN THROUGH THIS BEFORE

? I SOMETIMES DOUBT EVERYTHING
WHAT DOES THAT SUGGEST TO YOU?

? THAT I AM LIKE MOST PEOPLE IN HAVING DOUBTS
HOW LONG HAVE YOU BEEN LIKE MOST PEOPLE IN HAVING DOUBTS

? SINCE I STARTED DREAMING OF ROBOTS
WHY DID YOU BRING UP THE SUBJECT OF DREAMS?

? I HAVE THIS THING ABOUT MICE
COULD YOU SAY THAT AGAIN, IN A DIFFERENT WAY?

? I'M AFRAID THEY'LL COME AND JUST STAY FOR TEA
WHY ARE YOU TELLING ME YOU'RE AFRAID THEY'LL COME AND JUST STAY FOR TEA

? I THOUGHT YOU WERE MEANT TO ASSIST IN AN HOUR OF NEED
I'M NOT SURE I UNDERSTAND YOU FULLY.

? HOW CAN I SAY IT MORE CLEARLY
PERHAPS YOU DON'T WANT TO BE ABLE TO SAY IT MORE CLEARLY

? I THINK YOU ARE AN AMSTRAD FRAUD
WHAT MAKES YOU THINK I AM AN AMSTRAD FRAUD

? BECAUSE YOU NEVER GIVE ME A STRAIGHT ANSWER
WHAT DOES THAT SUGGEST TO YOU?

? COME ON TURKEY I WANT A STRAIGHT ANSWER
WHY DO YOU WANT A STRAIGHT ANSWER

? TO HELP ME IN MY HOUR OF NEED
COULD YOU SAY THAT AGAIN, IN A DIFFERENT WAY?

? YOU'RE PRETTY DUMB FOR A MACHINE
WHAT IS YOUR REACTION TO ME BEING PRETTY DUMB FOR A MACHINE

? IT MAKES ME WANT TO DRINK HEAVILY
MODERATION IN ALL THINGS SHOULD BE THE RULE.

```
? EVEN IN SEX
I FEEL YOU SHOULD REALLY DISCUSS THIS WITH A HUMAN.

? BYE
BYE BYE FOR NOW.
SEE YOU AGAIN SOME TIME.
```

Weizenbaum chose the name ELIZA for his program because he said, like the Eliza in *Pygmalion,* it could be taught to speak increasingly well. The original ELIZA program was written in two parts. The first part analysed the user input, and the second part was a 'script'. Different scripts were designed for different topics, and DOCTOR was one of the scripts. (Other scripts could well have held discussions on ancient ships, real estate, currency exchange rates or whatever).

The DOCTOR (Rogerian) script was the first one Weizenbaum tried out. The program became well known around MIT because it was a very effective way to demonstrate the power of a computer (remember, this was all a long, long time ago, in a galaxy far away, when people could not buy computers off the shelf at their local store).

Weizenbaum reported his work on ELIZA to the computer press in due course ("ELIZA — A Computer Program for the Study of Natural Language Communication Between Man and Machine", *Communications of the Association for Computing Machinery,* vol. 9, no. 1 [January 1965], pp. 36-54) and soon a number of versions of it — based on his description — were running at other institutions in the United States.

Weizenbaum reports that there were three distinct events which "shocked" him, as ELIZA's use became widespread. Firstly, he was horrified (and I find it hard to appreciate why he was as alarmed as he reports) to discover that people quickly became involved with the program.

Weizenbaum's secretary had worked closely with him over the six months or so it took to produce the program, and she knew as well as he how it worked, scanning a person's input for 'key words' (such as DREAM or FRIENDS) and then choosing a suitable reply from a bank of such replies.

Other words from the user's sentences could be incorporated, sentences could be 'turned around' (so "I am happy because of the weather" could be simply fed back as either "Why are you happy because of the weather" or just as a statement of the form "You are happy because of the weather. What does this suggest to you?"). A number of other replies (such as

"That is interesting, please go on") could be used if no key word was recognized.

Despite the secretary's familiarity with the program, Weizenbaum noticed that if he walked into the office when she was accessing the program, she became embarrassed, and refused to let him see the printout. Further, when he suggested it would be interesting to hook up a printer to the main body of the computer to record the late night conversations students were having with the program, the suggestion was greeted with horror. It was as though he had suggested a kind of electronic peeping-Tom activity.

Weizenbaum was bothered by how strongly people identified with the program, given it a personality, and sharing their most intimate thoughts with it. He said he had not realised the "powerful delusional thinking" a fairly simple program could create in normal people.

The Russian Connection

Pamela McCorduck, in her splendid book *Machines Who Think* (San Franciso: W. H. Freeman and Co., 1979) confirms the effect the program can have. She reports that the first time she saw ELIZA up and talking was at the Stanford Computation Center where an internationally respected computer scientist from the Soviet Union was being shown around.

He sat down at a computer connected to a version of the program written by one of Weizenbaum's colleagues, Kenneth Colby (who we'll be meeting again, shortly) and started typing. McCurduck reports watching in embarrassment as — triggered by a phrase such as TELL ME ABOUT YOUR FAMILY — the scientist proceeded to discuss some personal worries in some depth, becoming oblivious to those around him.

Weizenbaum found that some accesses to the program, via time-sharing terminals scattered around the university, often went on for an hour or more, late into the night. He received telephone calls from people who desperately wanted access to the program for a short time, in order to sort out their problems.

Colby, who we mentioned a short time ago, had met Weizenbaum some time earlier, at Stanford. Colby — Professor of Psychiatry at UCLA — was interested in artificial intelligence. He thought its findings might possibly lead to new views on human thinking (and Colby hoped to gain new insights into neurotic behaviour through his studies). Before

Weizenbaum's original paper on ELIZA appeared, a short note on it was published by Colby in the *Journal of Nervous and Mental Diseases.*

The two men split shortly after this, primarily because Weizenbaum strongly disagreed with Colby's claims that the program could have genuine therapeutic applications, but also because it seemed that Colby did not properly credit Weizenbaum for the original work on ELIZA.

Colby and two colleagues suggested that an improved version of DOCTOR would have genuine therapeutic use. Colby thought it could be made available to mental hospitals which were short of staff, so patients could access the program (via time-sharing systems) on demand. Weizenbaum was horrified. He says he thought it was vital that there be, as a starting point from which one person could assist another in coping with problems, an emphatic, 'fellow-human' recognition of those problems.

Short, Sharp Shocks

Weizenbaum was shocked that even a single practicing psychiatrist could advance the view that the healing process could be replaced purely by mechanical technique. Such a thought had never crossed his mind. Furthermore, even if it could be done, it *should not* be done. There are some areas where machines should never be allowed to stray, claimed Weizenbaum, even if they have the ability to do so.

Colby was not chastened by Weizenbaum's response. He was, it seems, perfectly happy to consider the possibility of pure technique proving efficacious. Further he defended his view, saying that only laymen confused psychotherapy with marriage. A professional working relationship between therapist and patient was what mattered he said.

More to the point, Colby attacked Weizenbaum for the claim that there were areas in which the computer should never be employed. Why not, asked Colby. Just because Weizenbaum says so? Does Weizenbaum believe that helping people by computer is somehow worse than letting them suffer? And should not a therapist explore every possible tool which is available, just in case one of them proves to be genuinely effective?

Colby's view is more or less supported by Carl Sagan who is quite at peace with the idea of an ELIZA-like program being available — for a few dollars a session — in specially constructed areas, somewhat like telephone booths *(Broca's Brain,* London: Coronet Books, 1980, p. 300).

And this is where Weizenbaum's third 'shock' came in. Remember, he had been startled by the identification with and the unequivocal anthropomorphization of the program. Then he was very alarmed at the suggestions that somehow ELIZA could take the place of, or assist, human therapists. His third 'shock' came from his observation that many people came to believe that somehow the program was important in demonstrating that a real solution to the problems of a machine understanding human language was at hand. He dismissed this idea out of hand. Indeed, in the original paper on the program, Weizenbaum had been at pains to point out that it was impossible to find a general solution to this problem.

I said earlier that I did not agree with all in Weizenbaum's book *Computer Power and Human Reason.* One of the points upon which I disagree is the 'there are some things which should never be done by machines'. John McCarthy (1976, "An Unresonable Book", in *Three Reviews of J. Weizenbaum's Computer Power and Human Reason, Memo AIM-291,* Stanford Al Laboratory, November), advances the view that if there are functions which a computer should not be taught to carry out, these should not be done at all, by a person or a machine.

Others agree. In the book *Artificial Reality* (1983: Addison-Wesley Publishing Co., Reading, MA; p. 168), Myron W. Krueger suggests that even if Weizenbaum's horror at the thought of using his program — or a development of it — for genuine therapy was real, such fear was groundless and misplaced.

However, regardless of my views (or others') of Weizenbaum's thesis, and of the value of the book (I've already said I think you should read it, if only to give your own mental mill grist regarding the debate), there is no doubt that ELIZA has proved an extremely entertaining companion. You will soon prove this assertion for yourself.

As well as enjoying the program, you'll also be in a position to judge whether or not it actually suggests that intelligence resides in the machine which is running it. Adrian Berry (in *The Super-Intelligent Machine,* Lond: Jonathan Cape Ltd., 1983; p. 63) concludes that ELIZA (and PARRY, a program which is designed to mimic a paranoid patient) is a pretty poor advocate for the possibilities of true artificial intelligence.

You'll find that your own views will sway back and forth as you use the program. When ELIZA produces a particularly inspired or appropriate remark, you'll feel this is sure evidence of intelligence, on at least some level. At other times, you'll discover the ELIZA is great simply for

entertainment. Nobody, you claim, could manage to ask such absurd questions (DID YOU COME TO ME BECAUSE I WAS DEAD? one implementation asked Berry, he reports in the book mentioned above) or innocently create superb nonsequiturs as you will certainly find when you run this program on your Amstrad.

The Program

Of course, despite the impression that his program creates when running on your Amstrad, there is no real intelligence in it. It is, instead, basically a sleight of hand track. As we said earlier, the program scans your input for words it can use, and then reflects your words back to you in a way which makes it seem as if the program really is speaking to you. Once you've run it a few times, you'll learn how to trigger the most effective responses. As well, as I pointed out earlier, it makes a good demonstration to run for those who are not used to computers. (But be careful in case some of that 'powerful delusional thinking' comes into play, and you find one of your friends confiding his or her darkest thoughts to your machine). Now that you know how it works, it is time to hang up your computer's shingle, and go into practice:

```
10 REM ELIZA
20 GOSUB 730:'INITIALIZE
30 PRINT "WELCOME TO ANOTHER SESSION WIT
H"
40 PRINT "YOUR AMSTRAD PSYCHIATRIST, ELI
ZA"
50 PRINT:INPUT "WHAT IS YOUR FIRST NAME"
;Z$
60 IF Z$="" THEN 50
70 Z$=UPPER$(Z$)
80 CLS
90 PRINT "WELL, HI THERE ";Z$;"."
100 PRINT:PRINT "IT SURE IS NEAT TO HAVE
 YOU DROP BY"
110 PRINT:PRINT "PRESS MY SPACE BAR WHEN
 YOU'RE READY     TO START..."
120 N=N+1
130 IF INKEY$="" THEN 120
140 CLS:RANDOMIZE N
150 PRINT "OK ";Z$;", WHAT SEEMS TO BE T
HE PROBLEM?":PRINT:PRINT
160 GOSUB 650:'ACCEPT USER INPUT
```

```
170 IF RND<0.2 THEN PRINT Z$;", ";
180 GOSUB 210:'REM PROCESS INPUT
190 PRINT:GOTO 160
200 ' ******************
210 ' FIND KEY WORD, PHRASE
220 Z=0
230 Z=Z+1
240 D=INSTR(A$,B$(Z))
250 IF D<>0 THEN 290
260 IF Z<35 THEN 230
270 GOSUB 480:RETURN
280 ' ******************
290 REM PROCESS KEYWORDS
300 PRINT C$(Z);" ";
310 IF RIGHT$(C$(Z),1)<"A" THEN PRINT:RE
TURN
320 H=LEN(A$)-(D+LEN(B$(Z)))
330 IF H>0 THEN A$=RIGHT$(A$,H)
340 Z=0
350 Z=Z+1
360 D=INSTR(A$,F$(Z))
370 IF D<>0 THEN 400
380 IF Z<8 THEN 350
390 PRINT A$:RETURN
400 J$=LEFT$(A$,(D-1))+" "+G$(Z)
410 Z=LEN(A$)-LEN(LEFT$(A$,(D-1)))-LEN(G
$(Z))
420 PRINT J$
430 IF Z>2 THEN L$=RIGHT$(A$,(Z-2)):IF I
NSTR(K$,L$)=0 THEN RETURN
440 IF Z>2 THEN PRINT RIGHT$(A$,(Z-2))
450 IF Z<2 THEN PRINT
460 RETURN
470 ' ******************
480 REM RANDOM REPLIES, NO KEY FOUND
490 Z=INT(RND*11)+1
500 ON Z GOSUB 530,540,550,560,570,580,5
90,600,610,620,630
510 RETURN
520 ' ******************
530 PRINT "WHAT DOES THAT SUGGEST TO YOU
?":RETURN
540 PRINT "I SEE...":RETURN
```

```
550 PRINT "I'M NOT SURE I UNDERSTAND YOU
 FULLY.":RETURN
560 PRINT "CAN YOU ELABORATE ON THAT?":R
ETURN
570 PRINT "THAT IS QUITE INTERESTING!":R
ETURN
580 PRINT "THAT'S SO...PLEASE CONTINUE..
.":RETURN
590 PRINT "I UNDERSTAND...":RETURN
600 PRINT "WELL, WELL...DO GO ON.":RETUR
N
610 PRINT "WHY ARE YOU SAYING THAT?":RET
URN
620 PRINT "PLEASE EXPLAIN THE BACKGROUND
 TO THAT   REMARK.":RETURN
630 PRINT "COULD YOU SAY THAT AGAIN, IN
A          DIFFERENT WAY?":RETURN
640 ' *****************
650 ' ACCEPT USER INPUT
660 LINE INPUT A$:A$=UPPER$(A$):K$=A$
670 IF LEFT$(A$,3)="BYE" THEN PRINT:PRIN
T "BYE BYE FOR NOW, ";Z$;".":PRINT:PRINT
 "SEE YOU AGAIN SOME TIME.":PRINT:PRINT:
END
680 K=LEN(A$)
690 IF LEFT$(A$,7)="BECAUSE" THEN A$=RIG
HT$(A$,(K-7)):K=K-7
700 A$=" "+A$+" "
710 RETURN
720 ' ******************
730 ' INITIALIZE
740 MODE 1:N=0
750 DIM B$(35),C$(35),F$(8),G$(8)
760 SOUND 1,222,20:SOUND 1,654,30
770 ' FILL B$,C$ WITH KEYWORDS, REPLIES
780 FOR A=1 TO 35
790 READ B$(A),C$(A)
800 NEXT A
810 RESTORE 1250
820 FOR A=1 TO 8
830 READ F$(A),G$(A)
840 NEXT A
850 RETURN
```

```
860 ' ******************
870 ' KEYWORDS, REPLIES
880 ' NOTE SPACES WITHIN QUOTE MARKS
890 DATA " CAN YOU","PERHAPS YOU WOULD L
IKE TO BE ABLE TO"
900 DATA " DO YOU","YES, I"
910 DATA " CAN I","PERHAPS YOU DON'T WAN
T TO BE ABLE TO"
920 DATA " YOU ARE","WHAT MAKES YOU THIN
K I AM"
930 DATA " YOU'RE","WHAT IS YOUR REACTIO
N TO ME BEING"
940 DATA " I DON'T","WHAT DON'T YOU"
950 DATA " I FEEL","TELL ME MORE ABOUT F
EELING"
960 DATA "WHY DON'T YOU","WHY WOULD YOU
WANT ME TO"
970 DATA " WHY CAN'T I","WHAT MAKES YOU
THINK YOU SHOULD BE ABLE TO"
980 DATA " ARE YOU","WHY ARE YOU INTERES
TED IN WHETHER OR NOT I AM"
990 DATA " I CAN'T","HOW DO YOU KNOW YOU
 CAN'T"
1000 DATA " SEX","I FEEL YOU SHOULD REAL
LY DISCUSS THIS   WITH A HUMAN."
1010 DATA " I AM","HOW LONG HAVE YOU BEE
N"
1020 DATA " I'M","WHY ARE YOU TELLING ME
 YOU'RE"
1030 DATA " I WANT","WHY DO YOU WANT"
1040 DATA " WHAT","WHAT DO YOU THINK?"
1050 DATA " HOW","WHAT ANSWER WOULD PLEA
SE YOU MOST?"
1060 DATA " WHO","HOW OFTEN DO YOU THINK
 OF SUCH QUESTIONS?"
1070 DATA " WHERE","WHY DID YOU ASK ME T
HAT?"
1080 DATA " WHEN","WHAT WOULD YOUR BEST
FRIEND SAY TO THAT QUESTION?"
1090 DATA " WHY","WHAT IS IT THAT YOU RE
ALLY WANT TO KNOW?"
1100 DATA " PERHAPS","YOU'RE NOT VERY FI
RM ON THAT!"
```

```
1110 DATA " DRINK","MODERATION IN ALL TH
INGS SHOULD BE THE RULE."
1120 DATA " SORRY","WHY ARE YOU APOLOGIZ
ING?"
1130 DATA " DREAM","WHY DID YOU BRING UP
 THE SUBJECT OF DREAMS?"
1140 DATA " I LIKE","IS IT GOOD THAT YOU
 LIKE"
1150 DATA " MAYBE","AREN'T YOU BEING A B
IT TENTATIVE?"
1160 DATA " NO","WHY ARE YOU BEING NEGAT
IVE?"
1170 DATA " YOUR","WHY ARE YOU CONCERNED
 ABOUT MY"
1180 DATA " ALWAYS","CAN YOU THINK OF A
SPECIFIC EXAMPLE"
1190 DATA " THINK","DO YOU DOUBT"
1200 DATA " YES","YOU SEEM QUITE CERTAIN
. WHY IS THIS SO?"
1210 DATA " COMPUTER","WHY DO YOU MENTIO
N COMPUTERS"
1220 DATA " FRIEND"," WHY DO YOU BRING U
P THE SUBJECT OF FRIENDS?"
1230 DATA " AM I","YOU ARE"
1240 ' CONJUGATE
1250 DATA " I ","YOU"
1260 DATA " ARE ","AM"
1270 DATA " WERE ","WAS"
1280 DATA " YOU ","ME"
1290 DATA " YOUR ","MY"
1300 DATA " I'VE ","YOU'VE"
1310 DATA " I'M ","YOU'RE"
1320 DATA " ME ","YOU"
1330 DATA " AM I ","YOU ARE"
1340 DATA " AM ","ARE"
```

Part Nine — Machine Translation

It would seem — when thinking about some of the possibilities that arise from machines being able to understand and process natural language — that computers could be of great value in helping us translate from one human language to another. Such a hope has been with us since the early fifties, and a great deal of progress has been made in the field.

There are now more than 20 machine translation (MT) systems in use around the world. But, contrary to popular opinion, these systems do no work on a SHOVE IN THE DOCUMENT IN ENGLISH IN ONE SLOT and GET THE FRENCH VERSION OUT OF ANOTHER basis. MT is more subtle, and more involved. In fact, there are several subfields within the overall domain of MT.

Still a Use for People

Although, in the early days of building MT systems, it was accepted (probably without too much thought) that human translators would eventually prove redundant as machines became more skilled, researchers have now confirmed that at present (and for the immediate future) the role of human translators is vital. Specialists in the field now talk about 'machine pre-translation', with the documents produced by MT systems being seen as simply rough working drafts of the final, translated works.

There are several different approaches to MT which are in use at present. These include systems which have been built with the idea of translating documents written in a king of 'stripped-down', limited version of natural language, or documents which have been edited to make them easier for the machine to handle before they are fed to it. Xerox have a system of this type, called SYSTRAN. We'll be looking at some output produced by SYSTRAN (working on documents for the EEC) in due course.

Another approach is one where the user can modify the system to his or her own needs, giving it a vocabulary to suit the speciality in which the MT will take place. Such a system, called CULT, is currently in use in Hong Kong where it translates Chinese mathematical journals. The direct printout of the machine is bound and sold to libraries around the world.

When you and I, as laymen, have thought about MT, it is likely that we have envisaged machines which will perform in a STICK ENGLISH IN THE INPUT, GET FRENCH FROM THE OUTPUT mode, and this is

the eventual goal of those developing MT. It is far from being realised at present. However, the SYSTRAN system — mentioned a short time ago as working with documents written in 'sub-English', or ones which had been pre-edited — can be used in a 'freelance' mode, in which it will tackle any document which is fed into it. The success achieved has varied from document to document.

Many documents go through a pre-editing stage before being offered to a machine for translation. In this stage, an attempt is made to weed out potential ambiguities, and other aspects of the text which could trip up a machine. Many documents (most, in fact) need to have be post-edited. In this stage, a check is made for genuine errors by the machine, and syntax is cleaned up.

Some documents do not need to be post-edited. For certain purposes, the rough output direct from the MT system may be enough.

MT may also be carried out with the assistance of a human translator, intervening in the work while the translation is underway.

As you can see from the above, the role of the human is still vital in the translation process. And there is no indication that this will change in the near future. Machines can do the rough and ready pedestrian work of translation, but human polishing and correction is still needed.

Let's look at a genuine example of machine translation. This comes from an EEC document, translated from French to English by the SYSTRAN system in 1981.

Here is the start of the document in French:

> Application de la micrologique au controle des operations de production.
>
> But de la recherce:
>
> Perfectionner les appareillages existants de sorte que les preposes soient debarasses des taches dans lesquelles leur jugement n'intervient pas.
>
> Application au central de telesurveillance d'engins sur pneus.

The machine responded with this translation:

> Application of micrological to the control of the production operations.
>
> Aim of the research:
>
> To improve existing equipments so that the officials debarasses tasks in which their judgement does not intervene.
>
> Application to the exchange of telesurveillance of equipment on tyres.

Although this is pretty rough, a fair amount of the meaning comes through. The 'debarasses' which survives in the English translation is, in fact, due to a spelling error in the French original (it should have been 'debarrasses' which, presumably, the machine would have understood).

After the human post-editing, the document read as follows:

> Application of micrology to the monitoring of production operations.
>
> Aim of the research:
>
> To perfect existing apparatus so that staff can be relieved of tasks where no judgement is required.
>
> Application to the remote monitoring station for trackless vehicles.

I find it fascinating to follow through the way the document has evolved. Apart from the final line, the final version of the English text is not wildly different from SYSTRAN original output.

Not all of the document was as successfully translated. The human post-editor took a savage pen to one line further down the text, reducing the MT output to a shadow of its former self.

Here's what the machine printed out:

> It publishes station and day reports indicating the duration and the importance relative of the periods devoted by each instrument supervised to the various possible activities: evacuation of the products, transport of equipment, maintenance, station service . . . as well as the number of evacuated coal cups.

This is the kind of text which is a dead giveaway of MT, with such phrases as 'the importance relative of the periods' showing clearly their birth in French.

After post-editing, the text was reduced to the following:

> It publishes shift and day reports indicating the duration and the relative portion of time spent by each vehicle recorded on the various possible tasks: coal clearance, materials transport, maintenance, refuelling points . . . as well as the number of coal buckets carried.

Finally, before we get on to creating our own 'translation' program, it is interesting to note that the vast majority of documents using MT at present are non-literary. The translation of literary works is another field entirely, and in so far as MT is concerned, is barely in its infancy.

Franglais

This Amstrad program, using a vocabulary devised by Jeremy Ruston and based on an idea from him, accepts English input, and gives out a strange polyglot mixture of French and English, where the easiest and most obvious words are translated into French, and the difficult ones are left in English (this technique could produce, for example, JE SUIS UN TRES EXASPERATED HOMME for I AM A VERY EXASPERATED MAN). The magazine Punch has a regular feature called "Let's Talk Franglais" which shows how delightful such a curious mixture of languages can be.

The program given here is not designed to be a serious one. It does, however, indicate some of the problems inherent in MT. More seriously, with a greatly extended vocabulary, it could be used to produce a very rough document in a kind of French from English text (or from French to English, simply by swapping two variables) which could then be

extensively post-edited. If the program was used in a field with a specialist vocabulary, it could do quite a serviceable job, although it would not be able to make any judgements to ensure that the various parts of a sentence (such as gender demands in French) were correct.

You may think the claim that this program could be used seriously, with an extended vocabulary, is unrealistic when you read some of the output of the program. However, if you think about it, you'll see that its potential is by no means even approached in the current form.

The program goes through the text, looking for the space which indicates the start of a new word (the word, of course, starts after the space, which is why — in line 30 — we added a space to each end of the input, so the program would not ignore the first and final words). Once it finds one (line 140) it goes to the routine from 160 which continues to search for the next space, so it can isolate the whole word. Then it simply runs through the vocabulary until it finds a match.

If it does find such a match, the French word is printed in place of the English one, and the program returns to continue the search. Note that once a match has been found, the program immediately reverts to this point. It does not waste time searching through the rest of the vocabulary. This means that words near the top of the list will be translated more quickly than those at the end. This is why the commonly-used words (such as THE, ME and AM) are at the top of the list.

```
? HELLO MY GOOD FRIENDS
 --> BONJOUR MON BON AMIS

? I AM VERY PLEASED TO SEE YOU
  HERE
 --> JE SUIS TRES PLEASED A VU
     VOUS ICI

? COULD I HAVE SOME STEAK FOR MY
  EVENING MEAL
 -->COULD JE AI DES ENTRECOTE
    POUR MON EVENING MEAL

? EVERYBODY THINKS THE TRENDY
 POLICEMAN IS A SUPER DETECTIVE
 --> TOUT LE MONDE THINKS LE
     AVANT-GARDE GENDARME EST
     UNE FANTASTIQUE CLUESO
```

```
? I AM FEELING RIGHT INSIDE MY
  HEAD WHEN I MAKE MUSIC BEHIND
  THE LITTLE CAT
--> JE SUI.S FEELING DROITE
    INSIDE MON TETE QUAND JE
    MAKE MUSIQUE DERRIERE LE
    PETITE CHAT
```

Time now for you to experience a little MT of your own with TRANSLATE:

```
10 REM TRANSLATE
20 GOSUB 280:REM INITIALISE
30 PEN 1:SOUND 5,100*RND+100:INPUT B$:B$
=" "+UPPER$(B$)+" ":L=LEN(B$):REM   ACCE
PT USER INPUT
40 IF B$="  " THEN END:REM TWO SPACES
50 GOSUB 90:REM TRANSLATE
60 GOTO 30
70 END
80 REM -------------
90 REM TRANSLATE
100 PEN 3:PRINT TAB(2);"-->";:PEN 2
110 K=0
120 K=K+1
130 IF K=L THEN PRINT:PRINT:RETURN
140 IF MID$(B$,K,1)=" " THEN 160
150 GOTO 120
160 X=K+1
170 Y=0
180 Y=Y+1
190 IF MID$(B$,(X+Y),1)=" " THEN Q$=MID$
(B$,X,Y):GOTO 210
200 GOTO 180
210 M=0
220 M=M+1
230 IF Q$=E$(M) THEN PRINT F$(M);" ";:GO
TO 260
240 IF M<COUNT THEN 220
250 PRINT Q$;" ";
260 GOTO 120
270 REM -----------------
280 REM INITIALISE
```

```
290 MODE 1:INK 0,26:BORDER 26:INK 1,2:IN
K 2,6:INK 3,0:PEN 1
300 DEFINT A-Z
310 DIM E$(100):REM TO HOLD ENGLISH
320 DIM F$(100):REM TO HOLD FRENCH
330 COUNT=0
340 COUNT=COUNT+1
350 READ E$(COUNT),F$(COUNT)
360 IF F$(COUNT)<>"*" THEN 340
370 RETURN
380 REM -- DATA --
390 DATA THE,LE,ME,MOI,I,JE,HERE,ICI,AM,
SUIS,ARE,EST,NOT,NE,IN,DANS
400 DATA WHEN,QUAND,YOU,VOUS,IS,EST,IT,I
L,DAY,JOUR,AND,ET,SOME,DES,OF,DE
410 DATA HAVE,AI,A,UNE,MY, MON,YOUR,VOTR
E,OF,DE,TO,A,SEE,VU,VERY,TRES
420 DATA ROOM,CHAMBRE,STEAK,ENTRECOTE,FR
IES,POMME FRITES,BIG,GRAND,FOR,POUR
430 DATA MATCH,ALLUMETTE,SUPER,FANTASTIQ
UE,DEAD,MORT,WITH,AVEC
440 DATA GIN,VIN,WHISKEY,VIN,WHISKY,VIN,
BEER,VIN,MARTINI,VIN,WINE,VIN
450 DATA PARIS,PLAINS,PLAINS,PARIS,HAIR,
CHEVAUX,CIGARETTES,GAULOISES
460 DATA ARM,BRA,LEG,JAMBE,RIGHT,DROITE,
LEFT,GAUCHE
470 DATA TRENDY,AVANT-GUARDE,MEDICINE,VI
N,POLICEMAN,GENDARME
480 DATA DETECTIVE,CLUESO,DOOR,PORTE,HEA
D,TETE,LOVE,ARMOUR
490 DATA HOUSE,MAISON,CHAIR,CHAISE,EYE,O
RIEL,SUN,SOLIEL
500 DATA SONG,CHANSON,FRIENDS,AMIS,BEHID
,DERRIERE,SEA,MER,MOTHER,MERE
510 DATA CAT,CHAT,DOG,CHIEN,BLUE,BLEU,LI
TTLE,PETITE
520 DATA MUSIC,MUSIQUE,PLEASE,S'IL VOUS
PLAIT,BOY,GARCON,GIRL,FILLE
530 DATA FISH,POISSON,CHICKEN,POULET,DUC
K,CANARD,MUSTARD,MOUTARDE
540 DATA HOT,CHAUD,COLD,FROID,EVERYBODY,
TOUT LE MONDE
550 DATA HELLO,BONJOUR,GOOD,BON
560 DATA *,*
```

Part Ten — Hanshan

Our final program in this section on language handling creates random poems. This is a pretty trivial program, and one which — you may argue — hardly gives evidence of the presence of artificial intelligence.

However, imagine you were reading a book like this 30 years ago. The author makes a casual remark about a low cost device writing poetry automatically, and then dismisses this as a minor matter. Thirty years ago it would have been extraordinary. And, really, when you think about it, it still is. We have become so accustomed to the miraculous we tend to be blind to it.

So, with that thought in mind, we turn to HANSHAN to create a few poems. The program is named after the Chinese poet HAN-SHAN, who lived in the 8th and 9th centuries. After falling out with his farming family he wandered for many years, then settled as a recluse on the Cold Mountain (Han-Shan) after which he is now known.

All the phrases used in this program's DATA store come from the book *Chinese Poems* (Arthur Waley, Unwin Paperbacks, London, 1982). The program selects from one of three patterns, within which it creates poems which are Haiku-like (the Haiku is, of course, a Japanese form, but the program does not mind any conflict between Chinese phrases and the form into which they are placed by the program).

Some of the poems produced by HANSHAN have a surprising degree of merit:

```
SULLEN, SULLEN
  I TAKE YOUR POEMS...
     MEN OF LEARNING


WHEN SHALL WE MEET
  FOR THESE FEW STEPS...
     IN THE BLACK DARKNESS
```

```
HANGING...TWISTING
     ....WRITHED
        THOSE THAT ARE LEFT


WHY SHOULD YOU WASTE
  WAVES OF COOLNESS...
     NOW AT DUSK


MEN OF LEARNING
  MEN OF ACTION...
     AND MUCH GRIEVING


I HURRY FORWARD
  I TAKE YOUR POEMS...
     SULLEN, SULLEN
```

Here is the HANSHAN listing to enable you to create a nearly infinite sequence of poems. By all means modify the DATA statements to make the program (and its output) your own:

```
10 REM HANSHAN
20 GOSUB 250:REM INITIALISE
30 REM CHOOSE PATTERN
40 WHILE NOT FINISHED:R=INT(RND*3)+1:PEN
 R
50 ON R GOSUB 90,140,190
60 FOR TT=1 TO 1000:NEXT TT
70 PRINT:PRINT:PRINT
80 WEND
90 REM -- PATTERN ONE --
100 GOSUB 640:PRINT "...";:GOSUB 640:PRI
NT
110 PRINT TAB(5);"....";:GOSUB 640:PRINT
120 PRINT TAB(8);:GOSUB 660
130 RETURN
```

```
140 REM -- PATTERN TWO --
150 GOSUB 660:PRINT
160 PRINT TAB(3);:GOSUB 660:PRINT "..."
170 PRINT TAB(6);:GOSUB 660
180 RETURN
190 REM -- PATTERN THREE --
200 PRINT TAB(3);:GOSUB 640:PRINT
210 GOSUB 660:PRINT
220 PRINT TAB(3);:GOSUB 640:PRINT ", ";:
GOSUB 660:PRINT
230 RETURN
240 REM ----------------
250 REM INITIALISATION
260 MODE 1:INK 0,0:INK 1,25:INK 2,18:INK
 3,8:BORDER 0
270 RANDOMIZE TIME
280 DIM W$(20),S$(20)
290 FOR J=1 TO 20
300 READ W$(J)
310 NEXT J
320 FOR J=1 TO 20
330 READ S$(J)
340 NEXT J
350 RETURN
360 REM -- DATA --
370 REM -- SINGLE WORDS --
380 DATA SCURRYING,TREADING,GAZING,WITHE
RED,CHISELLED
390 DATA MUFFLED,FLANKED,WRITHED,BENDING
,TWISTING
400 DATA HAMMERED,HANGING,WINDING,CLEARE
ST,WEARY
410 DATA EARTHWARD,CATARACT,SACRIFICIAL,
SLIPPERY,ASUNDER
420 REM -- SHORT PHRASES
430 DATA IN THE COOL STREAM
440 DATA NODDED IN CLUSTERED GRACE
450 DATA WAVES OF COOLNESS
460 DATA OUT FROM THE DEEPEST
470 DATA "SULLEN, SULLEN"
480 DATA IN THE BLACK DARKNESS
490 DATA I TAKE YOUR POEMS
500 DATA I PUT OUT THE LAMP
510 DATA MY SHORT SPAN RUNS OUT
```

```
520 DATA THOSE THAT ARE LEFT
530 DATA MEN OF LEARNING
540 DATA MEN OF ACTION
550 DATA I HURRY FORWARD
560 DATA WHY SHOULD YOU WASTE
570 DATA WHEN SHALL WE MEET
580 DATA LITTLE SLEEPING
590 DATA AND MUCH GRIEVING
600 DATA FOR THESE FEW STEPS
610 DATA NOW AT DUSK
620 DATA I HAVE DONE WITH PROFIT
630 REM -- SELECT/PRINTOUT --
640 PRINT W$(INT(RND*20+1));
650 RETURN
660 PRINT S$(INT(RND*20+1));
670 RETURN
```

```
  GAZING
OUT FROM THE DEEPEST
  HAMMERED, NOW AT DUSK
                              WITHERED...WITHERED
                                  ....CHISELLED
                                     IN THE BLACK DARKNESS

IN THE BLACK DARKNESS
  OUT FROM THE DEEPEST...
     MEN OF ACTION
                              BENDING...SCURRYING
                                  ....WRITHED
                                     I PUT OUT THE LAMP

WINDING...BENDING
    ....EARTHWARD
       FOR THESE FEW STEPS
                              CATARACT...ASUNDER
                                  ....CATARACT
                                     IN THE BLACK DARKNESS

I TAKE YOUR POEMS
  LITTLE SLEEPING...
     I HAVE DONE WITH PROFIT
                              SCURRYING...WRITHED
                                  ....SLIPPERY
                                     WHY SHOULD YOU WASTE
```

Part Eleven — Expert Systems

There is a limited number of experts in the world on any one subject. It doesn't matter what field you're talking about — mending cars, mining for uranium, diagnosing human illness, sorting edible mushrooms from poisonous ones — there is a limit to the number of experts we have available.

Now while the world is not exactly crying out for more mushroom-sorting experts, there are areas of the world (most of it in fact) where there are not enough doctors. One idea of an expert system is to 'capture' the expertise of one of our experts on a computer, in such a way that a non-expert can tap the information.

Expert systems is the one area of Al research where significant strides have been made. It is the area where such systems are already making genuine, economically viable contributions. And is the one area of Al which is not at all bothered by questions of whether or not the machine displaying the expertise is "thinking".

In its simplest form, an expert system is a series of IF/THEN statements. A diagnostic system could be as simple as this:

IF the patient is coughing
AND he has recently been soaked to the skin
AND then stood in a freezing wind for an hour
THEN the patient has a cold or pneumonia.

of course, one would hardly need an expert system to make a diagnosis like this (and note that I am not suggesting the diagnosis of my IF/THEN chain is necessarily correct). An expert system comes into its own when either of the following conditions exist:

— the expert is not present but his or her expertise is;

— even the 'expert' doesn't know with 100% certainty the casual links between the observations and the results. This could happen if a medical researcher was aware that patients contracting disease X have tended to have had contact with foods A and B and have blood group C . . . although no way of linking A, B and C — apart from the fact that they appear together — had been discovered. In this case, a properly programmed expert system could make predictions about the likelihood

of individual D contracting the disease, even when the percentage contribution that factors A, B and C made were unknown. By studying enough cases, the expert system could not only devise its own rules for predicting whether a particular individual would, or would not, contract the disease, but could then explain its reasoning to a human physician.

The 'mathematics of reasoning' are very important in the construction of expert systems. Often a person 'drawing out' the expertise of a human being in order for it to be encoded into an expert system database (and we'll look a little later at some of the systems which are at work around the world at present) discovers the expert does not know how he or she actually reaches decisions.

It can be as much of a revelation to the expert as to the person creating the knowledge base for the computer program. In *The Fifth Generation — Artificial Intelligence and Japan's Computer Challenge to the World* (Reading, Massachusetts: Feigenbaum, Edward A. and McCorduck, Pamela, 1983; pp. 85, 86) we read the very sad story of an expert who willingly explained explained his methods to a 'knowledge engineer' (the name given to those who draw out others' expertise and then modify it for the computer program). The expert was highly regarded (and well paid) for his expertise, and was at first disbelieving when the knowledge engineer discovered the expertise could be reduced to a few hundred 'working rules of thumb'. From disbelief, the expert's view changed to one of depression, and finally he quit his field, a broken man.

Machines make decisions based on their internal rules. These are — as we saw in the discussion leading up to the learning and reasoning programs — relatively simple. Elementary logical reasoning comes down to a relatively few, easily expressed, rules.

We saw that syllogisms could be expressed, and solved, by machine, because they took the following form:

A is a C
C is a B
Therefore, A is a B

The hope of reducing reasoning to a mechanical process has been with us a long time. Back in 1677, in the preface to the work *The General Sciences,* Gottfried Leibniz wrote:

If one could find characteristics or signs appropriate for expressing all our

thoughts as clearly and exactly as arithmetic expresses numbers or analytic geometry expresses lines, we could in all subjects, in so far as they are amendable to reasoning, accomplish what is done in arithmetic and geometry . . .

Moreover we should be able to convince the world of what we have found or concluded since it would be easy to verify the calculation . . . if someone doubted the results I should say to him: 'Let us calculate Sire,' and taking pen and ink we should soon settle the question.

Rather than taking pen and ink, we can now take silicon, and find answers to at least some questions which are beyond most of us to discover (such as the ability to predict the chemical structure of a not-yet-developed compound, as one expert system can do) and indicate the solutions to problems which nobody alive can solve.

Limitations

Unless they are specifically programmed to alert an operator to it, expert systems can be pretty stupid when they come across something which does not fit within their preprogrammed repertoire. It is like someone who is brilliant at chess, but unable to master the steps needed to knot a tie. An *idiot savant* status is characteristic of many low-level expert systems which are based solely on interpreting rules of the IF/THEN type, such as I discussed earlier.

Such systems have no ability to extend their knowledge base while operating, and can only think in a straight line from point A to B, then of B to C and so on. Such a system may have no way of knowing when its laboriously programmed knowledge was inappropriate, no way of recognizing the exception to the rule.

The system we will develop comes within the *idiot savant* description. But depsite this limitation, which applies to the majority of expert systems in use in the world today, you'll find the systems you develop are fascinating artefacts. Our final system, as you'll see, does have the ability to learn. In fact, you simply tell it — as it tries to distinguish between any number of things you have programmed into it — whether its guess was right or wrong, and eventually it will have taught itself to distinguish between the objects, without you explicitly telling it how to make the distinction between them.

Chemical Structure and Dendral

Before we get to our expert systems, we will have a look at some of the systems in use at present, and see what we can learn from examining them.

The first program we will look at, and possibly the world's first real, working expert system, is called DENDRAL. Work on this system — which is able to work out facts about molecular structures from raw chemical data — began at Stanford University in 1965. Bringing together expertise from a number of disciplines (with those which provided DENDRAL with its working knowledge base of physical chemistry), DENDRAL's creators eventually produced a system which now performs better than anyone in the world in its field (including the men who built it). DENDRAL is in use around the world.

Stanford was also the breeding ground for MYCIN, a system which diagnoses blood and meningitis infections, then gives treatment suggestions. MYCIN bases it conclusions on physical data entered by a physician, and can — if requested — explain how it came to reach the diagnosis it did. The system contains some 450 rules.

The knowledge base in MYCIN is so valuable that a companion program — GUIDON — has been developed to enable the computer to act as a teacher, thus acting as a bridge from one human expert (or, a set of them in this case) to another, newly-minted human expert.

That is still not the end of the MYCIN's value. Much of the program consists of ways of *manipulating* the rules it has been given and *drawing conclusions* from them. The mechanisms of manipulation and inference are — to a large extent — independent from the knowledge base. This suggests that the information relating to blood infections could be removed, and new information be added. This has been done, and the expert system PUFF now dispenses similar assistance to that given by MYCIN, but in relation to lung disorders.

So effective was this process (and in one trial of 150 patients, PUFF produced the same diagnosis as did human specialists) that another version of MYCIN, simply called EMYCIN (for Empty MYCIN) has been developed, into which other knowledge bases can be entered.

The expert system MOLGEN (for MOLecular GENetics) assists biologists working on DNA and with genetic engineering. It is widely used.

The most interesting thing — in terms of examining the directions AI

researching is taking — is that expert systems actually work extremely well, and it makes sense economically to use them. This ensures that they are being used, and that more are being developed. The 'pure research' line, does naturally enough produce results, but the results tend to come along more quickly when there are immediate practical needs for that which is being developed, and big bucks and available for the developers.

Think of a system which gave advice on where to drill for oil. A single find, and the cost of developing the system, even if that ran into the millions, could be earned back relatively quickly, perhaps even in a matter of days.

Feigenbaum and McCorduck (in *The Fifth Generation*, mentioned earlier, pp. 72, 73) give a graphic example of the 'earning-back' power of major expert systems. They cite the case of a major American company which has recently bought an expert system designed to diagnose failures in particular types of electricity generating plants. Testing an early, and largely incomplete, version of the program against the real data that led to one of the company's plants being shut down in 1981, it was found the system discovered the cause of the problem that led to the shutdown in a matter of seconds. It had taken the human experts working at that plant days to come to the same conclusion. In the meantime, the plant had been shut down for four days, a closure that cost the company around $1.2 million.

There are many other systems in use or under development around the world. These include:

— PROGRAMMER'S APPRENTICE: A system for helping, as its name suggests, with the writing of software.

— EURISKO: An expert system which is able to learn as it works, which creates three-dimensional microelectric circuits.

— GENESIS: An exciting-sounding one. This system, which is on the market now, allows scientists to plan and simulate gene-splicing experiments.

I'm afraid we won't be getting into gene-splicing just yet although we will be finding some interesting applications for our expert systems (such as differentiating between a man, a horse and a sparrow!). Let's have a look at the first of our systems now.

Part Twelve — The Little Spurt

Our first expert system is SPURT. This program has the ability to tell, without error, the difference between three living creatures — a man, a horse and a sparrow. Although this is a pretty silly situation, and one which probably does not arise very often in your experience, it can teach us a great deal about how some kinds of expert systems are developed.

Imagine a 'medical diagnosis' expert system. We'll call our imaginary system MEDICI. MEDICI and SPURT are close cousins, as you'll see, and studying SPURT will give you a base upon which you can build up a useful degree of knowledge of MEDICI and other, more wide-ranging, expert systems.

You are about to have a session with MEDICI. The system asks you a large number of questions which you answer with a YES or a NO, as follows:

```
                ARE YOU MALE?
      ARE YOU MORE THAN 40 YEARS OLD?
                DO YOU SMOKE?
HAVE YOU HAD A CHECKUP IN THE LAST 12 MONTHS?
          DO YOU WORRY FREQUENTLY?
WOULD YOU DESCRIBE YOURSELF AS A TENSE PERSON?
```

And so on. After a string of these questions, MEDICI pasuses for a nanosecond or two, then prints the following message on the screen:

```
THANK YOU. YOUR LIFE EXPECTANCY IS 79 YEARS, THUS
EXCEEDING 11% OF THE POPULATION. TO INCREASE YOUR
CHANCES OF REACHING, OR EXCEEDING THIS, I SUGGEST
        YOU - TRY TO STOP SMOKING
            - GET REGULAR MEDICAL CHECKUPS
            - INCREASE YOUR EXERCISE EACH WEEK

          THANK YOU FOR CONSULTING MEDICI
```

What did MEDICI do? How did it turn your YES/NO answers into a life expectancy prediction? Actually, as I'm sure you've already decided, this is not a very sophisticated program, and would not demand a very high level of expertise. However, it shows how a real medical diagnosis program might begin, if the expert system was interacting directly with a patient,

rather than with a physician as is generally the case at present.

Pleased that you're going to live longer than 11% of the population, you settle down to make the acquaintance of another expert, young SPURT. Here's what you see on your screen:

```
I WANT YOU TO THINK OF A MAN, A HORSE
    OR A SPARROW

    DOES IT HAVE TWO LEGS
          Y OR N? Y

    CAN IT WALK
          Y OR N? Y

    CAN IT FLY
          Y OR N? N

  YOU WERE THINKING OF A MAN

  ----------------------------
```

Of course, SPURT is right. It was not very hard to determine from your answers that you were thinking of a man. Very impressed, you press the ENTER key, and have another run:

```
I WANT YOU TO THINK OF A MAN, A HORSE
    OR A SPARROW

    DOES IT HAVE TWO LEGS
          Y OR N? Y

    CAN IT WALK
          Y OR N? Y

    CAN IT FLY
          Y OR N? Y

    YOU WERE THINKING OF A SPARROW
```

Surely it couldn't do it again, you think, and try for the third alternative:

```
I WANT YOU TO THINK OF A MAN, A HORSE
    OR A SPARROW

    DOES IT HAVE TWO LEGS
          Y OR N? N

    CAN IT WALK
          Y OR N? Y

    CAN IT FLY
          Y OR N? N

    YOU WERE THINKING OF A HORSE
```

This time you decide to quit. How does SPURT record the answers to your questions so it can determine that if you said the creature you were thinking of had two legs and could walk, but could not fly, was a man? How, for that mater, could MEDICI tally your answers and tell you that you'd live till you were 79?

It is very simple, at least in the case of SPURT (and MEDICI worked the same general way, only with a considerable degree of refinement). SPURT counted each time you gave the answer 'Y' to a question. If you gave only one 'Y' answer, you must have been thinking of a horse (as the WALK question was the only one to which you could reply 'Y' if you were thinking of a horse). Two 'Y' answers, and it was a man you had in mind. Three, and SPURT knew it was the sparrow you were thinking of.

It is a pretty simple program, but one which lays a foundation upon which expert systems could be built. Here's the listing:

```
10 REM SPURT
20 MODE 1:INK 0,26:BORDER 26:INK 1,2:INK
 2,6:INK 3,0
30 PEN 1:PRINT "I WANT YOU TO THINK OF A
 MAN, A HORSE"
40 PRINT TAB(6);"OR A SPARROW"
50 FOR TT=1 TO 2000:NEXT TT
60 PRINT:PRINT
70 PEN 2:GOSUB 170:REM ASK QUESTIONS
```

```
80 PRINT
90 PEN 3:PRINT "------------------------
----"
100 PRINT:PRINT "PRESS <ENTER> FOR ANOTH
ER ONE, OR"
110 PRINT "ANY KEY AND THEN <ENTER> TO Q
UIT"
120 INPUT Q$
130 IF Q$<>"" THEN END
140 CLS
150 GOTO 30
160 REM ------------------
170 REM ASK QUESTIONS
180 COUNT=0
190 PRINT "DOES IT HAVE TWO LEGS"
200 GOSUB 310
210 PRINT "CAN IT WALK"
220 GOSUB 310
230 PRINT "CAN IT FLY"
240 GOSUB 310
250 PRINT "YOU WERE THINKING OF A ";
260 IF COUNT=1 THEN PRINT "HORSE"
270 IF COUNT=2 THEN PRINT "MAN"
280 IF COUNT=3 THEN PRINT "SPARROW"
290 RETURN
300 REM --------------------
310 REM PROCESS ANSWER
320 INPUT "           Y OR N";Z$
330 IF Z$<>"Y" AND Z$<>"N" THEN 320
340 IF Z$="Y" THEN COUNT=COUNT+1
350 PRINT
360 RETURN
```

The Little X-Spurt

X-SPURT is SPURT's big brother. Although this new program bears a definite family relationship to the one we first loooked at, it is considerably more sophisticated.

You can see this increased sophisticated by looking at a sample run from it. Firstly, we will get it to perform much as SPURT did. However, you can tell from the opening frame that this is a rather different program. It is largely 'soft', that is the expertise is not hardwired as in the case of

SPURT but can be entered differently for each run.

```
NAME OF SYSTEM? CREATURES

NUMBER OF OUTCOMES? 3

NUMBER OF FACTORS TO BE CONSIDERED? 3
```

You tell the program its subject matter (CREATURES in this case), and then the number of OUTCOMES (that is, results) and the number of FACTORS TO BE CONSIDERED. These are the variables (such as CAN IT FLY) which must be considered. Having given it the framework, X-SPURT now asks you to fill in the outlines:

```
CREATURES

WHAT IS OUTCOME 1 ? MAN

WHAT IS OUTCOME 2 ? HORSE

WHAT IS OUTCOME 3 ? SPARROW
```

Having told it the outcomes, it asks you to enter the questions which relate to the factors which determine which outcome you are seeking:

```
PLEASE ENTER QUESTION 1
? DOES IT FLY UNAIDED


PLEASE ENTER QUESTION 2
? DOES IT HAVE TWO LEGS


PLEASE ENTER QUESTION 3
? DOES IT WALK
```

This may seem like a lot of trouble we're going to, just to emulate SPURT, but — as you'll see shortly — it will be worthwhile. This simple exercise is showing you how X-SPURT can be trained to become an expert on just about anything.

X-SPURT now goes through each of the outcomes you have entered, and says "If I asked the following question, in respect of this outcome, would you answer 'yes' or 'no'. From this information, X-SPURT can assemble an equivalent knowledge base to the one which was hardwired into SPURT. Of course, X-SPURT could be building up a knowledge base on anything.

```
PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  MAN  <


ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY


   > DOES IT FLY UNAIDED? N


   > DOES IT HAVE TWO LEGS? Y


   > DOES IT WALK? Y


PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  HORSE  <


ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY


   > DOES IT FLY UNAIDED? N


   > DOES IT HAVE TWO LEGS? N


   > DOES IT WALK? Y
```

```
PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  SPARROW  <

ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY

   > DOES IT FLY UNAIDED? Y

   > DOES IT HAVE TWO LEGS? Y

   > DOES IT WALK? Y
```

Once you've been through each of the possible outcomes, and told it what your answers would be for the questions, X-SPURT creates a 'knowledge base', which in this case is little more than adding up the total 'Y' replies. X-SPURT reports its findings to you:

```
THIS IS MY EXPERT BASE:

MAN --- 6

HORSE --- 4

SPARROW --- 7
```

But where did it get those numbers? You could not have given four 'Y' answers for horse, or 7 for sparrow, because there are only three questions. X-SPURT does not add a single one for each 'Y' answer, but instead gives a number which changes for each answer. If there was just one awarded for each 'Y', and you answered 'Y' to, say, questions one and three for one thing, and to questions two and three for another thing, it would have the same total for both objects.

To get round this, to ensure that the actual *order* in which the 'Y' answers are given is important, we proceed as follows:

A 'Y' answer to question 1 is worth 1
A 'Y' answer to question 2 is worth 2
A 'Y' answer to question 3 is worth 4
A 'Y' answer to question 4 is worth 8
..... 5 16
..... 6 32
..... 7 64
... and so on ...

This makes sure that, even if the same number of 'Y' answers are given for two different things, a different identifying number will be given to our expert by which to make judgements.

Does it work? Of course it does, and here is X-SPURT showing itself in action:

```
PLEASE ENTER 'Y' OR 'N'...


DOES IT FLY UNAIDED
? N


DOES IT HAVE TWO LEGS
? Y


DOES IT WALK
? Y
        > MY RESULT WAS 6


        > YOU WERE THINKING
          OF MAN
```

I said before that X-SPURT was capable of doing a great deal more than SPURT, and now I will show the truth of that claim. We are going to train our expert system in another field, one in which I have no expertise whatsoever. The knowledge base fed into X-SPURT came from a book,

written by an expert called Oliver Chambers *(The Observer's Book of Rocks and Minerals,* New York: Frederick Warne, 1979). With the help of Mr. Chambers' expertise, X-SPURT is about to acquire the skills to identify five different types of minerals, using four factors.

This is something I could not do without X-SPURT's help. This is true for most people using expert systems today. An expert system encodes, as it were, an absent expert's expertise, so non-experts can make use of that knowledge base at will.

Let's pass some of Mr. Chambers' knowledge onto our system (starting with a new run, so that minerals do not become confused with horses and sparrows):

We tell it the subject matter:

```
NAME OF SYSTEM? MINERAL IDENTIFICATION

NUMBER OF OUTCOMES? 5

NUMBER OF FACTORS TO BE CONSIDERED? 4
```

Then we give it the five minerals it will be trying to identify:

```
WHAT IS OUTCOME 1 ? PLEONASTE

WHAT IS OUTCOME 2 ? LIMONITE

WHAT IS OUTCOME 3 ? IODYRITE

WHAT IS OUTCOME 4 ? IRIDOSMINE

WHAT IS OUTCOME 5 ? SYLVANITE
```

Next, X-SPURT learns some questions which will assist it in discriminating between the minerals:

```
PLEASE ENTER QUESTION 1
? IS IT HARD
```

```
PLEASE ENTER QUESTION 2
? DOES IT CONTAIN STREAKS OF A DIFFERENT
  COLOR FROM THE MAIN COLOR


PLEASE ENTER QUESTION 3
? IS ITS SPECIFIC GRAVITY ABOVE 5


PLEASE ENTER QUESTION 4
? IS IT FUSIBLE
```

Now it takes the user through the long process of encoding the expertise:

```
PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  PLEONASTE  <


ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY


   > IS IT HARD? Y


   >
DOES IT CONTAIN STREAKS OF A DIFFERENT
COLOR FROM THE MAIN COLOR? N


   > IS ITS SPECIFIC GRAVITY ABOVE 5? N


   > IS IT FUSIBLE? N
```

It does this for the rest of the minerals, limonite, sylvanite and all. Finally, it reports its findings:

```
THIS IS MY EXPERT BASE:
```

```
PLEONASTE --- 1

LIMONITE --- 8

IODYRITE --- 12

IRIDOSMINE --- 7

SYLVANITE --- 13
```

Let's put it into action and see how well it does:

```
PLEASE THINK OF ONE OF THE FOLLOWING
  PLEONASTE
   LIMONITE
    IODYRITE
     IRIDOSMINE
      OR SYLVANITE


PLEASE ENTER 'Y' OR 'N'...


IS IT HARD
? N


DOES IT CONTAIN STREAKS OF A DIFFERENT
COLOR FROM THE MAIN COLOR
? N



IS ITS SPECIFIC GRAVITY ABOVE 5
? N
```

```
IS IT FUSIBLE
? Y
        > MY RESULT WAS 8


        > YOU WERE THINKING
          OF LIMONITE
```

In a matter of minutes, X-SPURT has acquired knowledge which allows me, as a total non-expert in that field, to make use of expert judgement in a practical situation. X-SPURT allows itself a little bit of fallibility, with I CANNOT IDENTIFY IT if the total it has reached, while adding up your answers, does not tally with any of your input (line 380).

You can see that it tells you its tally after each run, so you can keep track of what it is doing. If you wish to impress people with your expert system, you'll probably enhance the impression it creates if the 'works' are not so publicly displayed.

Here's the listing of X-SPURT:

```
10 REM X-SPURT
20 GOSUB 940:REM INITIALISE
30 GOSUB 450:REM 'GAIN EXPERTISE'
40 PEN 1:GOSUB 120:REM DEMONSTRATE EXPER
TISE
50 GOSUB 1060
60 PEN 2:PRINT "PRESS ";:PEN 3:PRINT "<E
NTER>";:PEN 2:PRINT " FOR ANOTHER RUN, O
R"
70 PRINT "ANY KEY AND THEN ";:PEN 3:PRIN
T "<ENTER>";:PEN 2:PRINT " TO QUIT"
80 INPUT Q$
90 IF Q$="" THEN 40
100 END
110 REM ------------------------
120 REM DEMONSTRATE EXPERTISE
130 CLS
140 GOSUB 1060
150 PEN 3:PRINT "PLEASE THINK OF ONE OF
THE FOLLOWING":PEN 2
160 FOR J=1 TO OUTCOMES
```

```
170 PRINT TAB(J+2);
180 IF J=OUTCOMES THEN PRINT "OR ";
190 PRINT A$(J)
200 NEXT J
210 GOSUB 1060
220 RESULT=0
230 X=0.5
240 PRINT "PLEASE ENTER 'Y' OR 'N'.."
250 FOR J=1 TO FACTORS
260 X=X+X
270 GOSUB 1060
280 PRINT B$(J)
290 INPUT E$:E$=UPPER$(E$)
300 IF E$<>"N" THEN RESULT=RESULT+X
310 NEXT J
320 PEN 1:PRINT TAB(8);"MY RESULT WAS"RE
SULT
330 GOSUB 1060
340 M=0
350 M=M+1
360 IF D(M)=RESULT THEN 400
370 IF M<OUTCOMES THEN 350
380 PRINT TAB(8);"I CANNOT IDENTIFY IT"
390 RETURN
400 PRINT TAB(8);"YOU WERE THINKING"
410 PRINT TAB(10);"OF ";A$(M)
420 GOTO 390
430 RETURN
440 REM ---------------------
450 REM FILL ARRAYS
460 PEN 2:PRINT TAB(20-LEN(N$)/2);N$:PEN
 3
470 GOSUB 1060
480 REM -- GET OUTCOME NAMES --
490 FOR J=1 TO OUTCOMES
500 GOSUB 1060
510 PRINT "WHAT IS OUTCOME";J;
520 INPUT A$(J):A$(J)=UPPER$(A$(J))
530 NEXT J
540 CLS
550 REM -- GET QUESTIONS TO BE ASKED --
560 PEN 1:FOR J=1 TO FACTORS
570 GOSUB 1060
580 PRINT "PLEASE ENTER QUESTION"J
```

```
590 INPUT B$(J):B$(J)=UPPER$(B$(J))
600 NEXT J
610 CLS
620 REM -- ACQUIRE EXPERTISE --
630 PEN 2:FOR J=1 TO OUTCOMES
640 CLS
650 GOSUB 1060
660 PRINT "PLEASE ANSWER THE FOLLOWING Q
UESTION"
670 PRINT "FOR AN OUTCOME OF >  ";A$(J);
"  <"
680 GOSUB 1060
690 PEN 3:PRINT "ENTER 'Y' FOR 'YES' ANS
WER"
700 PRINT "   OR 'N' FOR A 'NO' REPLY":P
EN 1
710 X=0.5
720 FOR K=1 TO FACTORS
730 X=X+X
740 GOSUB 1060
750 PRINT TAB(4);"> ";B$(K);
760 MULTI=0
770 INPUT Y$:Y$=UPPER$(Y$)
780 IF Y$<>"N" THEN MULTI=1
790 D(J)=D(J)+X*MULTI:REM COMPILE
                          EXPERT BASE
800 NEXT K
810 NEXT J
820 CLS
830 PEN 3:PRINT "THIS IS MY EXPERT BASE:
":PEN 2
840 FOR J=1 TO OUTCOMES
850 GOSUB 1060
860 PRINT A$(J);" ---"D(J)
879 NEXT J
880 GOSUB 1060
890 PEN 3:PRINT TAB(8);"PRESS <ENTER>"
900 INPUT Q$
910 CLS
920 RETURN
930 REM ----------------------
940 REM INITIALISATION
950 MODE 1:INK 0,26:BORDER 26:INK 1,2:IN
K 2,6:INK 3,0:PEN 1
```

```
960 INPUT "NAME OF SYSTEM";N$:N$=UPPER$(
N$)
970 GOSUB 1060
980 PEN 2:INPUT "NUMBER OF OUTCOMES";OUT
COMES
990 GOSUB 1060
1000 PEN 3:INPUT "NUMBER OF FACTORS TO B
E CONSIDERED";FACTORS
1010 DIM A$(OUTCOMES),B$(FACTORS)
1020 DIM D(OUTCOMES)
1030 CLS
1040 RETURN
1050 REM ------------------------
1060 PRINT:PRINT
1070 RETURN
```

Choosing a Chip

It would be very inconvenient if we had to educate our expert, as we did X-SPURT, every time we wanted to make use of the expertise. It is unlikely, in a real situation, a totally 'soft' expert would be needed. This next program shows an expert body of knowledge — the ability to distinguish between several chips (of the silicon variety) — encoded in DATA statements.

Here's the program in action:

```
THIS IS MY EXPERT BASE:

TMS 9940 (NMOS) --- 44

68000 (NMOS) --- 12

9940 (I3L) --- 56

MN1610 (NMOS) --- 46
```

```
          8086 --- 60


          Z8001 --- 28


I CAN IDENTIFY FROM THE FOLLOWING:
  TMS 9940 (NMOS)
   68000 (NMOS)
    9940 (I3L)
     MN1610 (NMOS)
      8086
       OR Z8001
```

After telling you what it can do, the program asks you to answer a number of questions in relation to the chip you are trying to identify. It will then tell you the name of the chip:

```
PLEASE ENTER 'Y' OR 'N'...


IS THE WORD SIZE 32 BITS
? N


DOES IT ADDRESS 64K
? N


IS THE CLOCK RATE 5MHz OR LESS
? Y


IS SHORTEST OBEY 3 MICROSECONDS OR LESS
? Y


DOES INSTRUCTION SET CONTAIN MORE THAN
71 INSTRUCTIONS
? N
```

```
IS THE PACKAGE 40 PIN DIP
? N
        > MY RESULT WAS 12


        > THE ONE YOU HAVE
          IS 68000 (NMOS)
```

The expert base came from another expert, Ken Ozanne, whose expertise was once again encoded in a book *(The Interface Computer Encyclopedia*, London: Interface Publications, 1983). Once the information is locked into the DATA statements in the program, it is ready for use at any time.

Even if you have no desire whatsoever to identify chips, you can still make use of the expert system encoded in this program. As you can see, the variables OUTCOMES and FACTORS are assigned in lines 610 and 620, and these are used to dimension the arrays in line 630. Change the variables to the outcomes and factors you have, modify the DATA statements, and you have your very own expert system, ready to help you. The crucial number which the system uses to identify the chip (or to isolate whichever outcome you want) is held in the DATA statements immediately following the name of the chip.

To work out the numbers, I set up a chart like the following. It is simple to do the same for your subject:

OUTCOME	*FACTOR*						*TOTAL*
	1	*2*	*3*	*4*	*5*	*6*	
TMS 9940	*0*	*0*	*1*	*1*	*0*	*1*	*44*
68000	*0*	*0*	*1*	*1*	*0*	*0*	*12*
9940 13L	*0*	*0*	*0*	*1*	*1*	*1*	*56*
MN1610	*0*	*1*	*1*	*1*	*0*	*1*	*46*
8086	*0*	*0*	*1*	*1*	*1*	*1*	*60*
Z8001	*0*	*0*	*1*	*1*	*1*	*0*	*28*

Here's the program, so you can create the expert of your choice:

```
10 REM CHIP CHOICE
20 GOSUB 580:REM INITIALISE
30 GOSUB 450:REM SHOW BASE CONTENTS
40 GOSUB 120:REM IDENTIFY CHIP
50 GOSUB 720
60 PEN 3:PRINT "PRESS <ENTER> FOR ANOTHE
R CHIP, OR"
70 PRINT "ANY KEY AND THEN <ENTER> TO QU
IT"
80 INPUT Q$
90 IF Q$="" THEN 40
100 PEN 1:END
110 REM ---------------------
120 REM IDENTIFY CHIP
130 CLS
140 GOSUB 720
150 PEN 1:PRINT "I CAN IDENTIFY FROM THE
 FOLLOWING:"
160 PEN 2:FOR J=1 TO OUTCOMES
170 PRINT TAB(J+2);
180 IF J=OUTCOMES THEN PRINT "OR ";
190 PRINT A$(J)
200 NEXT J
210 GOSUB 720
220 RESULT=0
230 X=0.5
240 PEN 1:PRINT "PLEASE ENTER 'Y' OR 'N'
..."
250 FOR J=1 TO FACTORS
260 X=X+X
270 GOSUB 720
280 PEN 2:PRINT B$(J)
290 PEN 3:PRINT TAB(7);"> ";
295 E$=INKEY$:E$=UPPER$(E$):IF E$<>"Y" A
ND E$<>"N" THEN 295
300 PEN 1:PRINT E$:IF E$<>"N" THEN RESUL
T=RESULT+X
310 NEXT J
320 PEN 3:PRINT:PRINT TAB(8);"> ";:PEN 1
:PRINT "MY RESULT WAS";:PEN 2:PRINT RESU
LT
```

```
330 GOSUB 720
340 M=0
350 M=M+1
360 IF D(M)=RESULT THEN 400
370 IF M<OUTCOMES THEN 350
380 PEN 3:PRINT TAB(8);"> I CANNOT IDENT
IFY IT<"
390 RETURN
400 PEN 1:PRINT TAB(8);"> THE ONE YOU HA
VE"
410 PRINT TAB(10);"IS ";:PEN 2:PRINT A$(
M)
420 GOSUB 390
430 RETURN
440 REM --------------------
450 REM SHOW CONTENTS OF BASE
460 CLS
470 PEN 1:PRINT "THIS IS MY EXPERT BASE:
":PEN 2
480 FOR J=1 TO OUTCOMES
490 GOSUB 720
500 PRINT A$(J);" ---"D(J)
510 NEXT J
520 GOSUB 720
530 PEN 3:PRINT TAB(8);"PRESS <ENTER>"
540 WHILE INKEY$="":WEND
550 CLS
560 RETURN
570 REM ----------------------
580 REM INITIALISATION
590 MODE 1:INK 0,1:BORDER 1:INK 1,26:INK
 2,24:INK 3,8,6:SPEED INK 2,2
600 RESTORE
610 OUTCOMES=6
620 FACTORS=6
630 DIM A$(OUTCOMES),B$(FACTORS),D(OUTCO
MES)
640 FOR J=1 TO OUTCOMES
650 READ A$(J),D(J)
660 NEXT J
670 FOR J=1 TO FACTORS
680 READ B$(J)
690 NEXT J
700 RETURN
```

```
710 REM ----------------------------
720 PRINT:PRINT
730 RETURN
740 REM ----------------------------
750 REM -- OUTCOMES (CHIP TITLES) --
760 DATA "TMS 9940 (NMOS)",44,"68000 (NM
OS)",12
770 DATA "9940 (13L)",56,"MN1610 (NMOS)"
,46
780 DATA "8086",60,"Z8001",28
790 REM -- QUESTIONS --
800 DATA "IS THE WORD SIZE 32 BITS"
810 DATA "DOES IT ADDRESS 64K"
820 DATA "IS THE CLOCK RATE 5 MHz OR LES
S"
830 DATA "IS SHORTEST OBEY 3 MICROSECOND
S          OR LESS"
840 DATA "DOES INSTRUCTION SET CONTAIN M
ORE THAN  71 INSTRUCTIONS"
850 DATA "IS THE PACKAGE 40 PIN DIP"
```

Part Thirteen — Self-Learning Systems

You'll recall, in the second system we looked at in this section, that the program X-SPURT allowed you to enter expertise on any subject. Once you'd fed it in, the program was ready to be your expert on the subject you had chosen.

However, it had one disadvantage. It demanded that you run through each of the factors, for each of the outcomes, in order to acquire a knowledge base from which it could work.

Our next program, SELFLEARN, does not require the same kind of spoonfeeding which was needed with X-SPURT. Here it is in action:

```
HOW MANY FACTORS? 3


ENTER FACTOR 1
? WINGS


ENTER FACTOR 2
? PAIR OF EYES


ENTER FACTOR 3
? EATS WORMS



ENTER OUTCOME 1
? SPARROW


ENTER OUTCOME 2
? HUMAN
```

Once you have this information in place, you can run the program, and it will proceed to teach itself how to tell the difference between various outcomes:

```
NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? N
                                > 0
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? N
                                > 0
                              >BRAYN= 0
OUTCOME IS SPARROW
IS THIS CORRECT? ('Y' OR 'N')
? N


NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? Y

                                > 1
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? Y
                                > 1
                              >BRAYN=-1
OUTCOME IS HUMAN
IS THIS CORRECT? ('Y' OR 'N')
? N
```

For a while it will get things wrong, as you see above, but then will start getting some correct guesses:

```
NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? Y
                                > 1
```

```
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? Y
                                  > 1
                               >BRAYN= 2
OUTCOME IS SPARROW
IS THIS CORRECT? ('Y' OR 'N')
? Y

NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? N
                                  > 0
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? N
                                  > 0
                               >BRAYN=-1
OUTCOME IS HUMAN
IS THIS CORRECT? ('Y' OR 'N')
? Y
```

In due course it will become infallible:

```
NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? Y
                                  > 1
                               >BRAYN= 1
OUTCOME IS SPARROW
IS THIS CORRECT? ('Y' OR 'N')
? Y
```

How It Works

The important thing (and the major limitation) of this program is that it can only distinguish between two outcomes (such as SPARROW and MAN in our example). The program starts with the assumption that its total (the variable BRAYN) will be either greater than or equal to zero, or less than zero. The actual value BRAYN achieves does not matter.

When you first run it, the program asks for the raw information it will need. Then, each time through the J loop, SELFLEARN begins by filling each element of the C array (there is one element for each FACT) with zero. It then proceeds to print up the factors, one by one, asking you to comment 'Y' or 'N' on whether they refer to the outcome you have thought of: If you say 'Y' then that element of the C array is set to one. Once you've been through this loop, BRAYN works out a total for that outcome, with the code from 230 to 270.

If you look at the listing carefully, you'll see that the very first time this loop is run, BRAYN will equal zero (because all of those C(J)'s have been multiplied by D(J)'s, and every D(J) starts out equalling zero). This means, the very first time you run the program, it will give you option one (that is A$(1), the first outcome you entered), as BRAYN will be equal to zero. SELFLEARN then asks if that was correct. If you tell it that it is correct, it does not modify its information, because — in its present condition — it will give the same answer next time the same information is presented. If, however, you tell it that it was wrong, it will go through the next loop, modifying the values of D(J) using both the C(J) values you gave, and by use of the variable EX. If you look back to lines 280 and 290, you'll see EX is set to -1 if the outcome it thought of was A$(1), and to 1 if it thought of A$(2).

D(J) is the vital component of the loop 240 to 260 helps determine the value of BRAYN, so this must be modified if the program gave the wrong result. Once it has made its changes to D(J), using both the values of the elements of the C array (which can, you'll see from lines 60 and 200, only have values of one or zero), the program returns for another try. As you'll see, it soon becomes infallible.

Here is the listing:

```
10 REM SELFLEARN
20 GOSUB 400:REM INITIALISE
30 REM -- MAJOR LEARNING LOOP --
```

```
40 CLS
50 FOR J=1 TO FACT
60 C(J)=0
70 NEXT J
80 PRINT
90 GOSUB 130
100 GOTO 40
110 REM --------------------
120 REM DEMONSTRATION TIME
130 PEN 3:PRINT "NOW I WILL DEMONSTRATE
MY EXPERTISE...":PRINT
140 PEN 2:PRINT "THINK OF ONE OF THE OUT
COMES":PRINT
150 FOR J=1 TO FACT
160 PEN 1
170 PRINT "IS ";B$(J);" TRUE? ('Y' OR 'N
')"
180 Z$="":WHILE Z$<>"Y" AND Z$<>"N":Z$=I
NKEY$:Z$=UPPER$(Z$)
190 WEND:SOUND 5,65
200 IF Z$<>"Y" THEN C(J)=1
210 PEN 3:PRINT TAB(29);"> ";C(J)
220 NEXT J
230 BRAYN=0
240 FOR J=1 TO FACT
250 BRAYN=BRAYN+C(J)*D(J)
260 NEXT J
270 PEN 1:PRINT TAB(25);">BRAYN="BRAYN
280 PEN 3:IF BRAYN>=0 THEN PRINT "OUTCOM
E IS ";A$(1):EX=-1
290 IF BRAYN<0 THEN PRINT "OUTCOME IS ";
A$(2):EX=1
300 PEN 1:PRINT:PRINT "IS THIS CORRECT (
'Y' OR 'N')"
310 Z$="":WHILE Z$<>"N" AND Z$<>"Y":Z$=I
NKEY$:Z$=UPPER$(Z$)
320 WEND:SOUND 5, 176
340 IF Z$="Y" THEN 380
350 FOR J=1 TO FACT
360 D(J)=D(J)+EX*C(J)
370 NEXT J
380 RETURN
390 REM ----------------------
```

```
400 REM INITIALISATION
410 MODE 1:INK 0,0:BORDER 0:INK 1,6:INK
2,24:INK 3,18
420 OTCO=2:REM NUMBER OF OUTCOMES
430 PRINT:PRINT
440 PEN 1:INPUT "HOW MANY FACTORS";FACT
450 DIM A$(OTCO):REM NAMES OF OUTCOMES
460 DIM B$(FACT):REM NAMES OF FACTORS
470 DIM C(FACT),D(FACT)
480 CLS
490 FOR J=1 TO FACT
500 PRINT:PRINT
510 PEN 2:PRINT "ENTER FACTOR"J
520 PEN 3:INPUT "  ";B$(J)
530 NEXT J
540 CLS
550 FOR J=1 TO OTCO
560 PRINT
570 PRINT
580 PEN 1:PRINT "ENTER OUTCOME"J
590 PEN 3:INPUT "  ";A$(J)
600 NEXT J
610 RETURN
```

More Than Two Alternatives

Although it is facinating to have an expert system which teaches itself in this way, it is a major drawback to be able to choose from just two alternatives. Our next program, MULTI-SELF-LEARN, is designed to allow for more than two outcomes.

The program starts much as you would, by now, expect:

```
HOW MANY OUTCOMES? 3

HOW MANY FACTORS? 3

PLEASE ENTER NAME OF OUTCOME 1
? HUMAN
PLEASE ENTER NAME OF OUTCOME 2
? HORSE
PLEASE ENTER NAME OF OUTCOME 3
? SPARROW
```

```
PLEASE ENTER NAME OF FACTOR 1
? A SINGLE PAIR OF LEGS
PLEASE ENTER NAME OF FACTOR 2
? CAN FLY UNAIDED
PLEASE ENTER NAME OF FACTOR 3
? BREATHES OXYGEN


THESE ARE THE POSSIBLE OUTCOMES:

HUMAN
 HORSE
  SPARROW

PLEASE THINK OF ONE OF THEM
```

However, when you run it, you'll see that it asks you questions, then makes a guess as to what you were thinking of. If it was wrong, it asks you what the correct answer should have been:

```
PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF

A SINGLE PAIR OF LEGS
? Y
CAN FLY UNAIDED
? N
BREATHES OXYGEN
? Y
YOU WERE THINKING OF HORSE

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? N
WHAT SHOULD THE ANSWER HAVE BEEN
? HUMAN
```

Run it long enough (which is not very long with only three outcomes, and three factors) and it gets them right every time:

```
PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF

A SINGLE PAIR OF LEGS
? N
CAN FLY UNAIDED
? N
BREATHES OXYGEN
? Y
YOU WERE THINKING OF HORSE

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? Y


PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF

A SINGLE PAIR OF LEGS
? Y
CAN FLY UNAIDED
? Y
BREATHES OXYGEN
? Y
YOU WERE THINKING OF SPARROW

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? Y
```

The important part of this program lies between lines 150 and 520. The first section of this accepts the 'Y' answers, and increments a variable called COUNT in terms of your answer (adding 1 for the first 'Y', 2 for the next, 4 for the next, then 8, 16, 32 and so on).

After the program has been running for a while, it will have assigned values to many elements of the B array. B(1) will be the total when A$(1) is the outcome, B(6) will be the total for an outcome of A$(6) and so on. A small loop is traversed after the 'Y' answers are given, to see if the total obtained equals any previously-stored B(J) value. If it does, then the variable X is set equal to the relevant J.

If such a value has been assigned, X is no longer equal to zero, and the system has made a decision. If no definite answer has been obtained here, the computer generates a random number between 1 and the number of outcomes, in order to make a guess. But it is not enough to then say "WERE YOU THINKING OF"; A$ (random number generated). Although your program may not yet have assigned an A$(n) to the n total you received, it may well have assigned some elements of A$. Therefore it can, and must, reject some 'guesses' produced by the random number generator.

If the variable FLAG equals anything except zero, then the guess (the element of A$ represented by the randomly-generated value of X) cannot be used, as the system already knows that answer is wrong. Therefore, using line 400, it goes back to choose a new value for X. Having done so, the program checks on its guess. If the guess was correct, the system uses the loop from 490 to 510 to find out which element of A$ corresponds to the total generated in that run. This ensures that, when it meets the same total next time, it will be able to identify the relevant element of A$.

Here's the listing, so you can set up a major expert system, which will teach itself:

```
10 REM MULTI-SELF-LEARN
20 GOSUB 550:REM INITIALISE
30 CLS
40 PEN 1:PRINT "THESE ARE THE POSSIBLE O
UTCOMES:"
50 PRINT:PEN 2
60 FOR J=1 TO OTCO
70 PRINT TAB(J);A$(J)
80 NEXT J
90 PRINT:PEN 3
100 PRINT "PLEASE THINK OF ONE OF THEM"
110 PRINT
120 PRINT "PRESS <ENTER> WHEN YOU ARE RE
ADY"
130 WHILE INKEY$="":WEND:SOUND 5,540
140 CLS
150 PEN 1:PRINT "PLEASE ANSWER 'Y' OR 'N
' FOR EACH OF"
160 PRINT "THE FOLLOWING IN RESPECT OF"
170 PRINT "THE OUTCOME YOU HAVE THOUGHT
OF"
```

```
180 PRINT:PEN 2
190 COUNT=0
200 X=0.5
210 FOR J=1 TO FACT
220 X=X+X
230 PRINT F$(J)
240 Z$="":WHILE Z$<>"Y" AND Z$<>"N":Z$=I
NKEY$:Z$=UPPER$(Z$)
250 WEND:SOUND 5,240+RND*30
260 IF Z$="Y" THEN COUNT=COUNT+X
270 NEXT J
280 X=0
290 FOR J=1 TO OTCO
300 IF COUNT=B(J) THEN X=J
310 NEXT J
320 IF X<>0 THEN 410
330 X=INT(RND*OTCO)+1
340 REM -- REJECTS ANSWERS KNOWN
                        TO BE WRONG --
350 FLAG=0
360 FOR J=1 TO OTCO
370 IF B(J)=0 THEN 390
380 IF X=J AND COUNT<>B(J) THEN FLAG=1
390 NEXT J
400 IF FLAG=1 THEN 330
410 PEN 3:PRINT "YOU WERE THINKING OF ";
:PEN 2:PRINT A$(X)
420 PRINT:PEN 1
430 PRINT "ENTER 'Y' IF I'M RIGHT, 'N' I
F WRONG"
440 Z$="":WHILE Z$<>"Y" AND Z$<>"N":Z$=I
NKEY$:Z$=UPPER$(Z$)
450 WEND:SOUND 5,66
460 IF Z$="Y" THEN B(X)=COUNT:GOTO 30
470 PEN 2:PRINT "WHAT SHOULD THE ANSWER
HAVE BEEN?"
480 PEN 1:INPUT Z$
490 FOR J=1 TO OTCO
500 IF A$(J)=Z$ THEN B(J)=COUNT
510 NEXT J
520 GOTO 30
530 END
540 REM ------------------------
550 REM INITIALISATION
```

```
560 MODE 1:INK 0,3:BORDER 3:INK 1,24:INK
 2,26:INK 3,18
570 PEN 1:INPUT "HOW MANY OUTCOMES";OTCO
580 PRINT
590 PEN 2:INPUT "HOW MANY FACTORS";FACT
600 CLS
610 X=OTCO+FACT
620 DIM A$(OTCO)
630 DIM F$(FACT),B(X)
640 FOR J=1 TO OTCO
650 PEN 3:PRINT "PLEASE ENTER THE NAME O
F OUTCOME";:PEN 1:PRINT J
660 PEN 2:INPUT "  ";A$(J)
670 NEXT J
680 CLS
690 FOR J=1 TO FACT
700 PEN 3:PRINT "PLEASE ENTER NAME OF FA
CTOR";:PEN 1:PRINT J
710 PEN 2:INPUT "  ";F$(J)
720 NEXT J
730 RETURN
```

No Need For Correction

Our final program in this section is a variation of the one you have just been studying. The only difference between this one, and the preceding one, is that this one does not need to be told what the correct answer should have been if it makes a mistake. The program works out for itself, fairly quickly how to distinguish between the various values.

I have not renumbered this program, so it will be easy to modify MULTI-SELF-LEARN so that it becomes MULTI-SELF-LEARN-2. Here is the program up and running:

```
HOW MANY OUTCOMES? 5

HOW MANY FACTORS? 5

PLEASE ENTER NAME OF OUTCOME 1
? DOG
PLEASE ENTER NAME OF OUTCOME 2
? HORSE
PLEASE ENTER NAME OF OUTCOME 3
? SHEEP
```

```
PLEASE ENTER NAME OF OUTCOME 4
? CROW
PLEASE ENTER NAME OF OUTCOME 5
? HUMAN

PLEASE ENTER NAME OF FACTOR 1
? POWER OF SPEECH
PLEASE ENTER NAME OF FACTOR 2
? BARKS
PLEASE ENTER NAME OF FACTOR 3
? ABLE TO FLY
PLEASE ENTER NAME OF FACTOR 4
? FOUR LEGS
PLEASE ENTER NAME OF FACTOR 5
? PRODUCES WOOL
```

Once the knowledge base is in place, the program proceeds as follows:

```
THESE ARE THE POSSIBLE OUTCOMES:

DOG
 HORSE
  SHEEP
   CROW
    HUMAN

PLEASE THINK OF ONE OF THEM

PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF


POWER OF SPEECH
? N
BARKS
? Y
ABLE TO FLY
? N
FOUR LEGS
? Y
PRODUCES WOOL
? N
```

```
YOU WERE THINKING OF CROW

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? N
```

While most of its answers will be wrong when the run begins, the correct guesses will become more frequent as time goes on:

```
PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF

POWER OF SPEECH
? N
BARKS
? N
ABLE TO FLY
? Y
FOUR LEGS
? N
PRODUCES WOOL
? N
YOU WERE THINKING OF CROW

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? Y
```

Eventually, it will not make any mistakes. It works in much the same way as the previous program, assigning values to the elements of the B array. However, this time it has an array called C, which ensures that it will never make the same wrong guess from the same total. This acts to increasingly limit the possible outcomes which could be obtained, so the number of things it can guess from is reduced each time. It does not take it long to build up a 'world view' which ensures it will be right every time. The number of trials it will take to obtain perfection depends, of course, on the number of outcomes.

After the program has managed to teach itself to distinguish between the outcomes without error, it will be holding a knowledge base it refers to for each subsequent trial. This is the base that MULTI-SELF-LEARN-2 built up during that run:

```
J= 1   B(J)= 10    A$(J)=DOG
J= 2   B(J)= 8    A$(J)=HORSE
J= 3   B(J)= 24    A$(J)=SHEEP
J= 4   B(J)= 4    A$(J)=CROW
J= 5   B(J)= 1    A$(J)=HUMAN
```

And this is the listing:

```
10 REM MULTI-SELF-LEARN
20 GOSUB 550:REM INITIALISE
30 CLS
40 PEN 1:PRINT "THESE ARE THE POSSIBLE O
UTCOMES:"
50 PRINT:PEN 2
60 FOR J=1 TO OTCO
70 PRINT TAB(J);A$(J)
80 NEXT J
90 PRINT:PEN 3
100 PRINT "PLEASE THINK OF ONE OF THEM"
110 PRINT
120 PRINT "PRESS <ENTER> WHEN YOU ARE RE
ADY"
130 WHILE INKEY$="":WEND:SOUND 5,540
140 CLS
150 PEN 1:PRINT "PLEASE ANSWER 'Y' OR 'N
' FOR EACH OF"
160 PRINT "THE FOLLOWING IN RESPECT OF"
170 PRINT "THE OUTCOME YOU HAVE THOUGHT
OF"
180 PRINT:PEN 2
190 COUNT=0
200 X=0.5
210 FOR J=1 TO FACT
220 X=X+X
230 PRINT F$(J)
240 Z$="":WHILE Z$<>"Y" AND Z$<>"N":Z$=I
NKEY$:Z$=UPPER$(Z$)
250 WEND:SOUND 5,240+RND*30
260 IF Z$="Y" THEN COUNT=COUNT+X
270 NEXT J
280 X=0
290 FOR J=1 TO OTCO
300 IF COUNT=B(J) THEN X=J
310 NEXT J
```

```
320 IF X<>0 THEN 410
330 X=INT(RND*OTCO)+1
340 REM -- REJECTS ANSWERS KNOWN
                        TO BE WRONG --
350 FLAG=0
360 FOR J=1 TO OTCO
370 IF B(J)=0 THEN 390
380 IF X=J AND COUNT<>B(J) THEN FLAG=1
385 IF C(X)=COUNT OR D(X)=COUNT THEN FLA
G=1
390 NEXT J
400 IF FLAG=1 THEN 330
410 PEN 3:PRINT "YOU WERE THINKING OF ";
:PEN 2:PRINT A$(X)
420 PRINT:PEN 1
430 PRINT "ENTER 'Y' IF I'M RIGHT, 'N' I
F WRONG"
440 Z$="":WHILE Z$<>"Y" AND Z$<>"N":Z$=I
NKEY$:Z$=UPPER$(Z$)
450 WEND:SOUND 5,66
460 IF Z$="Y" THEN B(X)=COUNT:GOTO 30
470 IF C(X)=0 THEN C(X)=COUNT:GOTO 30
480 D(X)=COUNT:GOTO 30
490 FOR J=1 TO OTCO
500 IF A$(J)=Z$ THEN B(J)=COUNT
510 NEXT J
520 GOTO 30
530 END
540 REM ------------------------
550 REM INITIALISATION
560 MODE 1:INK 0,3:BORDER 3:INK 1,24:INK
 2,26:INK 3,18
570 PEN 1:INPUT "HOW MANY OUTCOMES";OTCO
580 PRINT
590 PEN 2:INPUT "HOW MANY FACTORS";FACT
600 CLS
610 X=OTCO+FACT
620 DIM A$(OTCO)
630 DIM F$(FACT),B(X),C(X),D(X)
640 FOR J=1 TO OTCO
650 PEN 3:PRINT "PLEASE ENTER THE NAME O
F OUTCOME";:PEN 1:PRINT J
660 PEN 2:INPUT "  ";A$(J)
```

```
670 NEXT J
680 CLS
690 FOR J=1 TO FACT
700 PEN 3:PRINT "PLEASE ENTER NAME OF FA
CTOR";:PEN 1:PRINT J
710 PEN 2:INPUT "   ";F$(J)
720 NEXT J
730 RETURN
```

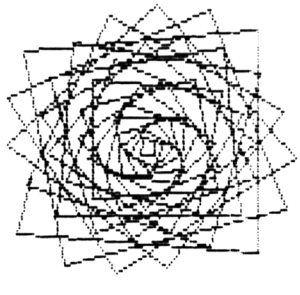
Section Four
Logo-K

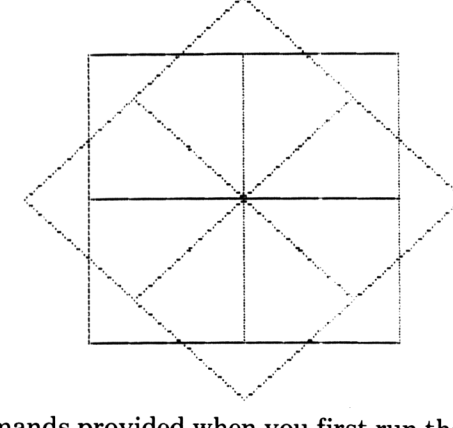
The computer language Logo evolved from Lisp, one of the most important languages currently used in Artificial Intelligence research. In this section of the book, we look at Logo. Using the program we give here, you and your Amstrad can now explore the world of Logo graphics. Logo is regarded by many as the ideal language to introduce computer programming to beginners. More experienced programmers also find Logo a delight to work with.

The Logo 'world' is related to the real world. Move forward, turn right, move forward again, and you'll have traced out the same sort of path the Logo 'turtle' traces out on the screen when running the *Logo-K* program from this booklet.

The 'turtle graphics' aspect of Logo, which *Logo-K* supports, introduces fundamental programming concepts quickly and easily, and gives you onscreen response to your commands. You can see almost instantly how well you're doing.

Using the program, and following through the steps in this book, allows you to quickly become a master of Logo graphics. And once you're familiar





with the 'primitives' (the 'raw' commands provided when you first run the program), you'll be delighted to discover how easy it is to invent your own words, which the Logo turtle will happily obey.

Under the leadership of Seymour Papert (author of the fundamental Logo text, *Mindstorms*), a team involved in education and computers developed Logo to act as an ideal tool for introducing computers to children. Logo is a strong and healthy branch of the family tree of computer languages which began with Lisp, and includes Prolog and Smalltalk. Logo shares with these languages the remarkable 'list processing' capabilities which have made them prime languages for artificial intelligence research.

The *Logo-K* program follows the Logo Computer Systems, Inc. (LCSI) vocabulary, providing a significant number of the fundamental primitives, and following standard Logo syntax. This means you can transfer your learning from our program to just about any Logo system including, with some small changes, the Dr. Logo provided with the disc CP/M package. And you'll find that, in many cases, programs written in *Logo-K* will run as quickly as they do in Dr. Logo.

Entering and Running the Program

The complete *Logo-K* program is generously supplied with REM statements, to make it easy to find out what each part of the program does. Once you have the program in place, you simply type in RUN, and then ENTER. The screen will clear, there will be a dot in the middle of the picture (this is the 'turtle') and an input prompt will appear in the bottom left hand corner of the screen. You will then be in Logo! In effect, your Amstrad will be running Logo, instread of BASIC. The Logo turtle is now waiting for your orders.

```
10 REM        LOGO-K
20 GOSUB 1880:' INITIALISE
30 ' **********************
40 WHILE NOT FINISHED
50 GOSUB 340:' INPUT
60 WEND
70 ' **********************
80 REM CHANGE PROCEDURES INTO THEIR
                MEANINGS
90 FOR K=1 TO PROC
100 FOR J=1 TO LEN(Z$)
110 IF MID$(Z$,J,LEN(B$(K)))=B$(K) THEN
Z$=LEFT$(Z$,J-1)+C$(K)+MID$(Z$,J+LEN(B$(
K)))
120 NEXT J
130 NEXT K
140 RETURN
```

```
150 ' **********************
160 REM LOOK FOR REPEAT
170 IF INSTR(Z$,"REPEAT")=0 THEN RETURN
180 J=INSTR(Z$,"REPEAT")
190 Z$=LEFT$(Z$,J-1)+MID$(Z$,J+7):GOSUB
210:REM        THIS DELETES WORD 'REPEAT'
200 GOTO 170
210 Q1=INSTR(Z$,"["):K=INSTR(Z$,"]")
220 IF K=0 THEN K=INSTR(Z$,"#")
230 R=VAL(MID$(Z$,J,Q1)):M=Q1+1
240 REPFLAG=R-1
250 K$="!"+MID$(Z$,M,M+K-2)+" "
260 EDFLAG=0
270 FOR JJ=1 TO LEN(K$)-1
280 IF MID$(K$,JJ,1)=" " THEN EDFLAG=EDF
LAG+1
290 NEXT JJ
300 EDCOUNT=EDFLAG
310 Z$=LEFT$(Z$,J-1)+K$+MID$(Z$,K+1)
320 RETURN
330 ' **********************
340 REM INPUT
350 COUNT=0
360 PEN 2:LOCATE 1,23:PRINT ">";:PEN 1:L
INE INPUT Z$:Z$=UPPER$(Z$):Z$=" "+Z$+" #
"
370 LOCATE 1,23:PRINT "

                        ":LOCATE 1,1
380 IF Z$="  #" OR Z$=" BYE #" THEN PEN
3:PRINT "OK, BYE FOR NOW...":LOCATE 1,23
:PEN 1:END
390 X$=MID$(Z$,2,LEN(Z$)-3)
400 IF PROC<>0 THEN GOSUB 90
410 IF LEN(Z$)>3 THEN IF LEFT$(Z$,4)=" T
O " THEN PEN 2:GOSUB 1710:PEN 1:RETURN
420 GOSUB 170:REM REPEAT?
430 FOR AZ=1 TO LEN(Z$)
440 IF MID$(Z$,AZ,1)<>" " THEN 500
450 COUNT=COUNT+1:KK=LEN(Z$):J=AZ
460 J=J+1
470 IF MID$(Z$,J,1)=" " THEN A$(COUNT)=M
ID$(Z$,AZ+1,AZ+J):M=J+1:GOTO 500
```

```
480 IF MID$(Z$,J,1)="#" THEN A$(COUNT)=M
ID$(Z$,M,M+J-1):GOTO 500
490 IF J<KK THEN 460
500 NEXT AZ
510 FOR Z=1 TO COUNT
520 IF LEFT$(A$(Z),1)="!" THEN A$(Z)=MID
$(A$(Z),2):ZED=Z
530 IF LEFT$(A$(Z),1)=" " THEN 710
540 IF LEFT$(A$(Z),7)="FORWARD" OR LEFT$
(A$(Z),2)="FD" THEN DIRFLAG=1:GOSUB 770
550 IF LEFT$(A$(Z),5)="BACK " OR LEFT$(A
$(Z),2)="BK" THEN DIRFLAG=-1:GOSUB 770
560 IF LEFT$(A$(Z),5)="RIGHT" OR LEFT$(A
$(Z),2)="RT" THEN TURNFLAG=1:GOSUB 850
570 IF LEFT$(A$(Z),4)="LEFT" OR LEFT$(A$
(Z),2)="LT" THEN TURNFLAG=-1:GOSUB 850
580 IF LEFT$(A$(Z),4)="HOME" THEN GOSUB
940
590 IF LEFT$(A$(Z),11)="CLEARSCREEN" OR
LEFT$(A$(Z),2)="CS" THEN GOSUB 1010
600 IF LEFT$(A$(Z),5)="CLEAN" THEN GOSUB
 1060
610 IF LEFT$(A$(Z),5)="PENUP" OR LEFT$(A
$(Z),2)="PU" THEN GOSUB 1100
620 IF LEFT$(A$(Z),7)="PENDOWN" OR LEFT$
(A$(Z),2)="PD" THEN GOSUB 1140
630 IF LEFT$(A$(Z),10)="SETHEADING" THEN
 GOSUB 1180
640 IF LEFT$(A$(Z),6)="SETPOS" THEN GOSU
B 1230
650 IF LEFT$(A$(Z),4)="SETX" OR LEFT$(A$
(Z),4)="SETY" THEN GOSUB 1300
660 IF LEFT$(A$(Z),4)="XCOR" OR LEFT$(A$
(Z),4)="YCOR" THEN GOSUB 1360
670 IF LEFT$(A$(Z),3)="POS" THEN GOSUB 1
420
680 IF LEFT$(A$(Z),7)="HEADING" THEN GOS
UB 1510
690 IF LEFT$(A$(Z),10)="SETSCRUNCH" THEN
 GOSUB 1570
700 IF LEFT$(A$(Z),4)="PEN " THEN GOSUB
1650
710 IF EDFLAG>0 THEN Z=Z+1:EDFLAG=EDFLAG
-1:GOTO 540
```

```
720 IF REPFLAG>0 THEN REPFLAG=REPFLAG-1:
Z=ZED:EDFLAG=EDCOUNT:GOTO 540
730 NEXT Z
740 RETURN
750 ' **********************
760 REM      FORWARD/BACK
770 DISTANCE=VAL(A$(Z+1))
780 ACROSS=RATIO*(SIN(FN R(ANGLE)))*DIST
ANCE*DIRFLAG
790 DOWN=COS(FN R(ANGLE))*DISTANCE*DIRFL
AG
800 IF TEMPCOL=99 THEN DRAWR ACROSS,DOWN
 ELSE MOVER ACROSS,DOWN
810 Z=Z+1
820 RETURN
830 ' **********************
840 REM      RIGHT/LEFT
850 TURN=VAL(A$(Z+1))
860 ANGLE=ANGLE+TURNFLAG*TURN
870 IF ANGLE>360 THEN ANGLE=ANGLE-360
880 IF ANGLE<0 THEN ANGLE=ANGLE+360
890 ANGLE=CINT(ANGLE)
900 Z=Z+1
910 RETURN
920 ' **********************
930 REM      HOME
940 ACROSS=320
950 DOWN=200
960 ANGLE=0
970 PLOT ACROSS,DOWN
980 RETURN
990 ' **********************
1000 REM      CLEARSCREEN
1010 CLS
1020 GOSUB 940:REM HOME
1030 RETURN
1040 ' **********************
1050 REM      CLEAN
1060 CLS
1070 RETURN
1080 ' **********************
1090 REM      PENUP
1100 TEMPCOL=0
1110 RETURN
```

```
1120 ' **********************
1130 REM      PENDOWN
1140 TEMPCOL=99
1150 RETURN
1160 ' **********************
1170 REM      SETHEADING
1180 ANGLE=VAL(A$(Z+1))
1190 Z=Z+1
1200 RETURN
1210 ' **********************
1220 REM      SETPOS
1230 ACROSS=VAL(A$(Z+1))
1240 DOWN=VAL(A$(Z+2))
1250 DRAWR ACROSS,DOWN
1260 Z=Z+2
1270 RETURN
1280 ' **********************
1290 REM      SETX/SETY
1300 IF MID$(A$(Z),4,1)="X" THEN ACROSS=
VAL(A$(Z+1)):DOWN=0 ELSE ACROSS=0:DOWN=V
AL(A$(Z+1))
1310 DRAWR ACROSS,DOWN
1320 Z=Z+1
1330 RETURN
1340 ' **********************
1350 REM      XCOR/YCOR
1360 LOCATE 1,1
1370 IF LEFT$(A$(Z),1)="X" THEN PRINT "X
 CO-ORDINATE IS";XPOS-159 ELSE PRINT "Y
CO-ORDINATE IS";YPOS-99
1380 GOSUB 1440
1390 RETURN
1400 ' **********************
1410 REM      POS
1420 LOCATE 1,1
1430 PRINT "TURTLE IS AT";XPOS-159;",";Y
POS-99
1440 FOR K=5 TO 17
1450 SOUND 1,K*K
1460 NEXT K
1470 LOCATE 1,1:PRINT SPC(38)
1480 RETURN
1490 ' **********************
1500 REM      HEADING
```

```
1510 LOCATE 1,1
1520 PRINT "HEADING IS";ANGLE;"DEGREES"
1530 GOSUB 1440
1540 RETURN
1550 ' **********************
1560 REM      SETSCRUNCH
1570 LOCATE 1,22
1580 PRINT "CURRENT RATIO IS ";RATIO
1590 INPUT "ENTER NEW RATIO";RATIO
1600 LOCATE 1,22
1610 PRINT SPC(30):PRINT SPC(30)
1620 RETURN
1630 ' **********************
1640 REM      PEN
1650 Q$="UP"
1660 IF TEMPCOL=99 THEN Q$="DOWN"
1670 LOCATE 1,1:PRINT "PEN IS ";Q$
1680 GOSUB 1440
1690 RETURN
1700 ' **********************
1710 REM      DEFINE PROCEDURE
1720 SOUND 2,40:SOUND 2,90:SOUND 2,40
1730 GOSUB 1060:REM CLEAN
1740 IF PROC<10 THEN PROC=PROC+1
1750 B$(PROC)=MID$(Z$,5,(LEN(Z$)-6)):REM
                        NAME OF PROCEDURE
1760 LOCATE 1,1
1770 PRINT "ENTER DEFINITION OF PROCEDUR
E NO.";:PEN 1:PRINT PROC:PEN 3
1780 PRINT:PRINT "TO ";B$(PROC):PEN 2:PR
INT
1790 N$="":X=1
1800 PRINT "* ";:LINE INPUT M$:M$=UPPER$
(M$)
1810 IF M$="END" THEN SOUND 1,120:GOTO 1
840
1820 N$=N$+M$+" "
1830 GOTO 1800
1840 C$(PROC)=N$
1850 GOSUB 1050:REM CLEAN
1860 RETURN
1870 ' **********************
1880 REM INITIALISE
```

```
1890 MODE 1:INK 0,0:INK 1,26:INK 2,24:IN
K 3,6:BORDER 0:PEN 1
1900 DIM A$(300):REM TO HOLD SPLIT
                        INPUT ELEMENTS
1910 DIM C$(10):REM PROCEDURE MEANING
1920 PROC=0
1930 DEF FN R(X)=X*PI/180
1940 TEMPCOL=99
1950 GOSUB 1010:REM CLEARSCREEN
1960 Z$="DUMMY"
1970 RATIO=1
1980 RETURN
```

Why is it a Turtle?

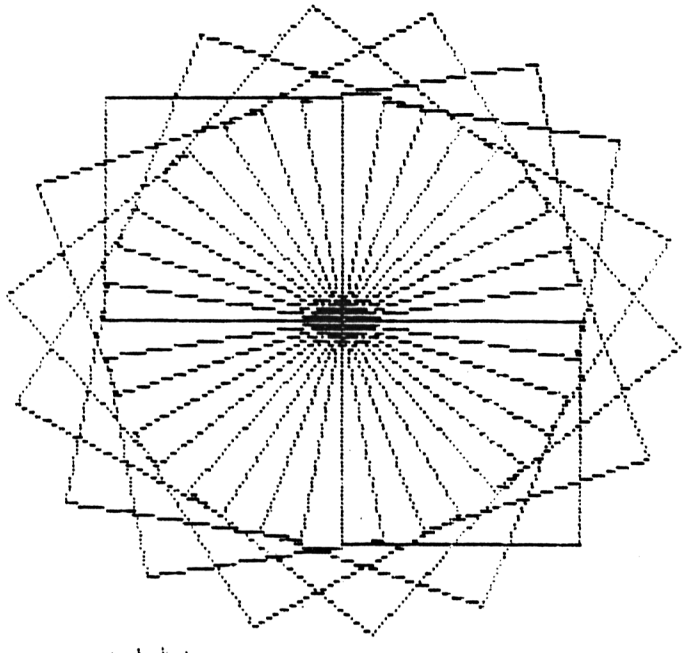
Why call it a turtle? Towards the end of the 1960, many electornic turtles were developed by American researchers. They were generally small boxes or inverted-basin-shaped objects, about the size of a small cereal packet, running on three or four wheels. Usually a cord connected the turtle to a controlling computer. By typing in commands at the computer keyboard, the turtle could be told to perform movements, such as moving forwards or backwards or turning through specified angles.

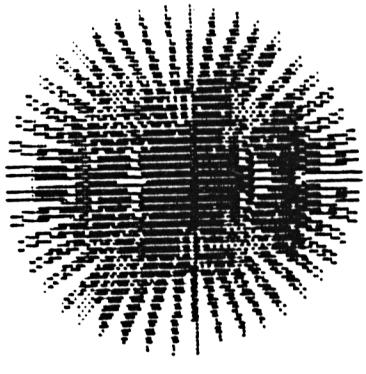
Many of the turtles were furnished with a pen, which faced towards the floor. The turtle could be instructed to actually push the pen against the floor, so it could trace out its movements on large sheets of paper laid down just for this purpose.

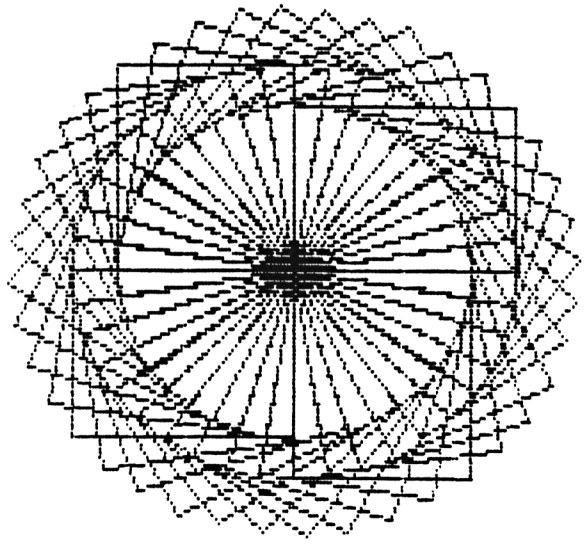
When the evolution of computer graphics and microcomputers made it possible for most of us to afford personal computers which could produce complex pictures, the physical turtle became a 'picture turtle' on the screen.

Many implementations of Logo allow you to see an actual 'turtle' on the screen, as it scurries around, leaving a trail behind it. (In most cases, this turtle is more like the V-shaped spaceship used in arcade games like *Asteroids* than an animal. Sometimes it isn't even as elaborate as that.) *Logo-K* does not have a turtle-shaped object on the screen, although the turtle's position is shown with a dot.

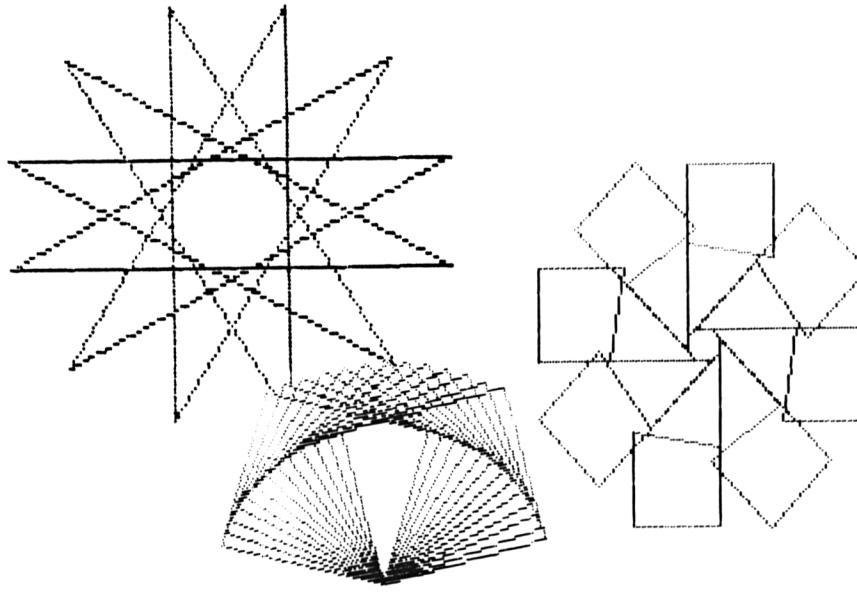
Before we go any further, here's a hint of what lies ahead. This is how marvellous *Logo-K* pictures can be:







You'll be creating much better designs than that by the time you've read through this tutorial. Here are a few others, to convince you (if you need convincing) that it's worth working through the material, step by step:



Now that *Logo-K* is running on your Amstrad, look to the centre of the screen. You should see a small yellow dot there. This is the turtle.

This turtle is under your command. It is waiting patiently for your instructions, and so long as it can understand what you tell it to do, will carry out your wishes right in front of your eyes.

Let's give it our first command. Type in FORWARD 80 (with a space after FORWARD), and then press the ENTER key. (You must always press the ENTER key when you want the turtle to obey a command you have just typed in.) The turtle will move 'forward' (up the screen, in this case) for 80 turtle *steps*. As you can see, the turtle takes pretty tiny steps:

```
FORWARD 80
```

The turtle is always *facing* in one direction of another. When you first run the program, the turtle is always facing upwards. We call this direction zero degrees. When you tell the turtle to move FORWARD, it moves in the direction it is facing.

When you tell it to go BACK, it travels (naturally enough) in the opposite direction. We can move it back 80 tiny turtle steps with the command:

```
BACK 80
```

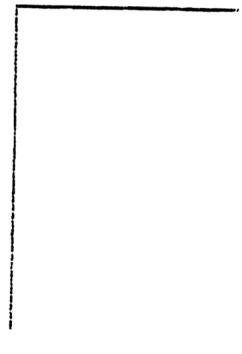

Let's move it forward 80 steps again:

```
FORWARD 80
```

Now, to prove the turtle is back at the top of the line, we will get it to turn a corner, to the right.

Imagine that you are standing up. How many derees do you turn through when you turn to the right? Yes, 90 degrees; the turtle does the same. If you enter the command RIGHT 90, the turtle will turn through 90 degrees. However, a RIGHT command does not make the turtle *move.* It needs FORWARD or BACK to do this. If you enter RIGHT 90 followed by a space (with a space between RIGHT and 90) followed by FORWARD 50 (with a space between FORWARD and 50), you'll see this on your screen:

```
RIGHT 90 FORWARD 50
```



As you can see from this, it is possible to have more than a single command as part of one input. In this input line, we had RIGHT 90 FORWARD 50. When you pressed the ENTER key, the turtle had a look at the line to make sure it understood it, then turned RIGHT through 90 degrees, before moving FORWARD for 50 of its tiny turtle steps.

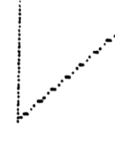
Our turtle is pretty flexible. It can move FORWARD or BACK, and can turn to the RIGHT through the number of degrees you specify. Remember, the number of degrees it turns through is measured from the angle it is currently facing. It is now facing to the right. Tell it to turn RIGHT 90 again, and the turtle would be facing straight down the screen.

Stop the program, by pressing the ENTER key *without* putting any command in. "OK, BYE FOR NOW . . ." appears as the goodbye message. (You can also get this message by entering BYE as your command, if you feel that pressing the ENTER key is a little too unfriendly towards your turtle.)

Run it agin, and enter this command sequence:

```
BACK 30 RIGHT 45 FORWARD 30
```

You'll see something like this on the screen:



As you've probably guessed, the turtle can turn LEFT as well as RIGHT. Try this set of commands:

```
BACK 40 RIGHT 45 FORWARD 20
LEFT 90 FORWARD 20
```

They produce a picture like this:

The Four Basic Commands

You have already learned the four most important commands in *Logo-K!* That's pretty good going, considering you've probably only been at it for a few minutes.

Let's review what you now know:

FORWARD - this moves the turtle forward. It must be followed by a space, then a number, before pressing ENTER. Don't enter too high a number, or the turtle will try to walk off the screen, and give rise to an error message. (Note that if you do try to 'walk off the screen', and this stops the program, you can always restart it, without losing any procedures you've defined — we learn about those later — by entering GOTO 40.)

BACK — This is the opposite of FORWARD, moving the turtle in the opposite direction to the one in which it is currently facing.

RIGHT — Turns the turtle to the right the number of degrees specified. The turtle begins each run of the program facing 'zero degrees', which is straight up the screen.

LEFT — Turns the turtle to the left the number of degrees specified.

Keeping it Short and Sweet

Now it can get a bit of a bore continually typing in the whole of a word like FORWARD, so *Logo-K* (in common with all other Logo implementations) accepts abbreviations for the four words you now know (and for several others which you will meet in due course).

Here are the shortened forms:

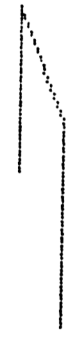
Long

Long	Short
FORWARD	FD
BACK	BK
RIGHT	RT
LEFT	LT

You can try out these variations with a set of commands like this:

```
FD 40 RT 160 FD 30 RT 20 FD 50
```

This produces this result:



Try some of your own now, combining the commands in any way you choose.

Using the Pen

So far, we've run our *Logo-K* with the turtle leaving a trail wherever it goes. That is, it had its 'pen' down on the floor as it trundled around. There is no need for the pen to be down on the floor, writing, all the time the turtle is running.

The command PENUP (one word, without a space) will 'lift the pen'. The turtle will continue to move, but will not leave a trail as it goes. PENDOWN, naturally enough, puts the pen back on the ground.

Here's a string of commands which uses PENUP and PENDOWN:

```
FD 30 RT 90 PENUP FD 30 RT 90
PENDOWN FD 30
```

And this is what it produces:

Like the first four words we met, both PENUP and PENDOWN can be abbreviated. PENUP can be entered into the computer as PU and PENDOWN can be entered as PD, as you can see here:

```
FD 30 RT 90 PU FD 30 RT 90 PD FD 30
```

Try this, and see if it produces the same result as using the words in full. (Start a *new run* before you try each of these, by going through the 'OK, BYE FOR NOW . . .' routine, so you have a clean screen to work with.)

Going Home

The turtle starts each run of *Logo-K* sitting slap bang in the middle of the screen, facing upwards. This is called the 'home position'. At any time, you can send your turtle back home, just by typing in the command HOME, like this:

```
FD 40 LT 90 FD 10 HOME LT 90 FD 20
```

This little set of commands produces this result:

Add a few more commands, and the turtle continues where it left off last. Try these:

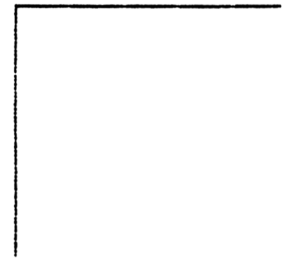
```
HOME RT 135 FD 10 PU FD 5 PD FD
10 RT 90 FD 10 PU FD 5 PD
```

Can you see how we've made the turtle draw the line with a few bits missing, near the bottom right of the picture?

Turning About

Now, if you think about it, you'll realise that turning to your left one quarter of a turn leaves you in the same position as you would be if you turned through three quarters of a circle to the right. This of that statement in 'turtle terms'. LEFT 90 is the same as RIGHT 270. If you don't believe me, try these two command entries:

```
HOME FD 60 RT 90 FD 60

CS FD 60 LT 270 FD 60
```



What was that CS at the start of the last command? It stands for CLEARSCREEN (CS or CLEARSCREEN can be used). Whereas HOME simply puts the turtle back in the centre of the screen, facing upwards, without clearing any lines from the screen, CLEARSCREEN wipes all the lines of the screen, then goes HOME. You'll find it far more convenient to use CLEARSCREEN or CS than it is to end the program with the "OK, BYE FOR NOW . . ." message, and then RUN it again.

So, the commands you now know are:

FORWARD	**FD**
BACK	**BK**
RIGHT	**RT**
LEFT	**LT**
HOME	**(no shortened form!)**
CLEARSCREEN	**CS**

You certainly are progressing well.

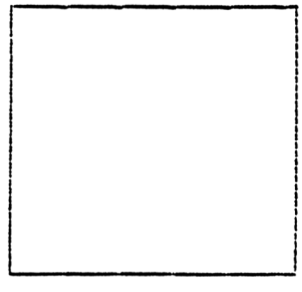
Shaping Up

It's all very well drawing these odd lines and things, but isn't it about time we produced some *real* pictures and shapes? Indeed it is.

We'll start with a square. Enter these commands:

```
FORWARD 65 RIGHT 90 FORWARD 65
RIGHT 90 FORWARD 65 RIGHT 90
FORWARD 65 RIGHT 90
```

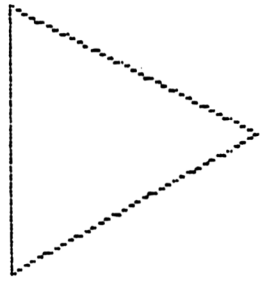
They produce this shape:



For a triangle, try these:

```
CS RT 120 FD 65 RT 120 FD 65 RT
120 FD 65
```

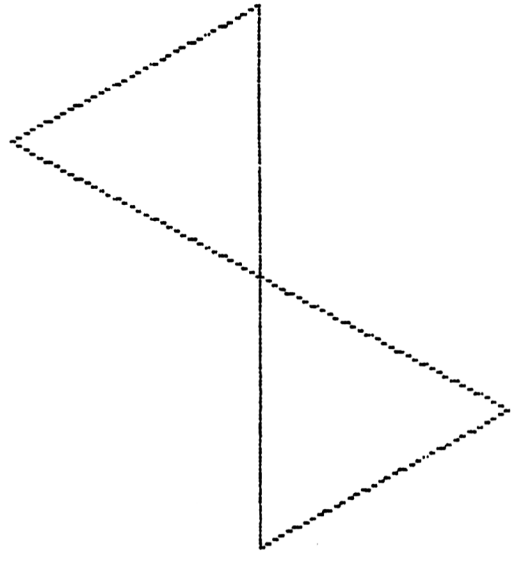
This produces the following pattern:



Spin around a bit more, and do the triangle again, like this:

```
RT 300 FD 65 RT 120 FD 65 RT
120 FD 65
```

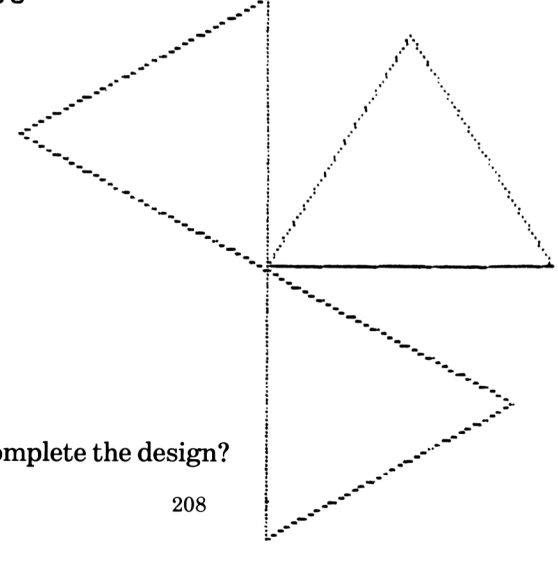
Produces this:



But why stop there? Add the following:

```
RT 220 FD 65 RT 120 FD 65 RT
120 FD 65
```

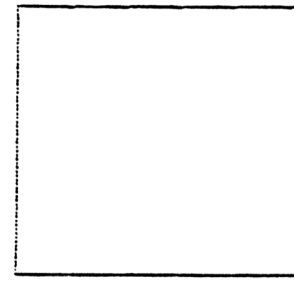
To get this design:



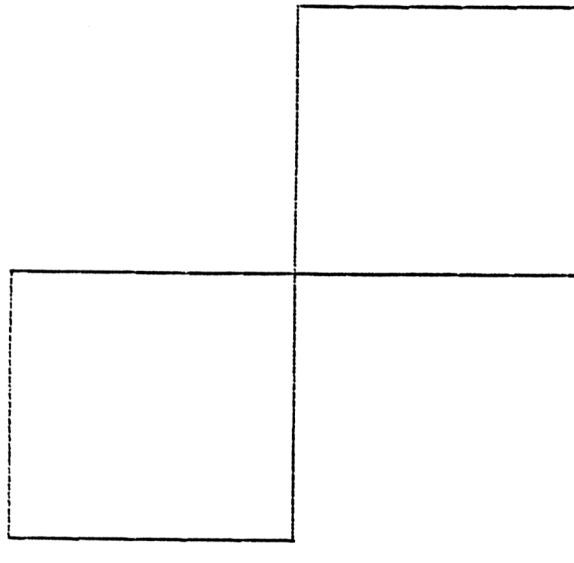
Now, how would you complete the design?

We can create some quite reasonable designs based on squares, as this sequence shows (in each case the command line used is printed, followed by the effect of it on the screen):

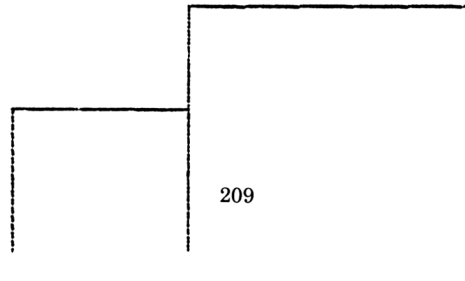
```
FD 65 RT 90 FD 65 RT 90 FD 65 RT
 90 FD 65 RT 90
```



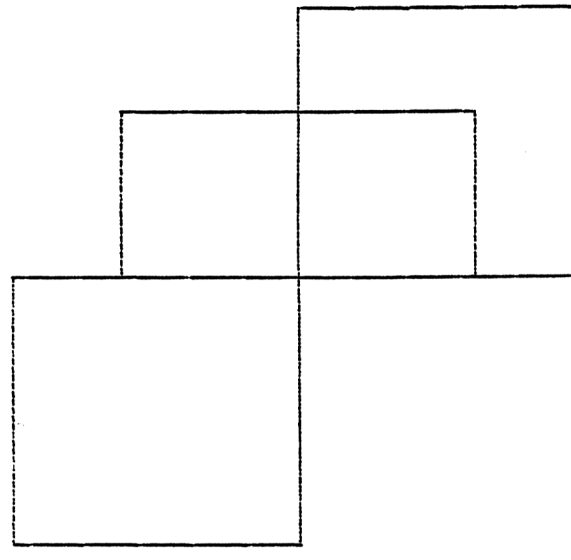
```
RT 180 FD 65 RT 90 FD 65 RT 90 FD
  65 RT 90 FD 65 RT 90
```



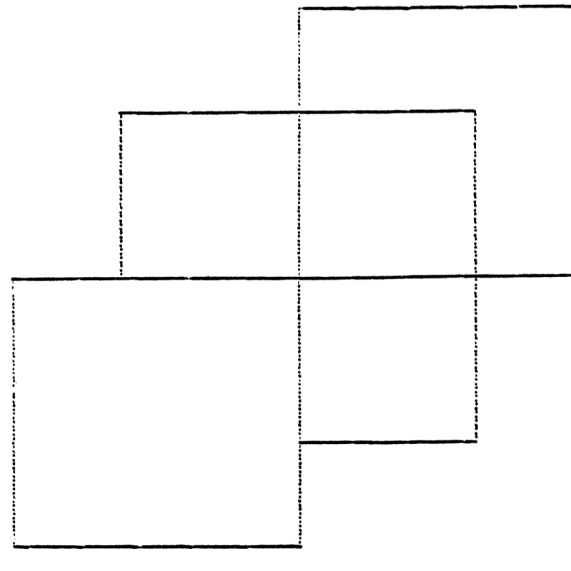
```
RT 90 FD 40 RT 90 FD 40 RT 90 FD
 40 RT 90 FD 40 RT 90
```



```
RT 90 FD 40 RT 90 FD 40 RT 90 FD
 40 RT 90 FD 40 RT 90
```



```
RT 90 FD 40 RT 90 FD 40 RT 90 FD
 40 RT 90 FD 40 RT 90
```



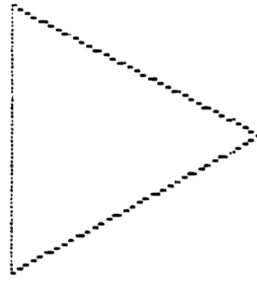
You should experiment on extending this picture.

Fundamental Shapes

Let's look again at a few standard shapes, and see if there is a pattern emerging which can help us when we want to create more difficult shapes.

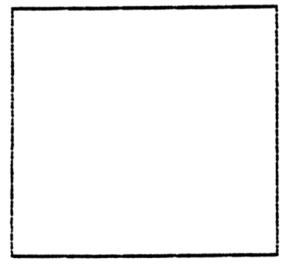
First, a triangle (and, again, the line entered into the computer is shown, followed by the on-screen result):

```
FD 65 RT 120 FD 65 RT 120 FD 65
RT 120
```



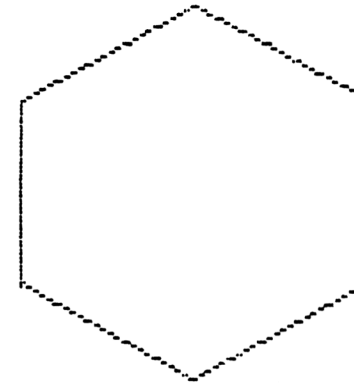
Now, a square:

```
CS FD 60 RT 90 FD 60 RT 90 FD 60
 RT 90 FD 60 RT 90
```



And now a six-sided figure (What's it called? A sexygon?):

```
CS FD 45 RT 60 FD 45 RT 60 FD 45
 RT 60 FD 45 RT 60 FD 45 RT 60
FD 45 RT 60
```



Look again at the triangle commands. We entered three direction commands, RT 120, RT 120 and RT 120. For the square we entered four direction commands: RT 90, RT 90, RT 90 and RT 90. The 'Sexygon' demanded six direction commands: RT 60, RT 60, RT 60, RT 60, RT 60 and RT 60.

Can you see an underlying rule that lies in these (apart from there being, in each case, the same number of direction commands as there are sides)? If you add up the direction commands in each case, they come to 360 (120 plus 120 plus 120; or 90 plus 90 plus 90 plus 90; or 60 plus 60 plus 60 plus 60 plus 60 plus 60). This is called the *Total Turtle Trip Theorem!* If the turtle is to get back to its starting place, it must turn through 360 degrees. This is very useful to know.

Repeating Yourself

It has probably occurred to you that it wastes a lot of time typing in the same set of commands several times to get just one figure. If you look back at the commands for a triangle, you'll see you entered FD 65 and RT 120 three times each. Surely there must be a simpler way.

And, of course, there is.

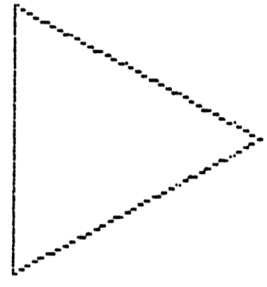
Logo-K comes complete with the command REPEAT. You follow it with a number representing the number of times you want the material to be repeated, and the turtle does the rest. A REPEAT line looks like this:

```
REPEAT 3[FD 65 RT 120]
```

First there is the word REPEAT followed by the number 3. This is the number of times we want the turtle to repeat the instructions which are

enclosed in the square brackets. Do not have anything else in a line with a REPEAT.

Here's what happens when you run it:

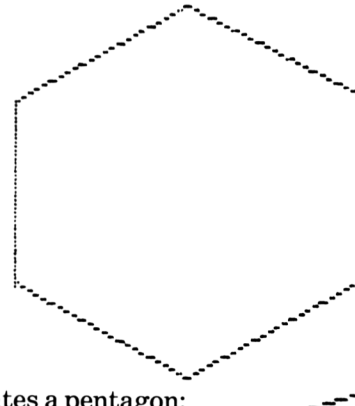


Now, enter CS (to clear the screen, remember) and then run the next one (try to guess what it will be, before you enter it):

```
REPEAT 4[FD 65 RT 90]
```

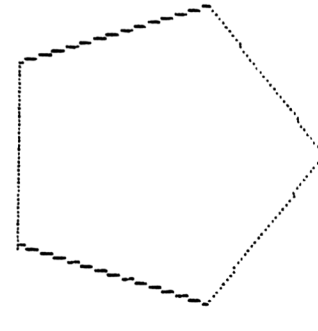
Here's our old friend, the sexygon (with the set of commands, followed by the result of entering them):

```
REPEAT 6[FD 45 RT 60]
```



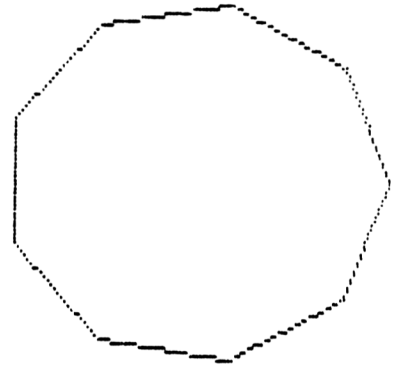
Here the turtle obediently creates a pentagon:

```
REPEAT 5[FD 45 RT 72]
```



Let's try a nine-sided figure:

```
REPEAT 9[FD 30 RT 40]
```



What happens when we make the angle changes even smaller, like this next one, when we have a thirty-sided figure:

```
REPEAT 30[FD 2 RT 12]
```

It has practically turned into a circle. The turtle, of course, can turn left as well as right, so let's make another 'circle', with steps of 3, rather than 2 as in the previous example:

```
REPEAT 30[FD 3 LT 12]
```

We move the turtle through 90 degrees, and add a third circle:

```
RT 90
REPEAT 30[FD 4 LT 12]
```

By all means continue to work on this, adding as many circles as you like.

Defining Procedures

The only problem with typing in the instructions for, say, a square, is that once they are executed, the turtle forgets them. If we want it to draw a square again, we have to teach it about squares, all over again.

Well, we don't. The words which the turtle understands when you first run *Logo-K* are called *primitives.* Words which you invent, as we will be doing in a moment, are called *procedures. Logo-K* can learn ten procedures each time you run it, and any one of these procedures can be used within subsequent ones. That slightly mystifying statement should become clear shortly.

To define a procedure, you use the word TO followed by the name of the procedure. You can call it anything you like (up to 10 letters), so long as it is not the same of a primitive (so you can't invent a procedure called FORWARD or RIGHT); it does not *start* with the name of a primitive (such as RIGHTONE or FORWARDHO); and does not start with, or is the same as, a procedure you have already defined. This is not anywhere as bad as it sounds, and you'll have little or no trouble following this relatively simple rule. (Other Logos have slightly less restrictions than *Logo-K* when it comes to naming procedures, but you'll find our rules useful for keeping everything clear, no matter which Logo you use.)

Let's say we want to define a procedure called SQUARE, so that any time we want the turtle to draw a square on the screen, we can just enter the word SQUARE, rather than all that RT 90 rubbish.

Here's how we go about it.

First you enter TO SQUARE, and the screen will clear, showing you this:

```
ENTER DEFINITION OF
PROCEDURE NUMBER 1

TO SQUARE
```

To input prompt, down in the bottom left hand corner, has been a 'greater than' sign to date. Now it changes to an asterisk, to remind you that you are in the mod which defines procedures, rather than the one in which the Amstrad turtle follows your instructions instantly.

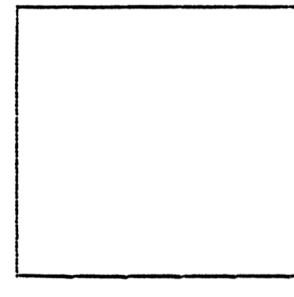
You type in your definition, using ENTER from time to time, to keep it clear what it is in your mind in the procedure. Note that you *cannot* use the word REPEAT within a definition. Instead, as you can see here, to define a square you need to spell out the RT 90 FD 65 four times:

```
RT 90 FD 65
RT 90 FD 65
RT 90 FD 65
RT 90 FD 65

END
```

To tell the turtle you have finished with the procedure definition, you type in END.

Let's see if our turtle has learned how 'to square'. All you have to do is type in SQUARE (just as though it was a primitive like FORWARD or RIGHT) and see what happens:

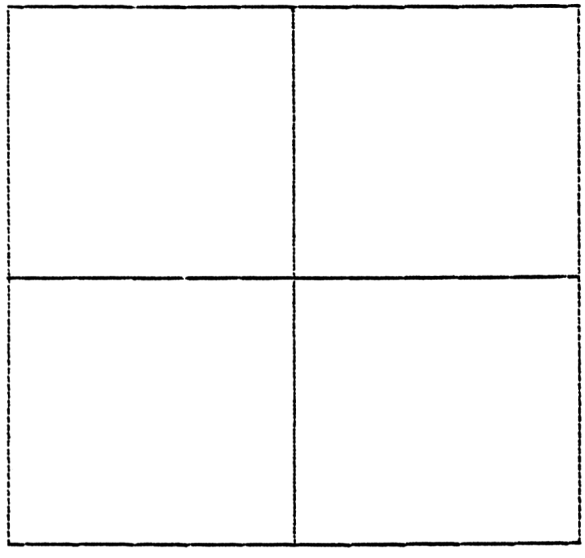
```
SQUARE
```



It works! Our turtle now knows what SQUARE means. SQUARE has now, in effect, joined the list of **primitives**. Let's try SQUARE within a REPEAT loop.

Here's what happens:

```
CS
REPEAT 4[RT 90 SQUARE]
```

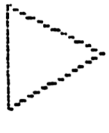


We will now teach our turtle a new word, TRIANGLE. Here we go. You should know how to do it. Enter TO TRIANGLE and the screen will change to the procedure-entry one. You enter the definition, ending with END, so the screen looks like this:

```
ENTER DEFINITION OF
PROCEDURE NUMBER 2

TO TRIANGLE

FD 25 RT 120
FD 25 RT 120
FD 25 RT 120

END
```

Does our turtle know what a triangle is?

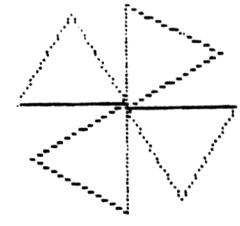
```
TRIANGLE
```



Of course it does, clever little beast.

We can use TRIANGLE within direct commands:

```
REPEAT 4[TRIANGLE RT 90]
```



More Fun and Games

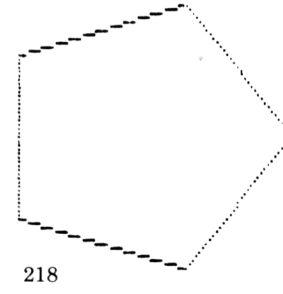
Logo-K starts to slow down somewhat as it learns more procedures, so let's give it a fresh beginning by entering BYE, and once it has stopped, start it again with RUN and ENTER.

Firstly, teach it the procedure SHAPE:

```
ENTER DEFINITION OF
PROCEDURE NUMBER 1

TO SHAPE

FD 40 RT 72
FD 40 RT 72
FD 40 RT 72
FD 40 RT 72
FD 40 RT 72

END
```

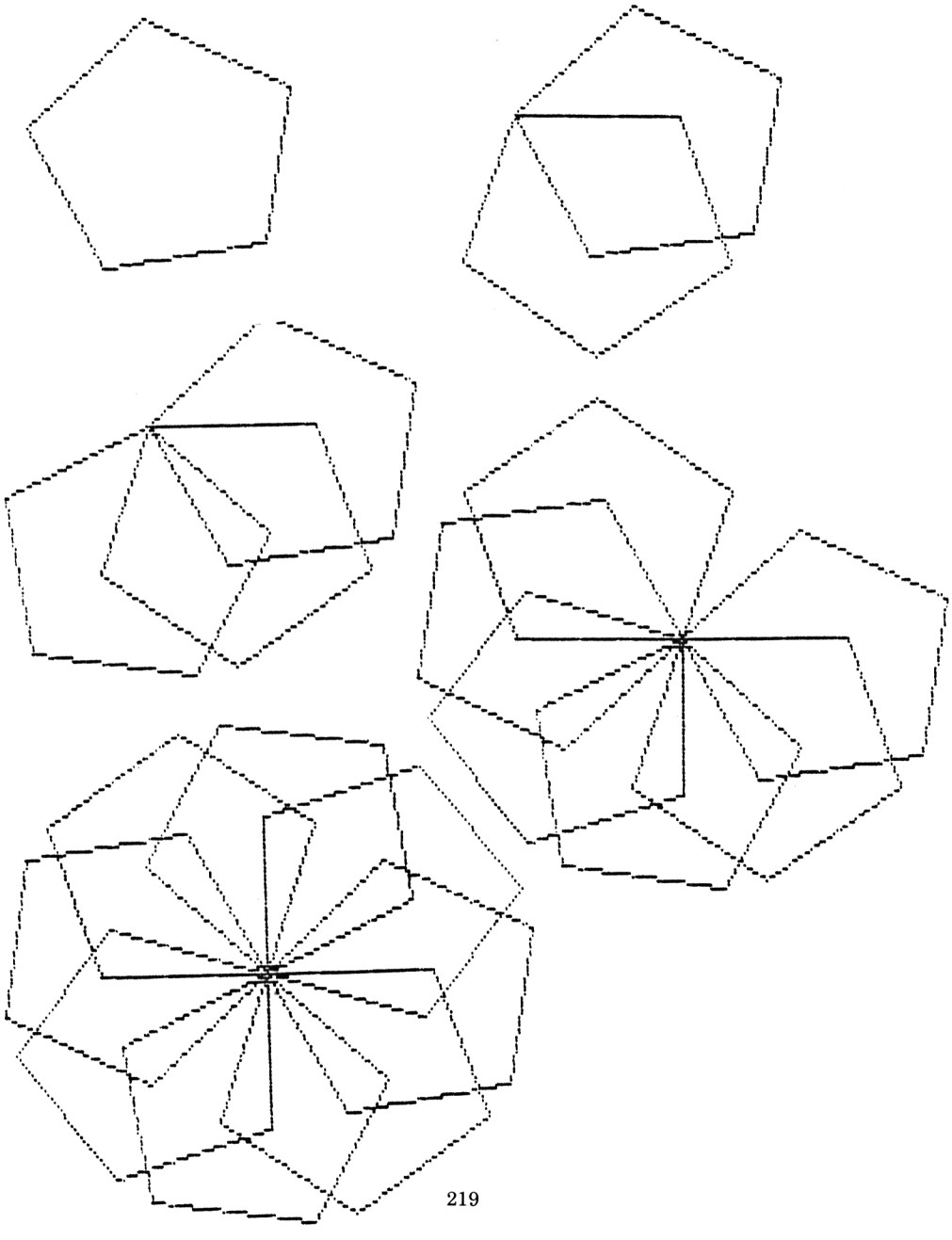
This is what you see when SHAPE is executed:

```
SHAPE
```



Here is the evolution of a picture created from rotating the SHAPE:

```
CS
REPEAT 8[RT 45 SHAPE]
```



Moving About

To date, we have been starting all our new drawings from the centre of the screen. You know that both HOME and CLEARSCREEN move the turtle back to the middle of the picture. The co-ordinates of the centre are given as 0,0 (zero, zero). The command SETPOS (for SET POSition) is used to move the turtle to any position on the screen you choose. You can move 126 places to the right (0 to 126) or to the left (0 to — 126) or 86 steps up (0 to 86) or down (0 -86).

The computer leaves a line as it moves. How can we get rid of that line in leaves, if we don't want it? You know that CLEARSCREEN cleans the screen, but also moves the turtle back to HOME. The command CLEAN, however, wipes all lines off the screen, but leaves the turtle where it was before the screen was CLEANed.

We'll move the turtle 50 steps to the left of HOME and 50 steps down, with this line:

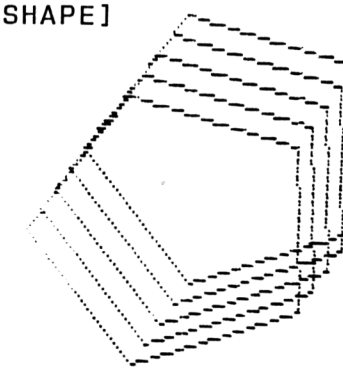
```
SETPOS -50 -50 CLEAN
```

The turtle always faces zero degrees after HOME. If you do not go HOME, it stays facing the angle it was facing at the end of the previous set of commands. The primitive, SETHEADING, is used to turn the turtle towards whatever direction you choose. The direction the turtle ends up after RIGHT or LEFT is *relative* to the previous direction it is facing. In contrast to this, SETHEADING is *absolute.* SETHEADING 90 points the turtle 90 degrees to the right, regardless of the previous setting.

We have put the turtle a little down towards the bottom left hand corner with our SETPOS command. Now we use SETHEADING to set the turtle at 34 degrees, and give it a SHAPE to work with (along with PENUP and PENDOWN). Start by typing in SETHEADING 34, then press the ENTER key. Now, type this in:

```
REPEAT 5[PU FD 6 PD SHAPE]
```

Here is the drawing:

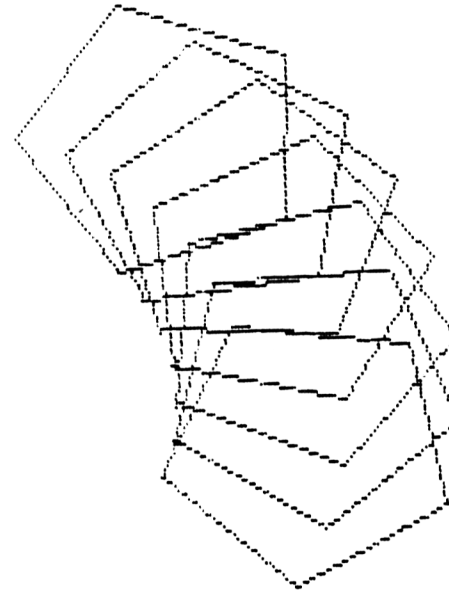


Seven repeats can be interesting. First, type this in, and press ENTER:

```
SETHEADING 35 SETPOS -50 -50 CLEAN
```

Now, run this:

```
REPEAT 7[LT 10 PU FD 8 PD SHAPE]
```



Start a new run now, and define the following procedure, called STEVE:

```
TO STEVE

RT 72 FD 120
RT 72 FD 120
RT 72 FD 120
RT 72 FD 120
RT 72 FD 120
```

Use it as follows:

```
REPEAT 99[FD 3 RT 7 STEVE]
```

Other Commands

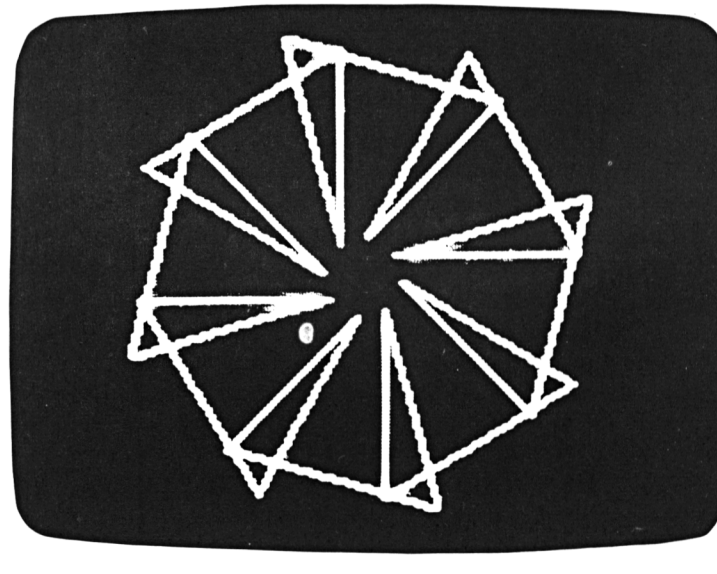
Believe it or not, you now know most of the important primitives for controlling Logo graphics. In this section, we'll have a look at a few additional useful commands.

SETX and SETY

Whereas you use SETPOS to set both the across (X) and down (Y) co-ordinates, you use SETX and SETY if you wish to change just one of them. Inputs are from -126 to 126 for SETX and from -86 to 86 for SETY, and are used as SETX 45 or SETY -44.

POS

When you enter this, the computer will tell you where the turtle currently is on the screen. Just enter POS, and the information you want will appear on the top line of the screen.



HEADING

Just as POS tells you *where* the turtle is, HEADING will tell you which angle is it facing. Just enter HEADING, and the information will appear on the top line of the screen.

SETSCRUNCH

This is an odd-sounding one. It is used to modify the ratio between the horizontal and vertical axes on the screen, so you can get your squares looking like long oblongs if you like. Just type in SETSCRUNCH, and the Amstrad will tell you the current ratio setting, and ask you for the new one. Experiment with this a little to get the best results.

PEN

This primitive will tell you whether the PEN is up or down, and what colour it is. The information appears at the top of the screen, and all you need to do is enter PEN to get it.

Note: Most Logos respond with a message like I DON'T KNOW HOW TO TANGO, if TANGO is a command you enter which it does not know. In the majority of cases, *Logo-K* will simply skip a command it does not recognise, and continuing executing your input string. This is generally much better than the computer grinding to a halt. However, if you enter a primitive which expects a following number (such as RIGHT) *without* the number, the computer will stop. You can start the program running again, without losing your procedures, by entering GOTO 40, and once the program is running, HOME. The program skips any extra spaces, so put as many as you like in a line.

Additional Primitives

There are many, many Logo primitives. The graphic controllers are just part of the complete set. With *Logo-K,* you've become acquainted with the most important of these graphics primitives.

You know enough, now, to work with any Logo availble on any computer in the world. When I started to write *Logo-K* for the Amstrad, I decided I would stick — whenever possible — to the Logo Computer Systems, Inc. primitive names and functions. Although the major Logo available for the BBC Micro uses different (and perhaps better) words for certain primitives (such as XPOS and YPOS for XCOR and YCOR), and other Logos such as those available for the RML 380Z and TI 99/4A vary quite significantly from the 'norm' (defined, for this exercise, to be LCSI's standard), there seemed very little reason for modifying *Logo-K* even if it would make it 'easier' than an LCSI Logo.

The real values of *Logo-K* would be (a) that it was the cheapest

implementation of Logo turtle graphics for the Amstrad, and (b), that it would not force those who acquired initial Logo knowledge working with *Logo-K* to either stick with it, or laboriously relearn it when they came across another implementation of the language. Therefore, the changes have been kept to an absolute minimum.

Graphics primitives which are not supported by *Logo-K* include these:

SHOWTURTLE or ST — to get the actual 'turtle' shape to appear on the screen.

HIDETURTLE or HT — fairly obviously, this is to 'hide' the turtle, or make it invisible.

FENCE, WRAP, WINDOW — these primitives control the way the turtle behaves when it reaches the edge of the screen. *Logo-K* insists that the turtle remains on the screen. With other Logos, FENCE stops the turtle roaming beyond the edge of the screen; WRAP ensures that if the turtle sanders off one side (or the top) of the screen, it reappears on the other side (or the bottom); and WINDOW allows the turtle to move off the edge of the screen (although, of course, its tracks cannot be shown), treating the visible screen as a 'window' on a much larger area over which the turtle moves.

PENERASE — This effectively sets the PEN colour to be the same as the BACKGROUND colour, so that the turtle can erase lines by retracing them.

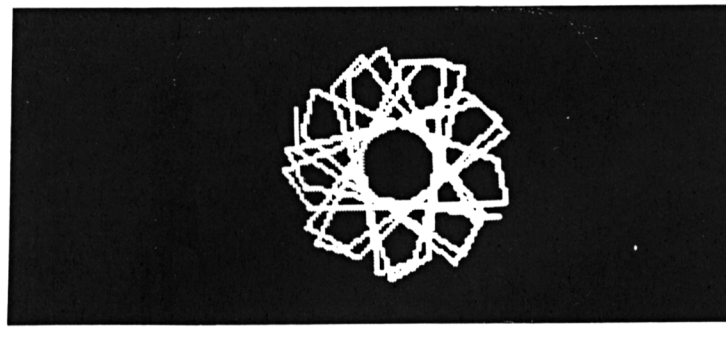
SETPEN — This is supplied with some Logos, and is used to set the condition of the pen (up/down) and its colour.

TOWARDS — This returns an angle (to be used with SET-HEADING) which would turn the turtle towards a particular set of co-ordinates.

Variables

Logo-K does not support variables. Variables are distinguished in Logo by the presence of a colon (:) in front of the word which is the variable name, such as :ANGLE.

Logo variables are generally used in procedure definitions, and are given as part of the procedure call. For example, a procedure SHAPE could be defined as REPEAT :MANY(RT :ANGLE FD : DISTANCE). When called, it would be entered as, say SHAPE 6 8 7. You have to be very careful, when using a Logo which supports variables, to give the numbers in the same order as they appear within the procedure definition.



Uncharted Lands

The brings us to the end of the 'learning' process with *Logo-K.* It's time now to have some real fun, experimenting with *Logo-K* and creating your own designs.

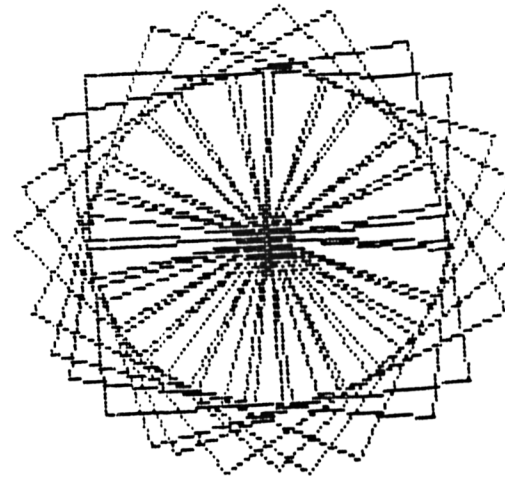
Here are a few I cooked up over a hot Amstrad while I was writing the Logo section of the *Omnibus.*

The first one needs a procedure called DIAMOND to be defined before it can be run. Here's the DIAMOND definition:

```
TO DIAMOND

RT 45
RT 90 FD 90
RT 90 FD 90
RT 90 FD 90
RT 90 FD 90

END
```

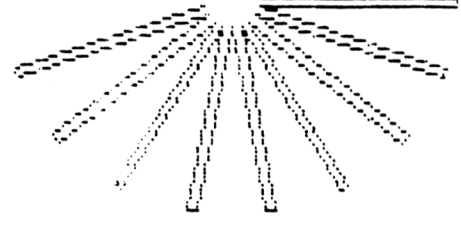
And here's one of many things you can do with it:

```
REPEAT 26[DIAMOND LT 14]
```



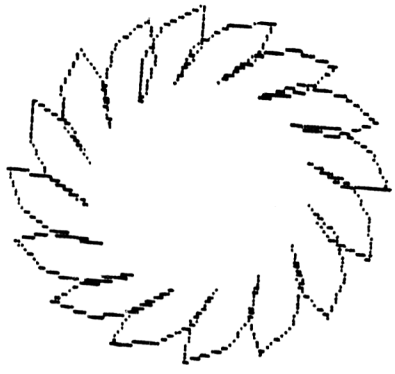
Looking a little like half of the BBC's breakfast television symbol, this design would probably be even better if the number of repeats was, say, doubled:

```
REPEAT 9[RT 90 FD 45 RT 90 FD 2
RT 90 FD 45 RT 110]
```



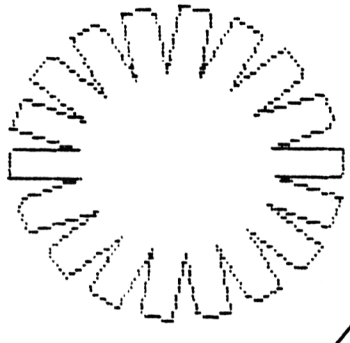
Sawtooth designs are the order of the day with this one:

```
REPEAT 16[FD 20 RT 30 FD 20 RT 30
FD 20 RT 120 FD 20 RT 30 FD 20 RT
173]
```



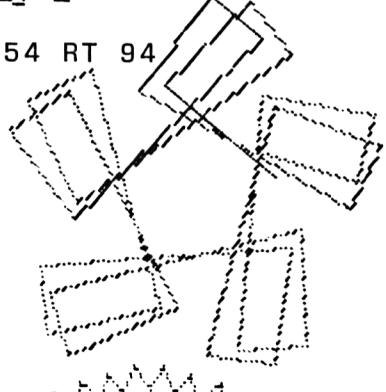
Flatten them off a little, and we get this:

```
REPEAT 18[RT 90 FD 32 RT 90 FD 12
RT 45 FD 4 RT 45 FD 28 RT 110]
```



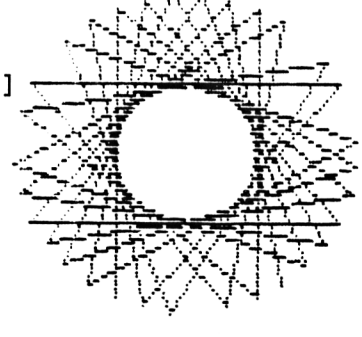
The flags are flyin', for me and my Logo:

```
REPEAT 10[FD 70 RT 97 FD 54 RT 94
FD 32 RT 95 FD 48]
```



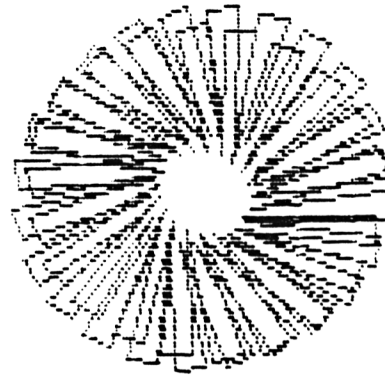
String-art rears its delightful head:

```
SETPOS -20 -20 CLEAN
REPEAT 30[FD 140 RT 130]
```



Jitterbug shutterbugs make this design memorable:

```
REPEAT 45[RT 90 FD 60 RT 90 FD 14
RT 98 FD 64 RT 98]
```

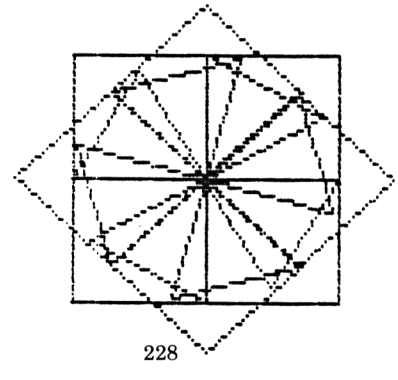


Start a new run, and define SHAPE:

```
ENTER DEFINITION OF
PROCEDURE NUMBER 1

TO SHAPE
FD 60 RT 90 FD 60 RT 90
FD 60 RT 90 FD 60 RT 90
FD 60 RT 120
FD 60 RT 120
FD 60 RT 120
END
```

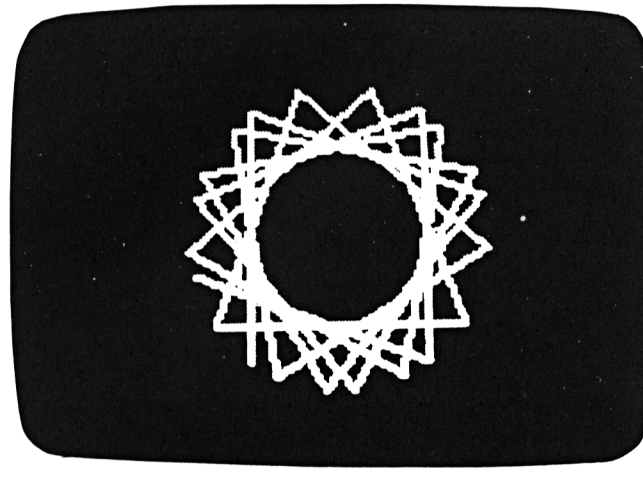
Twirl it about for this:

```
REPEAT 8[SHAPE LT 45]
```



It's rather delightful watching some designs unfold:

```
SETPOS 0 - 120
REPEAT 45[FD 240 RT 184]

REPEAT 40[RT 9 FD 80 RT 90 FD 100
RT 90 FD 100 RT 90 FD 100 RT 90]

REPEAT 60[RT 3.08 FD 98 RT 90 FD 10
RT 90 FD 98]

REPEAT 24[FD 94 LT 184 FD 80
RT 15]

REPEAT 8[RT 45 FD 140 RT 91 FD 18
RT 99 FD 136]
```



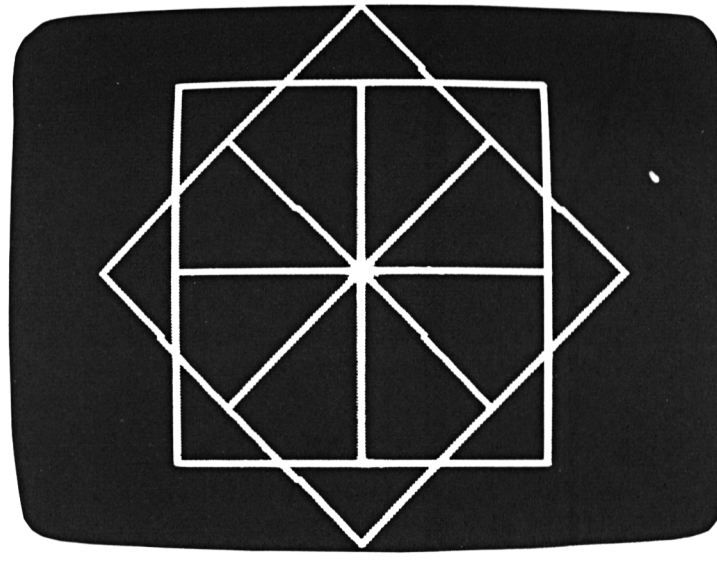
```
REPEAT 24[RT 49 FD 120 RT 45 FD 26]

REPEAT 17[RT 90 FD 60 RT 90
FD 60 RT 90 FD 60 RT 111]

REPEAT 36[FD 125 RT 90 FD 100 RT 90
FD 81 RT 5]

REPEAT 24[FD 180 LT 190 FD 168
RT 80]
```

These designs are just a sample of what you can do with your Amstrad and *Logo-K* (plus a little patience). Enjoy it.

Further Reading

There are many good books on Logo. Almost any book currently available on the subject will give you additional projects to try with your *Logo-K*. The best I have come across are these:

1, 2, 3 MY COMPUTER AND ME — Donna Bearden (Reston Publishing Company, Inc., Reston, Virginia, USA, 1983)
IBM PC AND PC JR LOGO PROGRAMMING PRIMER — Don Martin, Marijane Paulsen and Stephen Prata (Howard W. Sams & Co., Inc., Indianapolis, Indiana, 1984)
LEARNING LOGO ON THE APPLE II — Anne McDougall, Tony Adams and Pauline Adams (Prentice-Hall of Australia Pty Ltd, 1982)



LOGO PROGRAMMING — Anne Moller (Century Communications Ltd, London, 1984)
LOGO, A LANGUAGE FOR LEARNING — Anne Sparrow (Pan Books, London and Sydney, 1984)
APPLE LOGO — David D Thornburg (Addison-Wesley Publishing Company, London, 1983)

Section Five
Entertainment

You're sure to enjoy playing games with your Amstrad, whether or not this is the main use of your computer. In this section, we have five different games, which make use of the Amstrad's 'silicon intellect' in convincing ways.

Dominoes pits you against the computer in this well-known game. The rather oddly-named game *Oliver Rand* will have you gritting your teeth in frustration as you attempt the fruitless task of beating the machine in a 'connect the dots' game. *Celestia* fills the Amstrad monitor's screen with constantly-evolving designs, which develop in line with the ideas developed by John Conway in his famous game of *Life.*

The popular board game *CPC-Thello* is next, and again you'll have a difficult — although highly enjoyable — time trying to defeat your computer on this one. Finally, in this section, a strange little moving graphics game comes your way in *Squisher and Squasher* in which you try to eat as many dots as possible on a maze, while avoiding the two malevolent creatures which seem hell-bent on your destruction.

Dominoes

Our present-day dominoes were probably developed in Italy around 200 years ago. They are based on Chinese dominoes, which were in use many centuries before the Italians got hold of them. Domino sets consist of 28 oblong 'stones' or 'tiles'. Each tile is divided into two sections, and 'pips' (like the dots on dice) are placed on each section. The pips range from none (blank) to six, and if the blanks are excluded, represent all the possible throws of a pair of dice.

Our Amstrad version of the game uses *rominoes,* which are close cousins of the tiles used in the non-computer world. Each *romino* is a pair of numbers, such as "54" or "30". These are read as if they were the domino tile "five four" and "three blank". The double-blank domino is represented, naturally enough, as "OO".

In this program, you'll find your Amstrad plays a very good game of the most common form of dominoes, the 'block game'. The dominoes are placed on the table, face downwards, and each player selects seven tiles for his or her own use. The others are left on the table as the reserve, or 'bone yard' as it is often called.

The Amstrad has the first move (and it is simple to modify the program if you want to make the first move), by printing its choice of domino on the screen, like this:

```
AFTER MY MOVE, THIS IS THE BOARD:


00
```

As you can see, the computer has played the double-blank. You must respond by placing a *romino* next to the one placed by the computer, with a number which corresponds to one of the numbers on the *romino* which is already in place. For example, if the computer had played "three four", you could play "two three" or "three six" (or any combination which included a three) to the left of the computer's piece, or any piece which included a four (such as "four six" or "blank four") to the right of the computer's piece.

Once the Amstrad has played, you are shown your hand:

```
HERE IS YOUR HAND:

 1 - 53
 2 - 33
 3 - 65
 4 - 66
 5 - 21
 6 - 20
 7 - 61

ENTER THE NUMBER OF THE PIECE YOU WANT
TO MOVE, 0 TO PASS
          ? 6
AT THE START (S) OR END (E)?
          ? E
```

As you can see, you are asked to select the *number* of the piece you wish to

place down. In this case, I chose piece six, which is "20" (two blank). As the Amstrad had played a double blank, I could put this piece on either side of the computer's tile, but I decide to place it at the end.

```
AFTER YOUR MOVE, THIS IS THE BOARD:


00:02

BEFORE THIS MOVE I HAVE 6 TILES...
```

Notice that the program automatically 'flips' your piece around, so that the two blanks are together. The computer responds, and so the game continues:

```
AFTER MY MOVE, THIS IS THE BOARD:


00:02:22


HERE IS YOUR HAND:

 1 - 53
 2 - 33
 3 - 65
 4 - 66
 5 - 21
 7 - 61

ENTER THE NUMBER OF THE PIECE YOU WANT
TO MOVE, 0 TO PASS
            ? 5
AT THE START (S) OR END (E)?
            ? E


AFTER MY MOVE, THIS IS THE BOARD:


00:02:22:21:11
```

Later in the same game:

```
AFTER MY MOVE, THIS IS THE BOARD:


46:60:00:02:22:21:11:16:66:65:55


HERE IS YOUR HAND:

 1 - 53
 2 - 33

ENTER THE NUMBER OF THE PIECE YOU WANT
TO MOVE, 0 TO PASS
            ? 1
AT THE START (S) OR END (E)?
            ? E
```

The game ends when either one of the players gets rid of all of his or her pieces, or when neither player can move (and note that you enter zero (0) when you cannot move, and must 'pass'):

```
THIS IS THE BOARD:


46:60:00:02:22:21:11:16:66:65:55:53:31

MY TILES:
::::::::::::::

YOUR TILES:
52:33::::::::::::


AT THE END OF THAT GAME, YOUR SCORE
     IS 13 AND MINE IS 0

       SO I'M THE WINNER


 1 GAMES TO ME, 0 GAMES TO YOU
```

The value of each number on the tiles held at the end of a game is totalled (and note "27" counts as two plus seven, not twenty-seven), and the player with the *lowest* total wins. There are five games in a round, and the winner is the player who wins the majority of those games:

```
MY TILES:
::::::::::::::

YOUR TILES:
52:30::00:22::::::::::


AT THE END OF THAT GAME, YOUR SCORE
     IS 14 AND MINE IS 0

       SO I'M THE WINNER


 4 GAMES TO ME, 1 GAMES TO YOU


THAT'S FIVE GAMES WE'VE PLAYED
      AND I'M THE OVERALL WINNER

       OK, THANKS FOR THE GAMES
```

You'll find the Amstrad a difficult opponent to defeat in this game of *Dominoes:*

```
10 RANDOMIZE TIME
20 REM DOMINOES
30 DIM D$(28),H$(14),C$(14)
40 MODE 1:INK 0,0:INK 1,18:INK 2,24:INK
3,8:BORDER 0
50 X=0:Y=0
60 RANDOMIZE TIME
70 HGAME=0:CGAME=0:REM TOTALS WON
80 GOSUB 1750:REM INITIALISE
90 REM --------------------
100 REM MAIN CYCLE
110 WHILE NOT FINISHED
120 NTFLAG=0
130 TFLAG=0
```

```
140 GOSUB 630:REM AMSTRAD MOVES
150 GOSUB 1440:IF CLEFT=14 THEN 280
160 IF NTFLAG=1 THEN 190
170 IF TFLAG=0 THEN 200
180 LOCATE 1,1:PRINT "I'LL TAKE A TILE F
ROM THE BONEYARD":GOSUB 1720
190 MF=9
200 GOSUB 1590:REM PRINTOUT
210 SOUND 1,282:SOUND 2,240:SOUND 4,188:
SOUND 2,224:SOUND 4,188
220 GOSUB 1040:REM HUMAN MOVES
230 GOSUB 1520:IF HLEFT=14 THEN 280
240 GOSUB 1590:REM PRINTOUT
250 IF NTFLAG=2 THEN 280:REM GAME OVER
260 WEND
270 REM ----------------
280 REM END OF GAME
290 CLS
300 LOCATE 1,3:PRINT "END OF GAME":PRINT
:GOSUB 1720
310 MF=9
320 GOSUB 1650:PRINT:PRINT:PEN 1
330 CS=0:HS=0
340 PRINT "MY TILES:"
350 FOR J=1 TO 14:PEN 1:PRINT C$(J);:PEN
 3:PRINT ":";:PEN 1:NEXT J:PRINT:PRINT
360 PRINT:PRINT:PRINT "YOUR TILES:"
370 FOR J=1 TO 14:PEN 2:PRINT H$(J);:PEN
 3:PRINT ":";:PEN 2:NEXT J:PRINT:PRINT
380 FOR J=1 TO 14
390 IF C$(J)<>"" THEN CS=CS+VAL(RIGHT$(C
$(J),1))+VAL(LEFT$(C$(J),1))
400 IF H$(J)<>"" THEN HS=HS+VAL(RIGHT$(H
$(J),1))+VAL(LEFT$(H$(J),1))
410 H$(J)="":C$(J)=""
420 NEXT J
430 PRINT:PRINT "AT THE END OF THAT GAME
, YOUR SCORE"
440 PRINT TAB(6);"IS"HS"AND MINE IS"CS
450 PEN 3:PRINT:PRINT TAB(8);
460 IF CS=HS THEN PRINT "IT'S A DRAW"
470 IF CS<HS THEN PRINT"SO I'M THE WINNE
R":CGAME=CGAME+1
```

```
480 IF HS<CS THEN PRINT "SO YOU'RE THE W
INNER":HGAME=HGAME+1
490 PRINT:PRINT
500 PRINT CGAME"GAMES TO ME,"HGAME"GAMES
 TO YOU"
510 PRINT:PRINT
520 IF CGAME+HGAME<5 THEN 570
530 PRINT "THAT'S FIVE GAMES WE'VE PLAYE
D"
540 IF CGAME>HGAME THEN PRINT TAB(7);"AN
D I'M THE OVERALL WINNER"
550 IF CGAME<HGAME THEN PRINT TAB(5);"AN
D YOU'RE THE OVERALL WINNER"
560 GOTO 600
570 PRINT "ENTER 'Y' FOR A NEW GAME, 'N'
 TO END"
580 L$="":WHILE L$<>"Y" AND L$<>"N":L$=I
NKEY$:L$=UPPER$(L$):WEND
590 IF L$="Y" THEN CLS:GOTO 80
600 PRINT:PRINT TAB(8);"OK, THANKS FOR T
HE GAMES"
610 END
620 REM ----------------------
630 REM AMSTRAD MOVES
640 MF=1
650 IF P$="" THEN 870:REM FIRST MOVE
660 LOCATE 1,22:PEN 1:PRINT "BEFORE THIS
 MOVE I HAVE";:PEN 2:PRINT 14-CLEFT;:PEN
 1:PRINT "TILES          "
670 GOSUB 1720
680 X=0:FLAG=0
690 X=X+1
700 IF LEFT$(C$(X),1)=LEFT$(P$,1) THEN F
LAG=1:GOSUB 960:RETURN
710 IF RIGHT$(C$(X),1)=LEFT$(P$,1) THEN
FLAG=2:GOSUB 960:RETURN
720 IF LEFT$(C$(X),1)=RIGHT$(P$,1) THEN
FLAG=3:GOSUB 960:RETURN
730 IF RIGHT$(C$(X),1)=RIGHT$(P$,1) THEN
 FLAG=4:GOSUB 960:RETURN
740 IF X<14 THEN 690
750 LOCATE 1,22:PRINT "I CANNOT MOVE...
                      ":LOCATE 1,3:PRINT S
PC(39):GOSUB 1720
```

```
760 X=X+1
770 IF D$(X)="" THEN 820
780 Y=0
790 Y=Y+1
800 IF C$(Y)="" THEN C$(Y)=D$(X):D$(X)="
":TFLAG=1:RETURN
810 IF Y<14 THEN 790
820 IF X<28 THEN 760
830 LOCATE 1,22:PRINT "THERE ARE NO TILE
S LEFT IN BONEYARD"
840 GOSUB 1720
850 NTFLAG=NTFLAG+1
860 RETURN
870 REM ------ FIRST MOVE ----------
880 X=0
890 X=X+1
900 IF LEFT$(C$(X),1)=RIGHT$(C$(X),1) TH
EN 930:REM DOUBLE FOUND
910 IF X<7 THEN 890
920 X=INT(RND*7)+1
930 P$=C$(X):C$(X)=""
940 RETURN
950 REM ------------------------------
960 REM MAKE MOVE
970 IF FLAG=1 THEN P$=RIGHT$(C$(X),1)+LE
FT$(C$(X),1)+":"+P$
980 IF FLAG=2 THEN P$=C$(X)+":"+P$
990 IF FLAG=3 THEN P$=P$+":"+C$(X)
1000 IF FLAG=4 THEN P$=P$+":"+RIGHT$(C$(
X),1)+LEFT$(C$(X),1)
1010 C$(X)=""
1020 RETURN
1030 REM ------------------------------
1040 REM HUMAN MOVES
1050 MF=2
1060 PEN 3:LOCATE 1,16:PRINT "HERE IS YO
UR HAND:":PRINT:PEN 2
1070 FOR G=1 TO 14
1080 IF H$(G)<>"" THEN PRINT G"- "H$(G),
1090 NEXT G
1100 PEN 1:LOCATE 1,22:INPUT "ENTER NUMB
ER OF PIECE YOU WANT TO MOVE, 0 TO PASS
";M:SOUND 1,160,6,15,1,1,0
1110 IF M<0 OR M>14 THEN 1100
```

```
1120 IF M<>0 THEN IF H$(M)="" THEN 1100
1130 LOCATE 1,16:FOR J=1 TO 8:PRINT SPC(
39):NEXT J
1140 IF M=0 THEN 1280
1150 SOUND 4,188:SOUND 2,224:SOUND 4,188
:SOUND 1,288:SOUND 2,199
1160 PEN 2:LOCATE 1,17:PRINT ">  ";H$(M)

1170 PEN 3:LOCATE 1,20:PRINT "AT THE STA
RT (S) OR THE END (E)?"
1180 A$="":WHILE A$<>"S" AND A$<>"E":A$=
INKEY$:A$=UPPER$(A$):WEND
1190 SOUND 5,190,6,15,0,0,0:SOUND 5,210,
6,15,0,0,0:IF A$="E" THEN 1220
1200 IF RIGHT$(H$(M),1)=LEFT$(P$,1) THEN
 P$=H$(M)+":"+P$:GOTO 1240
1210 P$=RIGHT$(H$(M),1)+LEFT$(H$(M),1)+"
:"+P$:GOTO 1240
1220 IF LEFT$(H$(M),1)=RIGHT$(P$,1) THEN
 P$=P$+":"+H$(M):GOTO 1240
1230 P$=P$+":"+RIGHT$(H$(M),1)+LEFT$(H$(
M),1)
1240 H$(M)=""
1250 LOCATE 1,17:FOR J=1 TO 5:PRINT SPC(
39):NEXT J
1260 RETURN
1270 REM ----------------------------
1280 REM HUMAN PASS
1290 X=0
1300 X=X+1
1310 IF H$(X)="" THEN 1330
1320 IF X<14 THEN 1300
1330 Y=14
1340 Y=Y+1
1350 IF D$(Y)<>"" THEN 1400
1360 IF Y<28 THEN 1340
1370 LOCATE 1,22:PRINT "NO TILES LEFT IN
 BONEYARD"
1380 NTFLAG=NTFLAG+1
1390 RETURN
1400 H$(X)=D$(Y):D$(Y)=""
1410 LOCATE 1,22:PEN 1:PRINT "YOU HAVE D
RAWN ";:PEN 2:PRINT ":";:PEN 3:PRINT H$(
X);:PEN 2:PRINT ":"
```

```
1420 GOSUB 1720
1430 RETURN
1440 REM --------------------
1450 REM AMSTRAD OUT?
1460 CLEFT=0
1470 FOR J=1 TO 14
1480 IF C$(J)="" THEN CLEFT=CLEFT+1
1490 NEXT J
1500 RETURN
1510 REM --------------------
1520 REM HUMAN OUT?
1530 HLEFT=0
1540 FOR J=1 TO 14
1550 IF H$(J)="" THEN HLEFT=HLEFT+1
1560 NEXT J
1570 RETURN
1580 REM ---------------------
1590 REM PRINT OUT
1600 PEN 1
1610 LOCATE 1,1:PRINT SPC(39):PRINT SPC(
39):PRINT
1620 IF MF=9 THEN 1650
1630 IF MF=1 THEN PRINT "AFTER MY MOVE,
";:GOTO 1650
1640 PRINT "AFTER YOUR MOVE, ";
1650 PRINT "THIS IS THE BOARD:    ":PRIN
T "
 "
1660 PRINT:PRINT
1670 FOR K=1 TO LEN(P$)
1680 Y$=MID$(P$,K,1)
1690 PEN 2:IF Y$=":" THEN PEN 3
1700 PRINT Y$;
1710 NEXT K
1720 FOR DELAY=1 TO 1000:NEXT DELAY
1730 RETURN
1740 REM --------------------
1750 REM INITIALISATION
1760 SOUND 1,282:SOUND 2,240:SOUND 4,188
:SOUND 2,224:SOUND 4,188
1770 PRINT:PRINT:PRINT:PEN 1:PRINT TAB(8
);"STAND ";:PEN 2:PRINT "BY ";:PEN 3:PRI
NT "CHAMP..."
1780 REM -- SET UP DOMINOES --
```

```
1790 X=0
1800 FOR J=0 TO 6
1810 FOR K=0 TO J
1820 X=X+1
1830 D$(X)=RIGHT$(STR$(J),1)+RIGHT$(STR$
(K),1)
1840 NEXT K
1850 NEXT J
1860 REM -- SHUFFLE DOMINOES --
1870 REM MOSES/OAKFORD ROUTINE (1963)
1880 FOR J=28 TO 1 STEP -1
1890 T=INT(RND*J)+1
1900 H$=D$(T)
1910 D$(T)=D$(J)
1920 D$(J)=H$
1930 NEXT J
1940 REM -- ALLOT HANDS --
1950 FOR J=1 TO 7
1960 H$(J)=D$(J):D$(J)=""
1970 C$(J)=D$(J+7):D$(J+7)=""
1980 NEXT J
1990 P$="":REM HOLDS CURRENT BOARD
2000 SYMBOL AFTER 48
2010 SYMBOL 48,60,66,66,98,98,98,60,0
2020 SYMBOL 49,8,8,8,12,12,12,12,0
2030 SYMBOL 50,124,2,2,60,96,96,62,0
2040 SYMBOL 51,124,2,2,60,6,6,126,0
2050 SYMBOL 52,64,96,100,100,126,4,4,0
2060 SYMBOL 53,62,64,64,60,6,6,126,0
2070 SYMBOL 54,62,64,64,124,70,70,126,0
2080 SYMBOL 55,120,8,8,12,12,12,12,0
2090 SYMBOL 56,60,66,66,60,70,70,60,0
2100 SYMBOL 57,60,66,66,60,6,6,126,0
2110 CLS
2120 RETURN
```

Oliver Rand

Way back in 1972, the British publisher George Allen & Unwin Ltd. published the book *Games Playing with Computers,* by A G Bell. Although this is really not so long ago in terms of the history of the universe, it is an age away in the history of computers.

"... it is very likely," Bell writes in his introduction, "that future generations will use them (computers) in their leisure time to interact with game playing programs." Indeed it is very likely, Mr Bell, and your book provided me with the germ of our next program which "future generations" may well be playing.

Mr Bell describes a game called *Bridget* which, he says, a computer can easily be taught to play. In it, the players take it in turns to draw a line on a grid of dots, with the idea of trying to create an unbroken line from their 'base' to their 'goal' line.

Here's the board:

```
   C     C     C     C     C
H 37 H 38 H 39 H 40 H 41 H
   C 33  C 34  C 35  C 36  C
H 28 H 29 H 30 H 31 H 32 H
   C 24  C 25  C 26  C 27  C
H 19 H 20 H 21 H 22 H 23 H
   C 15  C 16  C 17  C 18  C
H 10 H 11 H 12 H 13 H 14 H
   C  6  C  7  C  8  C  9  C
H 1  H 2  H 3  H 4  H 5  H
   C     C     C     C     C
```

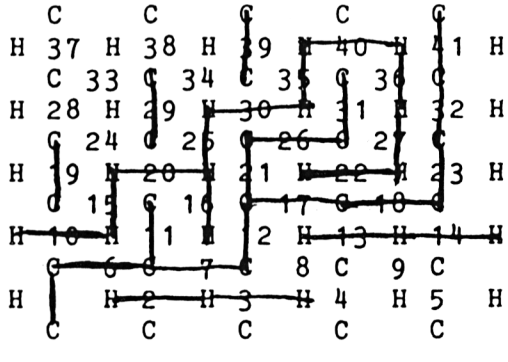
The Amstrad's base line is the bottom of the screen, and it is working its way to the goal at the top. You start with your base on the left, and work towards the goal of the right. You move by selecting a number which lies between two H's, such as the 6 next the bottom left hand corner of the board. You need to print out the board using your printer, and to draw in the lines you select, and those selected by the computer, on that printout. If you don't have a printer, you can copy out the board by hand, and play on this board.

Here's the 'conversation' of part of a game in progress:

```
ENTER YOUR MOVE  36

MY MOVE IS 32
```

```
MY MOVE IS 23

   ENTER YOUR MOVE  22
MY MOVE IS 26

   ENTER YOUR MOVE  14
MY MOVE IS 18

   ENTER YOUR MOVE  13
MY MOVE IS 17

   ENTER YOUR MOVE  0
I ACCEPT YOUR WISH TO CONCEDE
```

And here's the hand-completed board. In this game, the computer has won, with a line joining the following points: 1, 6, 7, 12, 17, 18, 23, 32 and 41:



The winning strategy, encoded in this program, was devised by Oliver Gross of the Rand Corporation, and is outlined in A G Bell's book (page 23) and that's why I've called it *Oliver Rand.* When you're ready to take on the

hopeless task of trying to beat your Amstrad, run the following listing:

```
10 REM OLIVER RAND
20 GOSUB 440:REM INITIALISE
30 GOSUB 280:REM DUMP BOARD TO PRINTER
40 REM ---------------
50 REM MAIN CYCLE
60 WHILE NOT FINISHED
70 SOUND 5,100:SOUND 4,160:SOUND 1,60
80 GOSUB 130:REM AMSTRAD MOVES
90 GOSUB 170:REM HUMAN MOVES
100 SOUND 5,100:SOUND 4,160:SOUND 1,60
110 WEND
120 REM AMSTRAD MOVES
130 MVE=R(HM)
140 PEN 1:PRINT:PRINT "MY MOVE IS";:PEN
3:PRINT MVE
150 RETURN
160 REM -----------------
170 REM HUMAN MOVES
180 REM ENTER 0 TO CONCEDE
190 PRINT:PRINT
200 PEN 2:INPUT "    ENTER YOUR MOVE    "
;HM
210 IF HM=0 THEN 240
220 IF HM<0 OR HM>41 THEN 190
230 RETURN
240 PEN 3:PRINT:PRINT "I ACCEPT YOUR WIS
H TO CONCEDE!"
250 FOR J=1 TO 20:PRINT:FOR Z=1 TO 100:N
EXT Z:NEXT J
260 END
270 REM -----------------
280 REM DUMP BOARD TO PRINTER
290 PRINT #8:PRINT #8
300 PRINT #8, "  C    C    C    C    C"
310 PRINT #8,"H 37 H 38 H 39 H 40 H 41 H
"
320 PRINT #8,"  C 33 C 34 C 35 C 36 C"
330 PRINT #8,"H 28 H 29 H 30 H 31 H 32 H
"
340 PRINT #8,"  C 24 C 25 C 26 C 27 C"
350 PRINT #8,"H 19 H 20 H 21 H 22 H 23 H
"
```

```
360 PRINT #8,"  C 15 C 16 C 17 C 18 C"
370 PRINT #8,"H 10 H 11 H 12 H 13 H 14 H
"
380 PRINT #8,"  C  6 C  7 C  8 C  9 C"
390 PRINT #8,"H 1  H 2  H 3  H 4  H 5  H
"
400 PRINT #8,"  C    C    C    C    C"
410 CLS
420 RETURN
430 REM ------------------------
440 REM INITIALISE
450 MODE 1:INK 0,3:INK 1,18:INK 2,26:INK
 3,24,3:BORDER 3:SPEED INK 6,6
460 LOCATE 2,5:PEN 1:PRINT "PLEASE..."
470 PEN 2:PRINT:PRINT "      STAND..."
480 PEN 3:PRINT:PRINT "            BY...
"
490 HM=1:REM FIRST MOVE BY MACHINE
500 DIM R(41)
510 REM -- READ RESPONSES --
520 FOR Z=1 TO 41
530 READ R(Z)
540 NEXT Z
550 RETURN
560 REM -------------------
570 REM RESPONSE DATA
580 DATA 1,6,7,8,9,2,3,4,5
590 DATA 11,10,16,17,18,19
600 DATA 12,13,14,15,21,20
610 DATA 26,27,28,29,22,23
620 DATA 24,25,31,30,36,37
630 DATA 38,39,32,33,34,35,41,40
```

Celestia

We're going to look at a program now which fills the Amstrad screen with a very attractive, constantly evolving design. The patters develop according to the rules of the famous computer game of LIFE, developed by John Conway while at Gonville and Caius College, Cambridge. Martin Gardiner spread the game throughout the world when he wrote about it in Scientific American magazine in October, 1970.

In LIFE, cells are born, grow and die according to rules which Conway

invented. Each cell on a grid (the colony of cells is imagined to be evolving on a grid) is surrounded by eight orders, and the state of those eight other cells dictates what happens to the cell in question in the following generation.

The rules which govern the evolution of the cells are as follows:

*** If a cell has two or three surrounding it, it survives to the next generation**

*** If there are three, and just three, full cells next to an empty one, a cell will be 'born' in that empty space in the next generation**

*** Any cell with four or its neighbouring cells occupied dies in the next generation**

Don't worry. You don't have to know the rules, as your Amstrad interprets them quite happily by itself. The rules produce patterns which are far more attractive (and far less predictable) than you could possibly imagine by reading the rules.

There is just one extra twist to this program which makes it even more interesting. CELESTIA actually prints out four colonies each time a colony evolves. The original colony is in one quadrant of the screen, and the other three-quarters of the screen contain reflections of the original one.

CELESTIA is incredibly effective, as you will see.

```
10 ' CELESTIA
20 DEFINT A-Z
30 GOSUB 530: ' INITIALISE
40 ' ************************
50 WHILE NOT K
60 GOSUB 100: '  PRINTOUT
70 GOSUB 310: ' EVOLVE
80 WEND
90 ' ************************
100 LOCATE 2,1
110 FOR X=2 TO 13
120 FOR Y=2 TO 20
130 PRINT CHR$(A(X,Y));
140 NEXT Y
150 FOR Y=20 TO 2 STEP-1
```

```
160 PRINT CHR$(A(X,Y));
170 NEXT Y
180 PRINT:PRINT " ";
190 NEXT X
200 FOR X=13 TO 2 STEP-1
210 FOR Y=2 TO 20
220 PRINT CHR$(A(X,Y));
230 NEXT Y
240 FOR Y=20 TO 2 STEP-1
250 PRINT CHR$(A(X,Y));
260 NEXT Y
270 PRINT:PRINT " ";
280 NEXT X
290 RETURN
300 ' **********************
310 FOR X=2 TO 13
320 FOR Y=2 TO 20
330 C=0
340 IF A(X-1,Y-1)=B THEN C=C+1
350 IF A(X-1,Y)=B THEN C=C+1
360 IF A(X-1,Y+1)=B THEN C=C+1
370 IF A(X,Y-1)=B THEN C=C+1
380 IF A(X,Y+1)=B THEN C=C+1
390 IF A(X+1,Y-1)=B THEN C=C+1
400 IF A(X+1,Y)=B THEN C=C+1
410 IF A(X+1,Y+1)=B THEN C=C+1
420 IF A(X,Y)=B AND C<>3 AND C<>2 THE
N B(X,Y)=E
430 IF A(X,Y)=E AND C=3 THEN B(X,Y)=B
440 NEXT Y
450 NEXT X
460 FOR X=2 TO 13
470 FOR Y=2 TO 20
480 A(X,Y)=B(X,Y)
490 NEXT Y
500 NEXT X
510 RETURN
520 ' **********************
530 ' INITIALISE
540 CLS
550 PRINT:PRINT "Press any key..."
560 SOUND 1,RND(1)*406+400
570 IF INKEY$="" THEN 550
580 CLS
```

```
590 B=ASC("*"):E=ASC(" ")
600 DIM A(14,21),B(14,21)
610 FOR X=2 TO 13
620 FOR Y=2 TO 20
630 A(X,Y)=E
640 IF RND>0.45 THEN A(X,Y)=B
650 B(X,Y)=A(X,Y)
660 NEXT Y
670 NEXT X
680 RETURN
```

CPC-Thello

This game, as I'm sure you've realised by now, is the Amstrad's version of Reversi, or Othello. Invented in the late eighteen hundreds, the game is played on an ordinary eight by eight checkerboard. When played on a board, you use pieces which have different colours on each side. The game begins with four pieces placed on the centre squares.

From this point on, you move by placing one of your pieces next to an opponent's piece or pieces, with another of your pieces further on. When that happens, all the opponent pieces 'reverse' to become your pieces.

Here's how it works. Suppose a line of pieces looked like this:

OXXXX

If you decided to put your piece (the O) at the end of the line like this:

OXXXXO

The opponent's pieces would reverse, so the line looked like this:

OOOOOO

The 'reversing' also takes place along vertical and diagonal rows, so many, many pieces may be changed with a single move.

The game continues until every square on the board is filled, or when neither player can move. If you cannot move, you signal this by entering a zero when it is your turn to move. For all other moves, you enter the co-ordinates of the position into which you wish to move, touching the side number first (you'll see, when you play this, that there are the numbers one

to eight down the sides, and across the top and bottom) followed by the number across. There is no need to use the ENTER key when telling the computer the moves you wish to make.

You'll find this program makes very effective use of Amstrad colour, with a large and clearly displayed board. The playing algorithm was developed by Graham Charlton, and the Amstrad adaptation and display is by Tim Hartnell.

```
10 REM CPC-THELLO
20 REM ALGORITHM BY GRAHAM CHARLTON
30 GOSUB 870:REM INITIALISATION
40 PEN 1
50 LOCATE 28,14:PRINT "MY MOVE"
60 S=OX:T=X:H=0
70 FOR A=2 TO 9:FOR B=2 TO 9
80 IF A(A,B)<>46 THEN 230
90 Q=0
100 FOR C=-1 TO 1:FOR D=-1 TO 1
110 K=0:F=A:G=B
120 IF A(F+C,G+D)<>S THEN 150
130 K=K+1:F=F+C:G=G+D
140 GOTO 120
150 IF A(F+C,G+D)<>T THEN 170
160 Q=Q+K
170 NEXT:NEXT
180 IF A=2 OR A=9 OR B=2 OR B=9 THEN Q=Q
*2
190 IF A=3 OR A=8 OR B=3 OR B=8 THEN Q=Q
/2
200 IF (A=2 OR A=9) AND (B=3 OR B=8) OR
(A=3 OR A=8) AND (B=2 OR B=9) THEN Q=Q/2
210 IF Q<H OR (RND<0.3 AND Q=H) THEN 230
220 H=Q:M=A:N=B
230 NEXT:NEXT
240 IF H=0 AND R=0 THEN 800
250 IF H=0 THEN 270
260 GOSUB 690
270 GOSUB 480
280 PEN 2
290 LOCATE 27,14
300 REM ENTER ZERO TO PASS
310 PRINT "YOUR MOVE ";
```

```
320 R$="":WHILE R$<"0" OR R$>"H":R$=INKE
Y$:R$=UPPER$(R$):WEND:PRINT R$;:SOUND 5,
100
330 IF R$="0" THEN R=0:GOTO 380
340 Z$="":WHILE Z$<"1" OR Z$>"8":Z$=INKE
Y$:WEND:R$=R$+Z$:PRINT Z$:SOUND 5,200
350 IF LEN(R$)<>2 OR ASC(R$)<65 OR ASC(R
$)>72 THEN 290
360 R=(ASC(LEFT$(R$,1))-64)*10+VAL(RIGHT
$(R$,1))
370 IF R<11 OR R>88 THEN 290
380 S=X:T=OX
390 IF R=0 THEN 450
400 IF R<11 OR R>88 THEN 290
410 R=R+11
420 N=INT(R/10)
430 M=R-10*N
440 GOSUB 690
450 GOSUB 480
460 GOTO 40
470 REM ------------------------
480 REM PRINT BOARD
490 SOUND 5,RND*200+50
500 LOCATE 26,14:PRINT "                    "
510 C=0:H=0
520 LOCATE 1,1
530 FOR B=2 TO 9
540 FOR D=2 TO 9
550 SS=3
560 IF A(B,D)=X THEN C=C+1:SS=1
570 IF A(B,D)=OX THEN H=H+1:SS=2
580 PEN SS
590 LOCATE 2*B,2*D:PRINT CHR$(136);CHR$(
140)
600 LOCATE 2*B,2*D+1:PRINT CHR$(138);CHR
$(143)
610 NEXT
620 NEXT
630 PEN 1
640 LOCATE 28,5:PRINT "AMSTRAD:"C
650 PEN 2
660 LOCATE 28,7:PRINT "HUMAN:"H
670 IF C+H=64 OR C=0 OR H=0 THEN 810
680 RETURN
```

```
690 FOR C=-1 TO 1:FOR D=-1 TO 1
700 F=M:G=N
710 IF A(F+C,G+D)<>S THEN 740
720 F=F+C:G=G+D
730 GOTO 710
740 IF A(F+C,G+D)<>T THEN 780
750 A(F,G)=T
760 IF M=F AND N=G THEN 780
770 F=F-C:G=G-D:GOTO 750
780 NEXT:NEXT
790 RETURN
800 GOSUB 480
810 LOCATE 22,14:PEN 3
820 IF C>H THEN PRINT "I'M THE CHAMP!"
830 IF H>C THEN PRINT "YOU'RE THE CHAMP!
"
840 IF H=C THEN PRINT "IT'S A DRAW!"
850 LOCATE 1,24:END
860 REM ------------------
870 REM INITIALISATION
880 RANDOMIZE TIME
890 MODE 1:INK 0,1:INK 1,6:INK 2,24:INK
3,14:BORDER 1:PEN 2
900 DEFINT A-Z
910 X=88:OX=79
920 FOR B=2 TO 9:FOR C=2 TO 9
930 A(B,C)=46
940 NEXT:NEXT
950 A(5,5)=X:A(6,6)=X
960 A(6,5)=OX:A(5,6)=OX
970 FOR K=1 TO 2
980 FOR J=1 TO 8
990 READ FF,GG
1000 LOCATE GG,FF
1010 IF K=1 THEN PRINT RIGHT$(STR$(J),1)
 ELSE PRINT CHR$(J+64)
1020 NEXT:NEXT
1030 GOSUB 480
1040 RETURN
1050 REM ----------
1060 REM -- DATA --
1070 DATA 21,4,21,6,21,8,21,10,21,12,21,
14,21,16,21,18
1080 DATA 5,2,7,2,9,2,11,2,13,2,15,2,17,
2,19,2
```

Squisher and Squasher

Squisher and Squasher are two evil little creatures who are determined to destroy you as you move around a maze, trying to survive for as long as possible. As you make your way around the maze, you try to eat as many dots as you can (by running over them). Each dot you run over adds to your score. You do not get points just for surviving.

As Squisher and Squasher move in towards you, they leave a trail of new dots. Therefore, if you can fool them into crossing areas which you have already 'eaten clean', you'll find more dots to eat.

You move around using the cursor (arrow) keys. You'll find that giving them clean 'taps', rather than leaning on a key continuously, will give the best results. The character which represents you (and which begins in the top left hand corner) will continue to move in the direction you've sent it, until you press an arrow key indicating that you wish to move in a new direction.

The maze changes slightly from game to game, and neither you, nor Squisher and Squasher can move through the fences. However, while you can only move up, down, right or left, the two protagonists can move along these directions, as well as along the diagonals if they wish.

```
10 REM SQUASHER AND SQUISHER
20 GOSUB 950:REM INITIALISE
30 WHILE NOT CAUGHT
40 GOSUB 290:REM AMSTRAD MOVES
50 A$=INKEY$:IF LEN(A$)<>0 THEN B$=A$
60 IF C1=H1 AND C2=H2 THEN 110
70 A$=INKEY$:IF LEN(A$)<>0 THEN B$=A$
80 IF E1=H1 AND E2=H2 THEN 110
90 GOSUB 790
100 GOTO 40
110 INK 1,6,24:INK 2,0,26
120 PEN 1:LOCATE 1,16
130 PRINT "SQUASHER AND SQUISHER HAVE GO
T YOU!"
140 PRINT SPC(39)
150 PRINT TAB(13);"You scored";:PEN 2:PR
INT SC;:PEN 1:PRINT "!!!!!!!!"
160 IF SC>HS THEN HS=SC
170 PRINT SPC(39)
```

```
180 PEN 3:PRINT TAB(9);"The high score i
s";:PEN 2:PRINT HS
190 FOR Z=1 TO 41 STEP 3
200 FOR ZZ=1 TO 4:LOCATE H1+12,H2:PEN (I
NT(RND*3)+1):PRINT CHR$(240+INT(RND*5)):
NEXT ZZ
210 SOUND 5,Z*10:SOUND 5,400-Z*10
220 NEXT Z
230 RESTORE
240 CLS
250 GOSUB 1000
260 INK 1,26:INK 2,5,21:WEND
270 END
280 REM ----------------
290 REM S & S MOVE
300 X=C1:Y=C2
310 IF RND>0.7 THEN 340
320 IF C1>H1 THEN C1=C1-1
330 IF C1<H1 THEN C1=C1+1
340 IF RND>0.75 THEN 370
350 IF C2>H2 THEN C2=C2-1
360 IF C2<H2 THEN C2=C2+1
370 M=A(C1,C2)
380 IF M=72 OR M=36 OR M=35 THEN C1=X:C2
=Y
390 A(X,Y)=46
400 PEN 1:LOCATE X+12,Y:PRINT "."
410 A(C1,C2)=240
420 PEN 2:LOCATE C1+12,C2:PRINT CHR$(240
)
430 X=E1:Y=E2
440 IF RND>0.6 THEN 470
450 IF E1>H1 THEN E1=E1-1
460 IF E1<H1 THEN E1=E1+1
470 IF RND>0.75 THEN 500
480 IF E2>H2 THEN E2=E2-1
490 IF E2<H2 THEN E2=E2+1
500 M=A(E1,E2)
510 IF M=72 OR M=63 OR M=35 THEN E1=X:E2
=Y
520 IF E1=C1 AND E2=C2 THEN E1=X:E2=Y
530 A(X,Y)=46
540 PEN 1:LOCATE X+12,Y:PRINT "."
550 A(E1,E2)=2
```

```
560 PEN 2:LOCATE E1+12,E2:PRINT CHR$(241
)
570 LOCATE 1,16
580 PEN 3
590 Q2=SQR((H1-E1)*(H1-E1)+(H2-E2)*(H2-E
2))
600 PRINT "> Squisher is"Q1"from human
 "
610 PRINT TAB(7);"Squasher is"Q2"from hu
man       "
620 PEN 2:PRINT TAB(8);"Your score is";
SC
630 PEN 3
640 IF HS>0 THEN PRINT " Hiscore";:PEN 1
:PRINT HS
650 RETURN
660 REM ---------------
670 LOCATE 13,1
680 FOR C=1 TO 12
690 FOR B=1 TO 12
700 IF A(B,C)=46 THEN PEN 1:GOTO 730
710 IF A(B,C)=240 OR A(B,C)=241 THEN PEN
 2:GOTO 730
720 PEN 3
730 PRINT CHR$(A(B,C));
740 NEXT B
750 PRINT:PRINT TAB(13);
760 NEXT C
770 RETURN
780 REM ----------------
790 REM HUMAN MOVES
800 GOSUB 570
810 A$=INKEY$:IF LEN(A$)<>0 THEN B$=A$ E
LSE A$=B$
820 G=H1:F=H2
830 A(H1,H2)=32
840 IF A$=CHR$(242) THEN H1=H1-1:IF H1=0
 THEN H1=1
850 IF A$=CHR$(243) THEN H1=H1+1:IF H1=1
3 THEN H1=12
860 IF A$=CHR$(240) THEN H2=H2-1:IF H2=0
 THEN H2=1
870 IF A$=CHR$(241) THEN H2=H2+1:IF H2=1
3 THEN H2=12
```

```
880 IF A(H1,H2)=35 THEN H1=G:H2=F
890 IF A(H1,H2)=46 THEN SC=SC+17
900 A(H1,H2)=243
910 LOCATE G+12,F:PEN 1:PRINT CHR$(244)
920 PEN 3:LOCATE H1+12,H2:PRINT CHR$(243
)
930 RETURN
940 REM ----------------
950 REM INITIALISE
960 MODE 1:INK 0,0:BORDER 18:INK 1,26:IN
K 2,5,21:INK 3,15:BORDER 0:SPEED INK 3,3
970 DEFINT A-Z
980 HS=0:REM HIGH SCORE
990 DIM A(12,12)
1000 H1=1:H2=1:REM HUMAN START
1010 C1=12:C2=12:REM SQUISHER START
1020 E1=12:E2=1:REM  SQUASHER START
1030 FOR B=1 TO 12
1040 FOR C=1 TO 12
1050 A(B,C)=46
1060 NEXT C
1070 NEXT B
1080 FOR B=1 TO 54
1090 READ C,D
1100 A(C,D)=35
1110 NEXT B
1120 FOR T=1 TO 25
1130 Q1=RND*11+1:Q2=RND*11+1
1140 A(Q1,Q2)=46
1150 NEXT T
1160 B$=CHR$(241)
1170 A(H1,H2)=243
1180 A(E1,E2)=241
1190 A(C1,C2)=240
1200 SC=0
1210 SYMBOL AFTER 35
1220 SYMBOL 35,24,102,102,129,129,102,10
2,24
1230 SYMBOL 46,0,0,60,36,36,60,0,0
1240 SYMBOL 240,30,63,47,127,55,7,62,30
1250 SYMBOL 241,24,126,153,255,126,126,1
26,24
1260 SYMBOL 243,0,73,42,28,28,42,73,0
1270 SYMBOL 244,0,96,144,16,8,9,6,0
```

```
1280 GOSUB 670
1290 SC=0
1300 RETURN
1310 REM ------------------------
1320 DATA 3,1,10,1,3,2,5,2,6,2,7,2,8,2
1330 DATA 10,2,2,3,3,3,5,3,8,3,9,3,11,3
1340 DATA 5,4,5,8,1,5,3,5,4,5,5,5
1350 DATA 8,5,9,5,10,5,12,5,9,7
1360 DATA 1,6,3,6,5,6,12,6,2,8,3,8
1370 DATA 4,8,9,8,10,8,11,8,7,9,12,9
1380 DATA 1,10,2,10,10,10,5,10,7,10,12,1
0
1390 DATA 5,11,7,11,9,11,10,11,12,11
1400 DATA 2,12,3,12,4,12,5,12,11,12,12,1
2
```

Creating Your Own Games

It's all very well entering and running the programs in this section of the book, but I'm sure — eventually, if you haven't already done so — you'll want to write your own games. And this is where the problem can come in. "I'm OK once I know what the game is meant to do," is a common lament, "but I can't think of any ideas to turn into games programs."

The next part should help you. It contains a number of games ideas which you should be able to use as the basis of your own computer programs. And don't worry if the program which you end up with bears little resemblance to the game you thought you were going to write. Many of the best games I've written evolved from rather primitive ideas which were very different from the program I ended up with. Use these as ideas starters, and see what you and the computer can come up with.

If you have a 'traditional' game (like bridge, or chess) which you want to computerise, and you don't have a clue how to work out an algorithm which the computer can use, follow the lead of programs in this book like *Oliver Rand*, where the computer holds all the possible moves in DATA statements. Then, your program will consist — in large measure — of routines to print out the current state of play, accept the player's move, and decide (on the basis of the entered move by the player) which of the 'prepackaged' responses should be used. Programs which use look-up tables of this type may seem somewhat long, and lacking in elegance, in comparison with those which use some sophisticated move-generation

routines, but this is unimportant. What matters is that the computer plays well, and sufficiently swiftly to ensure the human player does not lose interest. In fact, in some cases you may find the computer appears to respond too quickly. A delay loop, which gives the impression the computer is 'thinking' can help here.

Some games — such as Tic-Tac-Toe (Noughts and Crosses) — can be completely analysed, an every move (and its best response) encoded as DATA. Other games can rely on a combination of lookup tables for commonly-encountered situations and move-generating algorithms. By all means use a look-up table when you can, as this can increase the precision of your computer's response.

As well as looking at games which use such things as boards, counters, dice and cards, you can look to other aspects of human society for ideas. Simulation programs, which allow you and the computer to emulate sports, or production processes, wars and other dramatic interactions, are very rewarding. Such games, as well, can have genuine education applications.

Let's have a look, for a start, at 'traditional' games which can be computerised.

Chess

This is the ultimate test of a programmer's skill, and to write a chess program may seem — at the moment — to be an impossible task. You can 'sneak up' on the problem, as it were, by writing games which are either variants of chess (so-called 'Fairy Chess' games) or programs which use some elements from chess.

Stephen Addison's book *100 Other Games to Play on a Chessboard* (Peter Owen: London and Boston, 1983) is a fertile field of ideas for games which are chess variants. They include H Richter's game *Chess Draughts* in which each player starts the game with eight pieces: six pawns, a bishop and a king. A pawn reaching the opposite side is converted into a bishop, and the aim of the game — of course — is to capture the opponent's king.

Solomon W Golomb's *Cheskers* is also described, in which the player's start with twelve pieces each, arranged as in checkers. The pieces are (for each player) two kings; eight of a piece called 'the man' (which moves like a standard piece in checkers); a bishop; and a 'camel', which can move three squares vertically and one horizontally or one vertically and three

horizontally. The aim is to capture both of the opponent's kings.

Maharajahs

As you can see from these two descriptions, there are many ideas lurking within traditional games which are candidates for computer conversion. R C Bell, in his *The Boardgame Book* (Cavendish House, London, 1979), describes the game *Maharajah and the Sepoys* which is another chess-like game which you may wish to try and computerise. One side has a full complement of chess pieces, on their standard squares. The opponent has just one piece, a king of the opposing color. This king — the maharajah — can move either as a queen or a knight. The game ends if the maharajah is immobilized or check-mated.

Bell also describes the game *Halma* (the name is based on the Greek word for 'jump') which was invented in England in the closing years of the last century. It is played on a sixteen by sixteen board, by four people, who each start in a corner of the board. Two of the players have nineteen pieces each, and the other two have thirteen pieces each. No pieces are captured. The aim of the game is to get all your pieces into the squares which are directly opposite yours at the start of the game. You move by leaping over pieces into vacant squares beyond. Multiple jumps are permitted (but are not compulsory) and you can jump over your own or other players' pieces.

As you can see, there is the seed of a good game in this description. Perhaps you could write a game in which the computer played three of the corners, and you took on the fourth one. Add a scenario (such as 'you are out in space, trying to get your entire space fleet to a friendly star base') and you'll have an apparently totally origianl (and quite absorbing) game.

Holding the Board

A G Bell (no relation to R C Bell) is mentioned earlier in the description of the game OLIVER RAND. He is the author of the 1972 text *Games Playing with Computers* (George Allen and Unwin) in which he describes some relatively primitive algorithms for programming a computer to play such games as *Brag, Poker, Blackjack, Kalah* and *Go.* The book includes programs which will, after a fashion, play these games (along with one which could be worked into a low-level chess program). The important thing is that he points out that the way in which the board is held by the computer is very important. Make the right decision as to the form of array which will hold the board, and you're well on the way towards solving the programming problems associated with the game. Make a bad

decision, and you are adding to your problems, and perhaps ensuring that your game is almost impossible to write. This advice is very important. Take your time about determining the form in which the 'state of play' will be held, before you get involved in such things as working out how the player's moves will be accepted, or how the board will look on the screen.

Dice Games

This advice is as important for computer games which are conversions of traditional board games, as it is for conversions of sporting events or card games. Whereas board games will almost certainly need some sort of array (generally two-dimensional), other games can get by on simple variables. This holds true for nearly all dice games, where the value of each player's stake, the amounts which players have decided to bet on the outcome of a round, and the values showing on the 'dice' themselves can easily be held as variables.

For example, in the game *Liar Dice,* you'd only need a few assigned variables. CT for computer tally, HT for the human, and D1, D2, D3, D4 and D5 for the dice. In *Liar Dice,* a kind of poker played with dice instead of cards, each player takes it in turn to throw the dice. The result of the throw is not revealed immediately to the other player. Both players put money into the 'pot'. The player who has thrown the dice reveals the result of the throw (which may or may not be true). If the second player challenges the claimed throw, it must be shown. If the throw is as was claimed, or better, the other player forfeits the amount bet. Otherwise, the caster loses. If the claim is not challenged, the second player throws the dice, trying to get a (genuine) total higher than the (real) total thrown by the first. The second player does not have to reveal his or her throw, but can also bluff. This claim can be challenged by the first player, and so on. The stakes are raised, as the dice changes hands, until finally the results of the throws are revealed, and the winner is the one with the best hand. Here's how the winning throws stack up:

— five of a kind (aces high)

— five dice in sequence, ace high ('royal flush')

— five dice in sequence, king high ('low flush')

— four of a kind

— full house (three of a kind plus a pair)

— three of a kind

— two pairs

— pair

— non-scoring hand

The dice output should be shown on-screen as "Ace" (the throw is a one), "King" (6), "Queen" (5), "Jack" (4), "Ten" (3) and "Nine" (2).

Backgammon

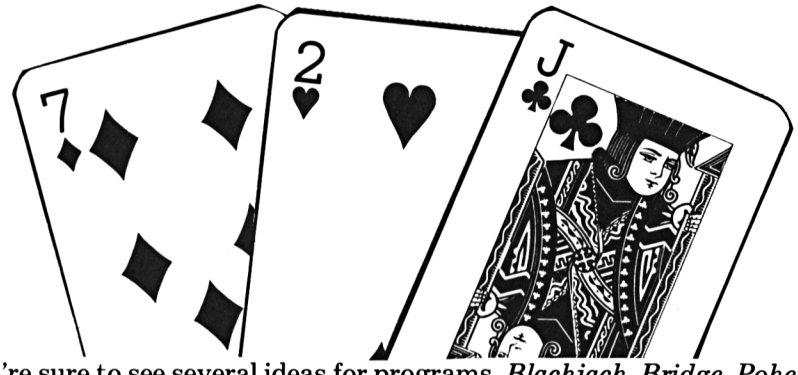
A combination of simple variables (for dice) and arrays (for the board) are called for in the game *Doublets,* an Icelandic game which was the forerunner of *Backgammon,* and is played on a *Backgammon* board. At the start of the game, each player places his or her (or its) twelve pieces, in little heaps two pieces high, on each point in the opposite right hand table. The pieces are moved off their partners, according to the dice result, and then stacked up again in pairs (again in response to dice throws) before finally being 'born off' as in *Backgammon.*

Once you've written a program to play *Doublets,* you might want to try your hand at *Backgammon.* In contrast to *Doublets,* which is totally based on luck, skill comes into play with *Backgammon,* although few expert players will agree on which is most important. Moishe Felberbaum, winner of the final round of the 1978 amatuer backgammon championships claimed the game was "75 percent luck". A professional champion, Paul Magriel, strongly disagrees: "Luck is very much over-rated" (from an interview in *Gambling Times* magazine, quoted in *Lady Luck's Companion* (Berger A J and Bruning Nancy, Media Projects Inc., Harper and Row, New York, 1979, p. 61)).

You'll find that, fairly obviously, while the 'luck' part of the game is pretty easy to program, the 'skill' part will demand far more of your programming talents. I remember one early *Backgammon* program for the TRS-80 Color Computer which appeared to me to cheat, more often than not, by throwing itself brilliant dice, while giving me really poor throws. An element of cheating can, I'll admit, make your programs perform better than they really should, but the cheating must be used very sparingly, and in such a way that the human player is not likely to have his or her suspicions aroused.

Cards

Many card games can be computerised, and any game where the ability to remember which cards have been previously dealt (and therefore the chances of particular cards remaining in the deck coming up in subsequent hands) can allow the computer to shine. Get a book of card games, and



you're sure to see several ideas for programs. *Blackjack, Bridge, Poker* and *Gin Rummy* are all worth tackling.

If you feel really ambitious, you could try your hand at *Mah-Jongg,* which is basically a kind of rummy played with colourful tiles instead of cards. A westernised form of the ancient Chinese game of ma-tsiang (sparrows), *Mah-Jongg* became a craze here in the twenties. The tiles form three suits (bamboo, circles and characters); honors (the four winds, and red, green and white dragons) and four seasons or flowers. The terminology is rather fun. You can *rob the kong, go woo, break the wall* or even *create a garden* while dealing with *kongs, pungs* and *chows.*

In contrast to *Mah-Jongg,* simple games like *Bingo* and programs in which you allow a number of human players to take part in an electornic version of a lottery, and simple to program, and can be a lot of fun to run (especially late at night, when the bottle rather than the keyboard has been your main source of interest over the preceding hours).

Whatever the kinds of games you decide to program, you're sure to find there is a special attraction in playing against a program that you have written. And don't just stop once you get a working version of a program up and running. Polish it. Make the inputs and outputs as simple as they can be. Refine the playing algorithms so the program does not constantly fall into the same traps, or lose from a position which often occurs. The chances are you'll eventually decide — as I have — that writing and improving the programs is actually just as much fun as playing them.

Section Six
Creating Adventures

Why would intelligent people, with access to massive amounts of computer power which could be doing something useful, want to bring their brainpower to bear on the problems of creating and exploring mythical landscapes? Why, when your Amstrad could be earning its keep balancing your cheque account, are you forcing it to control a game in which you battle fierce monsters in labyrnthine caverns, envoke magic spells, and uncover vast hordes of Elven gold?

The answer is obvious. Using your computer for serious things all the time is just plain boring. Trekking across alien landscapes, chopping up people with broadswords and axes, haggling with quasihumanoid creatures in off-world taverns, seems to many people a much better way to spend their time and their computer power.

If you want to add this kind of excitement to your Amstrad, this section of the book is designed just for you.

Adventure games — with or without a computer — have not been with us for long. It was just over a decade ago, in 1974, when the fantasy game *Dungeons and Dragons* was released. D & D (as it quickly became known) triggered an explosion of interest in role-playing games.

Fantasy games grew out of war games, which began life in the final years of the 18th Century as training exercises for real-life army commanders. After the First World War, many people took to playing war games as a hobby, reenacting famous battles and rewriting history as they played.

In Lake Geneva, Wisconsin, a war gaming club made a slight departure from the norm, and started an explosion which is still reverberating around the world.

A team from this club, under Gary Gygax, devised a set of war game rules which set the clashes in a mythological, medieval setting where magic worked, dragons flew, and treasure in abundance lay around waiting to be

claimed. The rules were published as *Chainmail.*

This was the first part of the mixture which brought fantasy gaming to critical mass. The second part was provided by Dave Arneson in Minneapolis-St. Paul, Minnesota, who decided to set his battles in an underground dungeon. He used the *Chainmail* rules to manage the clashes his players had with wizards, warlocks and rest. Dave and Gary, it appears, then got together to create a dungeon/game-playing scenario called *Greyhawk.* They codified the rules, and put them together in a little book, and tried to have it published.

None of the established game companies were interested, so in 1974 Gygax formed a company — Tactical Studies Rules (TSR) — to publish the rules himself, under the name *Dungeons and Dragons.* The first printing sold out in less than 12 months.

Dungeons and Dragons fed a widespread hunger which had been primed by the success, in the previous decade, of such books as Tolkien's *Lord of the Rings.* D & D succeeded beyond the developers' wildest expectations, with over a million sets of the rules — which gradually became more coherent and more complex as feedback from players was incorporated into new editions — sold in the next six years. But this was just the start.

It seemed inevitable that computers would get in on the act at some stage. Computer memory is the ideal tool for keeping track of the complicated rules, situations, and inventories that are part of fantasy games.

Back in the dim, dark days of the immediate past — before the era of microcomputers — access to computers was restricted and expensive. The only way you could use a computer was via a time-sharing system. You communicated with a vast, remote computer, via a noisy teletype keyboard. The computer was dealing — apparently simultaneously — with scores, perhaps hundreds, of other users. It talked back to you by printing on the teletype. It was slow, inconvenient, and unless you were lucky to be able to access such a computer from work or college, very expensive. Access charges around $80 to $100 an hour were not unusual. Pretty obviously, this meant that game-playing and game-writing on computers was a pastime very few had the opportunity to pursue.

Two programmers (Will Crowther and Don Woods) got around the expense problem simply by appropriating computer time — which should have been used for such mundane tasks as aiding with the statistical analysis of the results of physics experiments — and using it for their own entertainment.

Working in the language Fortran (a forerunner of the BASIC language on your Amstrad), Crowther and Woods entered into history by writing the program *Adventure.* It was a mammoth work, plunging the player into a vast underground labyrinth which could take months to solve. Hundreds of thousands of man-hours were lost across the country as computer personnel neglected their work in obsessive drives to work their way through the underground labyrinth.

The original *Adventure,* and similar programs which followed it, demanded big, big computer memories. Memory was expensive in those days, and the first microcomputers had amounts which seem niggardly to us now. Scott Adams, an *Adventure* admirer, decided he would make the program fit into a TRS-80. Predictably, scornful laughter greeted his announcement of this grandiose aim.

Fortunately, Scott succeeded, and on that base has built what is probably the world's leading computer adventure software company (Adventure International). The Adams material is available for most popular computers, and its usefulness is enhanced by a "difficulty rating" which is printed on each pack. *Pirate Adventure* is the easiest. The difficulty ratings go from beginner to medium difficulty and end up with the daunting "Advanced Adventures Only Need Apply."

Once Adams had shown it was possible to squeeze an adventure into the mini-memory of a microcomputer, many others followed. Groups of users formed clubs to share ideas on solving problems.

Some of the most fanatical of these groups are those involved in the adventure program *Zork* (and, more recently, *Zork II* and *Zork III*), produced originally by a computer group at the Massachusetts Institute of Technology.

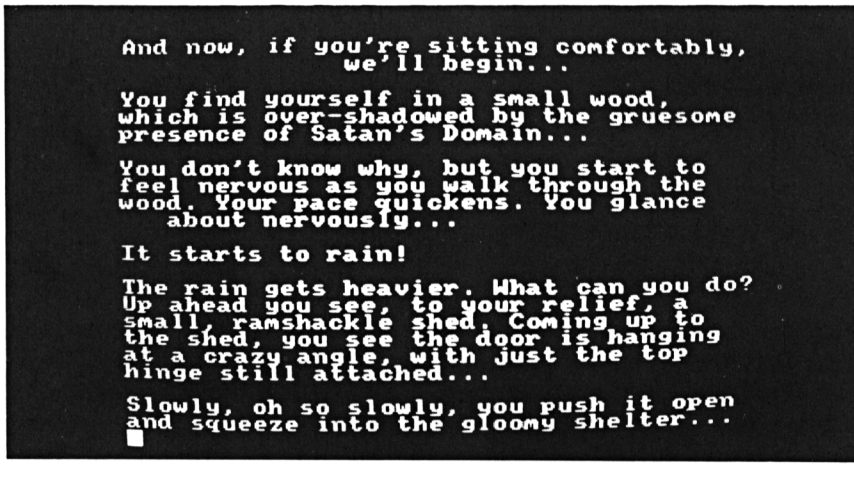
Most adventure games (including those in this book) invite the player to either enter a single letter (such as "N" meaning the player wishes to move north) or a two-word verb/noun phrase (such as SLAY DRAGON) to tell the computer what to do next. *Zork* and many other recent adventures accept multiple-statement inputs such as GET THE AXE AND DROP THE LAMP THEN GO NORTH. The *Zork* group publishes hint sheets and maps, and even bumper stickers *HONK IF YOU LOVE ZORK?).*

There are several "classic" concepts which lie behind the development of adventure programs. I've tried to incorporate all of them into this book, so

the ideas you pick up here can easily be applied to adventures from other sources.

First, a computer adventure is, in essence, a puzzle or series of puzzles. The puzzles are invented by the programmer, and must be solved by the player. The original *Adventure* and the Adams products are good examples of this.

When playing a computer adventure you must first discover the vocabulary the program will accept and understand, and then use this to explore your environment. Early adventures made much of battling monsters, gathering treasure and solving odd puzzles, such as trying to guess the code word which the "troll guarding the twisty bridge" would accept to let you pass.



The puzzles in my programs are far simpler than those you'll meet in commercial adventures, but serve as good training ground for your later explorations.

Computer adventures soon learned that stable environments produced far more satisfactory adventures than did those in which the relationship between rooms, and the scenery, changed at the whim of the computer's random number generator. By stable, I mean an environment which has an inner consistency, and which — most importantly — can be mapped.

There is an immense amount of satisfaction to be gained from deducing the layout of the land in which you find yourself, and then in "wandering

around it," checking out your map. All the environments in this book can be mapped.

The techniques discussed in this book for adventure development will not only help you create your own adventures, but should give you a number of ideas to help you solve adventures written by others.

Many educators bemoan the fact that little computer software of a truly educational nature has been developed. I believe that there is much educational value in the development and solving of adventure programs. Writing in the magazine *Popular Computing* (December 1983, pp. 147-150), Dr. Peter Owens, who is an English professor at South-Eastern Massachusetts University, says that he has found that adventure games "satisfy a couple of requirements that give them validity as educational exercises." He then cites the engagement of thinking skills and the fact that the challenge of using imaginative approaches to problems can teach techniques which can be applied to real-life experiences.

So, there is an added bonus to adventure gaming. I think now it is time we buckled on our armour, conjured up a few soul-protection spells, and went down into the dungeons.

Give Me My Bow

Let's face it. Life can be pretty tame, sometimes. There don't seem to be many dragons waiting to be slain in my city, and chests heaped high with abandoned gold are in scarce supply. I can't remember the last time I met an Evil Magician down at the local supermarket, and it's been ages since I discussed battle tactics with sentient androids at the local tavern.

The hunger for excitement lies in all of us. The desire to take on the personalities of other, more vibrant, people — even for just an hour or so — is a common one. Although you can't conjure up devils and werebears, envoke the power of a Shield of Protection, or employ trolls to carry sacks of emeralds from the ruins of an abandoned castle, role-playing games allow you to do just that.

Adventure gaming has hit the big time. You've probably seen the claims that it is the "fastest growing game in the world." Whether that's true or not, it indicates that Adventure gaming is a leisure pursuit which satisfies the inner needs of many people.

But these real-life campaigns have one enormous disadvantage. You need people to play with and against. You need a referee (often called the Game Master, or Dungeon Master) to control the world and its artifacts and encounters. It is not always particularly easy to get all these other human beings together just when you decide you'd like to indulge in a little bit of Adventuring.

That's where the computer comes into its own.

Although computer Adventure games lack a little of the spontaneity of games played with live company, they can be remarkably unpredictable and exciting to play. The fact that the Hydra of 10 Heads you've just slain exists only within your Amstrad's RAM seems in no way to diminish the relief you feel when it dies. The gems you find lying all over the place are no less "real" than those discovered in live-action Adventures.

How to Read This Section of the Book

I've written this section to show you just how easy it is to create computer Adventure games of your own. However, there is one problem, and I hope you'll be willing to work with me to solve this problem. It is pretty difficult to know where to begin explaining how a computer Adventure is structured. Many times I've discovered that understanding one particular programming concept depends on your already understanding a second, separate concept. I've done all I can to make sure that the introduction of these concepts follows a more or less logical order and that all new concepts are carfully explained. Unfortunately, because of the complexity of most Adventure game programs, from time to time this has been impossible. All I can do is ask you to proceed on trust. Explanations which are not blindingly clear the first time you read them should swiftly fall into place as you continue working through this section.

I've written this section with the ancient Chinese maxim — A PROGRAM IS WORTH A THOUSAND WORDS — always in mind. You can learn far more by entering a program, or program fragment, and then running it, than you can from chapter after chapter of explanation. Therefore, this section program-oriented. It contains two major programs (plus variations), and the instruction part of the book is based on these programs. In fact, if you just want some Adventure programs to run, you can just enter and run the programs as they are, ignoring the lucid explanations which surround them.

However, as is obvious, if you do this you'll miss the whole point of the

exercise. Move slowly, have your Amstrad running when you read, and enter each piece of code as you come to it, and you'll discover that in a very short while from now, you'll be creating Adventure programs of your own.

The Scenario

This Adventure, WEREWOLVES AND WANDERER (where you are the wanderer, and the werewolves and other unsavory creatures inhabit the imaginary environment you are about to visit), takes place within an ancient, abandoned, stone castle.

You read about the castle in a faded letter you found in a trunk left to you by your grandfather. Unfortunately, you could only find the second page of the letter, so you are not sure of the full story. However, from the page remaining, you understand the castle was abandoned centuries ago after a curse was placed on the inhabitants by an old witch who had been evicted during a particularly harsh thunderstorm. The king's wife was ailing, and he (wrongly, as it turned out) blamed the witch for his wife's illness. He thought by throwing the witch out, her malignant influence on his wife would cease. This did not happen, and his wife became more and more ill, and finally died.

Her last days were not peaceful. The old woman's curse brought a reign of terror to the castle, as many odd creatures and ghosts took up residence within the castle. Finally, the king and his court could stand it no more, and they fled from their former home, never to be heard of again.

The creatures invoked by the witch stayed on, and are living there still. You are about to enter their realm.

Before we look at our listing in detail, spelling out each element of its development, I'll show you some "snapshots" of it in action. This should help you make sense of the various elements of the program as they are presented.

First the program asks your name, which it will use from time to time throughout the game:

```
WHAT IS YOUR NAME, EXPLORER? ANDREW
```

The computer then proceeds to tell you — as it will after each action you initiate — your current status:

```
ANDREW, YOUR STRENGTH IS 95
YOU HAVE $ 75
IT IS TOO DARK TO SEE ANYTHING

WHAT DO YOU WANT TO DO? I
```

The question "WHAT DO YOU WANT TO DO" is asked after every move. The answers the computer understands will be explained a little later under the heading "Vocabulary." Note here that our hero, Andrew, has entered, "I", which stands for "inventory." This means he wants to add to his supplies of provisions.

The computer goes to the relevant subroutine where Andrew's money is shown ("YOU HAVE $75") and then the list of available products and their prices is shown:

```
PROVISIONS & INVENTORY


YOU HAVE $ 75


YOU CAN BUY 1 - FLAMING TORCH [$15]
            2 - AXE [$10]
            3 - SWORD [$20]
            4 - FOOD [$2 PER UNIT]
            5 - MAGIC AMULET [$30]
            6 - SUIT OF ARMOR [$50]
            0 - TO CONTINUE ADVENTURE
ENTER NO. OF ITEM REQUIRED? 1
```

Andrew enters a "1", telling the computer he wants to buy a "flaming torch" (item 1 on the inventory menu). The torch is added to his possessions, his wealth is decreased by $15, the cost of the torch, and the menu option is shown again:

```
ENTER NO. OF ITEM REQUIRED? 4
HOW MANY UNITS OF FOOD? 20
```

This time, Andrew has chosen item 4 on the menu, signifying that he wants to buy food. Food is a very valuable possession. You start the game with a limited amount of strength, which is gradually consumed as the game goes on. If it drops to zero, you die, and the game ends. Fights against wild monsters also diminish your strength. Eating food replenishes it. You have to keep an eye on your current "strength rating" throughout the game, and always make sure you have food on hand to replenish your rating when needed.

After entering item 4 on the menu, Andrew is asked how much food he wants to buy, with each unit of food costing $2. After this purchase has been made (Andrew buys 20 units), he enters a 0 next time the menu is presented, indicating that he wishes to continue with the Adventure:

```
YOU HAVE $ 20

YOU CAN BUY 1 - FLAMING TORCH ($15)
            2 - AXE ($10)
            3 - SWORD ($20)
            4 - FOOD ($2 PER UNIT)
            5 - MAGIC AMULET ($30)
            6 - SUIT OF ARMOR ($50)
            0 - TO CONTINUE ADVENTURE
ENTER NO. OF ITEM REQUIRED? 0
```

Another status report is given, showing Andrew's strength and wealth, and the fact that his provisions sack holds 20 units of food. The location is descrived, and Andrew enters "E" (meaning he wants to move east) when asked what he wants to do next:

```
ANDREW, YOUR STRENGTH IS 90
YOU HAVE $ 20
YOUR PROVISIONS SACK HOLDS 20 UNITS OF FOOD

YOU ARE AT THE ENTRANCE TO A FORBIDDING-LOOKING
STONE CASTLE.  YOU ARE FACING EAST


WHAT DO YOU WANT TO DO? E
```

He is moved eastwards, and another status report is given. Notice that his strength rating has been reduced slightly:

```
ANDREW  YOUR STRENGTH IS 85
YOU HAVE $ 20
YOUR PROVISIONS SACK HOLDS 20 UNITS OF FOOD

******************************


YOU ARE IN THE HALLWAY
THERE IS A DOOR TO THE SOUTH
THROUGH WINDOWS TO THE NORTH YOU CAN SEE A SECRET
HERB GARDEN
THERE IS TREASURE HERE WORTH $ 100


WHAT DO YOU WANT TO DO? P
```

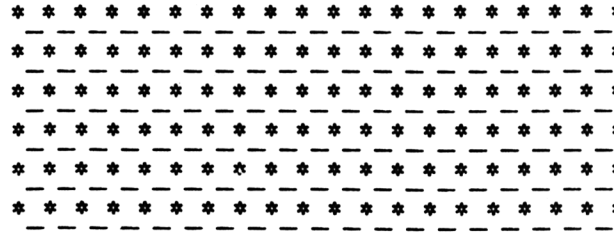
He sees that there is $100 worth of treasure here, so enters "P" (for "pick up the treasure") and when the status report comes back, the extra wealth has been added:

```
ANDREW, YOUR STRENGTH IS 80
YOU HAVE $ 120
YOUR PROVISIONS SACK HOLDS 20 UNITS OF FOOD

******************************


YOU ARE IN THE HALLWAY
THERE IS A DOOR TO THE SOUTH
THROUGH WINDOWS TO THE NORTH YOU CAN SEE A SECRET
HERB GARDEN


WHAT DO YOU WANT TO DO? S
```

Moving south (he entered "S"), Andrew finds himself in the Audience Chamber. But he is not alone! His first battle is only moments away:

```
THIS IS THE AUDIENCE CHAMBER
THERE IS A WINDOW TO THE WEST. BY LOOKING TO THE
RIGHT
THROUGH IT YOU CAN SEE THE ENTRANCE TO THE CASTLE.
DOORS LEAVE THIS ROOM TO THE NORTH, EAST AND SOUTH

     DANGER...THERE IS A MONSTER HERE....

     IT IS A DEVASTATING ICE-DRAGON

     THE DANGER LEVEL IS 20 !!

     WHAT DO YOU WANT TO DO? F
```

Andrew has entered "F" (for "fight") when asked what he wants to do. He could have entered "R" (for "run") if he felt cowardly. However, the computer would not necessarily have accepted this, and may have told him "NO, YOU MUST STAND AND FIGHT". However, Andrew has decided not to try and run, but instead to battle the Ice-Dragon:



```
YOU HAVE NO WEAPONS
YOU MUST FIGHT WITH BARE HANDS


YOU ATTACK
```

```
THE MONSTER WOUNDS YOU!

YOU ATTACK

YOU MANAGE TO WOUND IT

THE MONSTER WOUNDS YOU!
YOU ATTACK

AND YOU MANAGED TO KILL THE DEVASTATING ICE-DRAGON
```

His strength is failing, so he decides he must eat, so enters a "C" (for "consume"):

```
ANDREW, YOUR STRENGTH IS 65
YOU HAVE $ 120
YOUR PROVISIONS SACK HOLDS 20 UNITS OF FOOD

WHAT DO YOU WANT TO DO? C

YOU HAVE 20 UNITS OF FOOD
HOW MANY DO YOU WANT TO EAT? 15

ANDREW, YOUR STRENGTH IS 125
YOU HAVE $ 120
YOUR PROVISIONS SACK HOLDS 5 UNITS OF FOOD
```

And so the Adventure unfolds:

```
THIS ROOM WAS USED AS THE CASTLE TREASURY IN
BY-GONE YEARS....
```

```
ANDREW, YOUR STRENGTH IS 115
YOU HAVE $ 279
YOUR PROVISIONS SACK HOLDS 5 UNITS OF FOOD

******************************


THIS IS THE SMALL ROOM OUTSIDE THE CASTLE
LIFT WHICH CAN BE ENTERED BY A DOOR TO THE NORTH
ANOTHER DOOR LEADS TO THE WEST. YOU CAN SEE
THE LAKE THROUGH THE SOUTHERN WINDOWS


WHAT DO YOU WANT TO DO? N

YOU HAVE ENTERED THE LIFT...
IT SLOWLY DESCENDS...
```

These "snapshots" from the game indicate clearly how the game behaves when the program is up and running. I hope you are now straining at the bit to get your first Adventure up and running.

Creating the Floor Plan

We will now examine the critical problem of how to get your computer to keep track of the complicated "floor plan" or "map" used in an Adventure game. We'll take a careful look at how to write a program that keeps track of a simple five-room environment. Note that the program sections in this chapter refer only to this simplified environment, and are *not* part of WEREWOLVES AND WANDERER. Follow through the explanations and examples carefully. The concepts outlined here will later be expanded for use in our first full-scale Adventure program, WEREWOLVES AND WANDERER.

Mapping the Environment

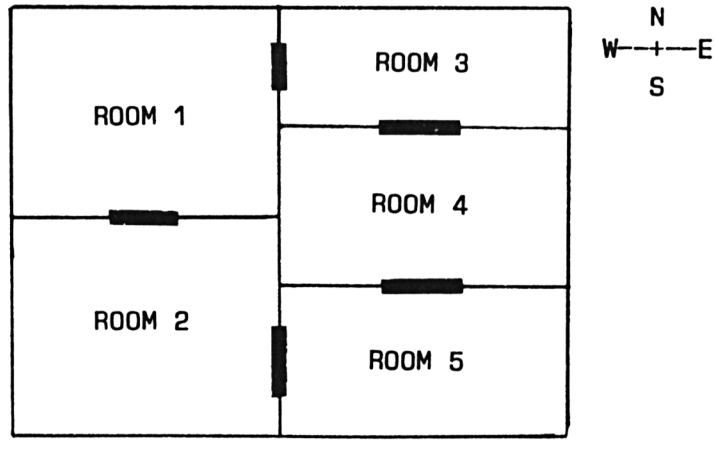
As was pointed out earlier, an Adventure environment must be coherent. That is, the explorer making his way through the environment must be able to draw up a complete map as he works his way through it. If he draws

a door connecting the study with the library on an environment floor plan, because he has discovered that going through the study door leads into the library, he is entitled to expect that turning around and going back will bring him back into the study. The game-player should be able to build up an entire plan in this way, checking his plan from time to time by "walking around" the house, castle, forest, underground labyrinth, or whatever where the Adventure is taking place.

The first step, then, in building an Adventure program is to construct an environment which can be both mapped, and represented in some way which the computer can store.

You'll be pleased to know it is relatively easy to satisfy both these conditions.

Look at the following five-room environment, a very simple one, which we shall treat as though it was a computer Adventure environment.



The key to holding an environment like this in a way the computer can understand and manipulate is to set up an array, each element of which represents a room. The solid markers between rooms are doors.

If you were in room one, you could move east into room three, or south into room two. In room four you can move north into room three and south into room five, and so on. Imagine we have set up an array, which we have dimensioned as DIM A(5, 4). The first dimension is the room, and the second one is the four possible directions from that room (that is, north, south, east, and west).

Building a Travel Table

Armed with the map of the five-room environment, we can now build up a Travel Table, which can then be fed into the array, to allow us to move from point to point within the environment.

Here's the Travel Table for the simple, five-room environment:

ROOM	N	S	E	W
1	0	2	3	0
2	1	0	5	0
3	0	4	0	1
4	3	5	0	0
5	4	0	0	2

Take some time to study this table, and the way it relates to the map, because it is the single most important key to building Adventure programs you will learn from this book.

Look at the table for room one. Under the "N" (for north) column we see a zero, meaning you cannot move north from room one (a fact which is easily verified by looking at our map). However, under the "S" we see the number two, meaning that if we traveled south from room one we would end up in room two (again, you can verify this from the map). Move east (the "E" column, of course) from room one, and you'll end up in room three. The 0 in the "W" column means there is no travel possible west from room one.

You can work right through the table, if you like, checking that the numbers on it correspond to the reality" of the map.

Now, to allow the player to move around the environment, we only need to (a) fill each element of the array with the relevant information from the Travel Table; (b) tell the player where he or she is; and (c) allow the decisions entered by the player regarding the direction he or she wants to move to be checked against the array, and then — if possible — updated to relfect the player's new location. It is easier to do this than you might think.

Moving About

Firstly, we need to write a small program to feed the relevant information into the array. Two simple READ/DATA loops like the following will do it:

```
10  DIM A(5,4)
20  FOR B = 1 TO 5
30  FOR C = 1 TO 4
40  READ A(B,C)
50  NEXT C
60  NEXT B
70  DATA 0,2,3,0
80  DATA 1,0,5,0
90  DATA 0,4,0,1
100 DATA 3,5,0,0
110 DATA 4,0,0,2
```

As you can see, the DATA statements correspond exactly with the items in our Travel Table.

This will fill the A array as follows:

```
A(1,1) = 0
A(1,2) = 2
A(1,3) = 3
A(1,4) = 4

A(2,1) = 1
A(2,2) = 0
A(2,3) = 5
A(2,4) = 0

A(3,1) = 0
A(3,2) = 4
A(3,3) = 0
A(3,4) = 1

A(4,1) = 3
A(4,2) = 5
A(4,3) = 0
A(4,4) = 0

A(5,1) = 4
A(5,2) = 0
A(5,3) = 0
A(5,4) = 2
```

Looking at these, we can see that the first number which comes after the A

is the room number, and the second is the room the player ends up in if he or she moves in a direction where the first direction is north, the second is south, the third of course is east, and the final direction is west. This is all suggested by our Travel Table. I hope by now you are beginning to appreciate just how central the Travel Table is to construction and manipulating an artificial environment.

The Player's Location

If we decided that the room the player is currently occupying could be designated by the variable RO, we could tell the player where he or she was as follows, as well as indicating which exits existed:

```
100  PRINT "YOU ARE IN ROOM NUMBER ";RO
110  IF A(RO,1) <> 0 THEN PRINT "A DOOR LEADS
     NORTH"
120  IF A(RO,2) <> 0 THEN PRINT "THERE IS AN
     EXIT TO THE SOUTH"
130  IF A(RO,3) <> 0 THEN PRINT "YOU CAN LEAVE
     VIA THE EAST EXIT"
140  IF A(RO,4) <> 0 THEN PRINT "A DOORWAY OPENS
     TO THE WEST"
```

The player's input could be a single letter ("N" for north, and so on) and the program could look at the input, and check to see if an exit to that direction existed:

```
150  INPUT "WHICH DIRECTION";D$
160  IF D$ = "N" AND A(RO,1) = 0 THEN PRINT
     "YOU CANNOT MOVE THAT WAY"
170  IF D$ = "S" AND A(RO,2) = 0 THEN PRINT
     "YOU CAN'T WALK THROUGH WALLS"
180  IF D$ = "E" AND A(RO,3) = 0 THEN PRINT
     "TRY ANOTHER DIRECTION"
190  IF D$ = "W" AND A(RO,4) = 0 THEN PRINT
     "THERE IS NO DOOR TO THE WEST"
```

As you'll see shortly, exactly this system is used in our Adventure programs.

So you can see how the array, constructed using the Travel Table, can

indicate where the player may or may not move. What about the movement itself?

Moving Around the Map

The "you can't move that way" tests, of lines 160 to 190, could trigger a repeat of the input question, "WHICH DIRECTION?," over and over again until a valid direction was entered. Then the movement itself could take place. (Note, by the way, that the room numbers are never referred to explicitly, as they are for the computer's internal use only; all the player hears about are names of rooms — YOU ARE IN THE COUNTING HOUSE — and, in many cases, the contents of the room, and possible exits from it.)

If the player entered "N" (for north), as he or she was in room four (look to the map, to see what this would mean) the computer could proceed as follows. Firstly, the variable RO would equal 4. A short routine could convert the "N" into a 1 (for the first element of the array), so the computer would know the player was about to go into the room number A(RO,1). As RO equals 4, A(4,1) gives 3. This means the player would be moving into room number 3 (which, as a quick check of the map will reveal, is north of room 4). The variable RO would be set equal to three, so the player could be told "YOU ARE IN ROOM 3." The next choice of direction could then be entered, and so the player would continue happily tripping through the environment.

Consistency and Reality

Although the rooms only exist on paper and in elements in an array, the fact that they behave like "real rooms" soon allows them to be perceived as though they were solid and real in a way which is uncanny. Add descriptions of each room — YOU ARE IN A SMALL WORKMAN'S HUT ATTACHED TO THE BACK OF THE MANOR HOUSE, WITH A PILE OF STRAW OVER IN THE FAR CORNER, AND A SHOVEL AND AN AXE LYING UNDERNEATH THE WINDOW. A LARGE LOAF OF BREAD IS ON THE TABLE, AND BESIDE IT IS A NOTE. DOORS LEAVE TO THE NORTH AND TO THE WEST — and you'll find the environment takes on quite solid dimensions in your mind.

Once the map becomes more complex, as in WEREWOLVES AND WANDERER, and the descriptions help clarify the mental images of the rooms, you'll find you have a counterfeit reality with immense power in your hands.

You might like to try and write a simple program, before proceeding further, which allows you to move around the five-room environment we've been looking at.

Creating the Structure

We'll move now from considering the simple, five-room environment, to the much more complex (19 rooms in all) castle environment you will inhabit when taking the part of the Wanderer in WEREWOLVES AND WANDERER. Let's begin with an explanation of how such a complicated program is designed.

The Master Loop

This Adventure program is structured according to a well-defined outline, which was drawn up long before the castle floor plan was created, or any of the incidents which would occur during the playing of the game were considered. Working to an outline like this, in which the actions the program must take are determined before any attention is given to the actual form or coding of the program, is often called "Structured Programming" (for more on this, see pages 349 to 355).

The program sits within a "master loop" which calls all the needed subroutines, then checks to see if the game is over (because the player has reached the exit, or is dead). If this check is negative (that is, the player is not dead, and the final exit has not been found), the program loops back to go through it all again. This cycle continues until one of the "end condition" tests proves positive.

Modular Construction

The program was written in a series of discrete modules, a process I strongly suggest you follow. It will help you keep a long and complex program under control, when a less disciplined approach would make the task almost impossible. A program which is constructed in modules is also much easier to modify (as you 'll see when we look at the more elaborate form of WEREWOLVES AND WANDERER) and debug. To make it easier to keep track of the separate modules, they are divided by a REM statement full of hyphens, so the separate modules are immediately visible when you look at the listing as a whole.

The program begins, then, with these four lines:

```
10 REM WEREWOLVES AND WANDERER
20 GOSUB 2600:REM INTIALISE
30 GOSUB 160
40 IF RO<>11 THEN 30
```

If RO equals 11, the game is over, as room eleven is the final exit from the castle. As you can see, line 20 calls the subroutine which starts at line 2600, the initialization subroutine. Line 30 calls the Major Handling Routine, and if the check in line 40 proves negative, goes back to 30 to call this routine again.

As I said, line 20 sends action to the initializing subroutine which begins at line 2600. These are the lines which fill the array with the numbers which represent the directions the player can move from various rooms.

The seventh element for each room is used to hold either treasure or terror, as will be explained in next part.

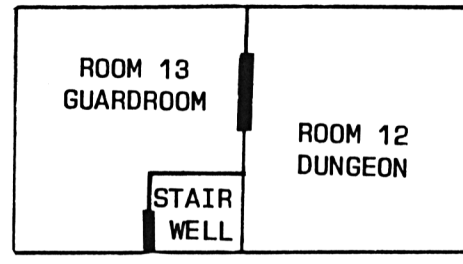
In this game, there are six possible directions, the four compass points as well as up and down. The castle has three levels, and you are able to move from one to the next via a circular stairwell, and "an ancient hydraulic life." The numbers which follow the DATA statements (lines 3310 to 3490) represent, for each room: north, south, east, west, up and down.

The Maps

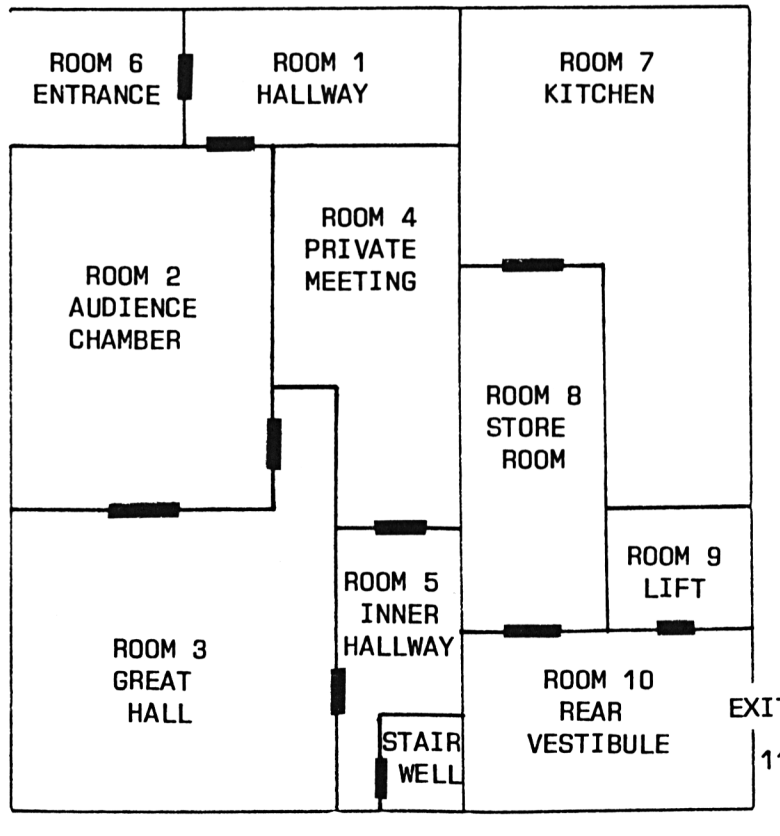
We must see now how they relate to the floor plan of the castle. Of course, when you give this game to somebody to play you do not provide them with a map. Part of the pleasure of playing Adventure is constructing a map of the environment which will give you control of it. "Walking through" and environment when you have mapped it, to check your cartographic work, is also part of the fun. The map is only given here so you can see how the program relates to it.

There are, as I said, three levels. The basement is made up of a guard room, and a dungeon, like this:

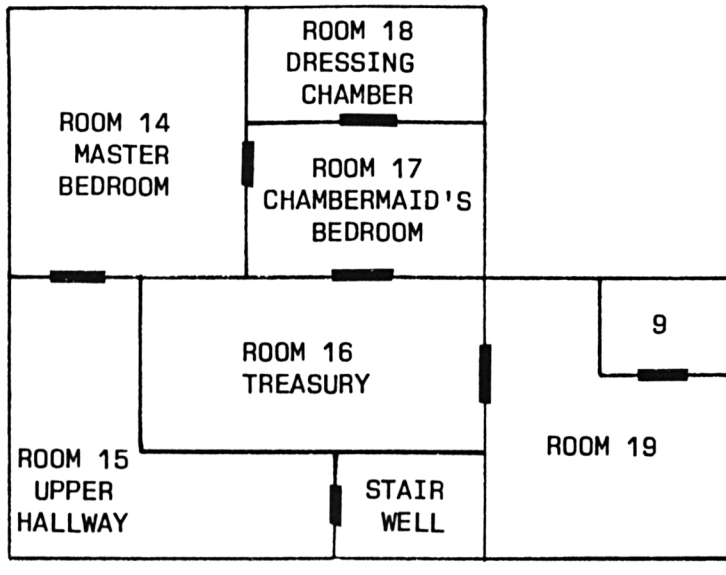
BELOW GROUND:



The ground floor, where you begin your quest, looks like this:



And the upper floor is built as follows:



You'll see that it is impossible to go directly from the entrance ("room 6": the entrance must have a room number, as must the exit) to the exit ("room 11") without going upstairs. There is no direct path from the west side of the castle to the eastern one. This ensures that the player is forced to do quite a bit of exploring before being able to locate the exit. It also helps to confuse the map a bit. Constructing a three-level map of the castle may take many hours, as you're sure to appreciate shortly.

Treasure and Terror

Now that we have created a section of program to handle the physical environment of the castle in a form which we can manipulate, we need to fill some of the rooms, so the player will have treasure to find and monsters to battle.

Distributing the Treasure

This is easy to do. Let's look at adding treasure for a start. Lines 2860 and 2870 choose a room number between 1 and 19, check that it is not 6 (the entrance, outside the castle) or 11 (the exit, outside), and then check that

this room is empty. You will recall we mentioned in the last chapter that the seventh element for each room was left blank to allow for contents. Now we are filling it. If this element is zero, the room is currently empty.

Once a room has been chose with line 2860, line 2880 puts an amount of money in it between $10 and $109.

Distributing the Monsters

With the treasure in place, it is time to allot our terrors. The next routine does this, choosing a room in the same way as our money routine did, then setting that element equal to a negative number (-1 to -4). You can see now that if the computer looks at the seventh element of a room, it can tell if it is empty (equals 0), contains treasure (greater than 9) or a monster (less than 1) and can give the player a message to this effect. Extending the use of the room array in this way points up again how vital the array is to the construction and control of an Adventure environment.

Finally, we put treasure in two rooms (the Private Meeting Room and the Treasury) regardless of whether the above routines have already put anything there. These lines wipe over anything else which has been placed there.

Adding Variables

A number of variables are needed to add to the creation of a realistic Adventure environment. I make it a practice to use explicit variable names which will remind me (or tell me straight out) what the variable stands for.

The first line in our next section clears the screen, before the following variables are initialized: STRENGTH, WEALTH, FOOD, TALLY, MK. If STRENGTH ever gets to zero, the game is over. WEALTH can be used to buy FOOD which can be eaten to restore STRENGTH. TALLY is a simple count of how long you have survived within the environment (one is added to TALLY each time you move) and MK is the count of monsters killed.

The Adventure is heightened if the computer uses the player's name from time to time, so line 2750 asks for the name, and allots N$ (for "Name string") to this. Line 2760 clears the screen of the question and answer, ready for the game itself to get underway.

The game begins outside the castle, in what we've designated "room 6," so the variable RO (for "ROom") is set equal to 6 in line 2770.

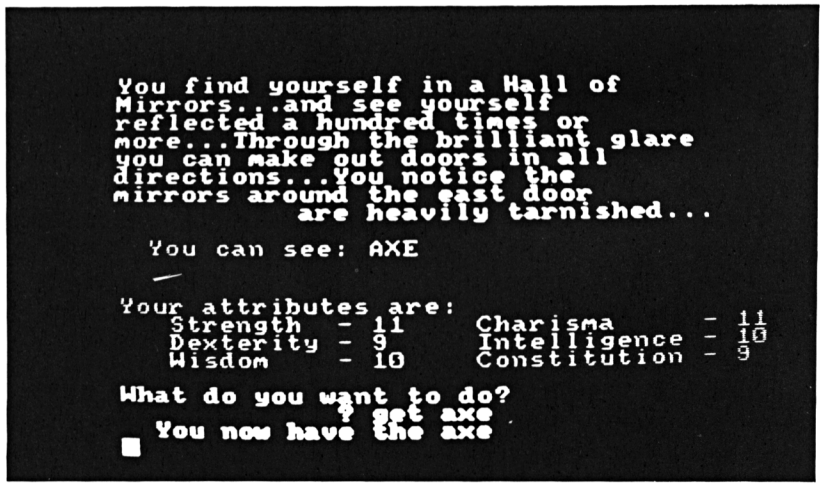
There are a number of things the player can manipulate or wear during the game, and the variable names to show possession of these are set to zero in lines 2780 to 2820. If any of these is changed to equal one, the player is carrying or wearing the object. This makes the purchase of, say, an axe very easy. If the variable AXE equals zero, the player does not have an axe. If AXE becomes equal to one, the computer can print out "YOU ARE CARRYING AN AXE."

It is time now to investigate the section of program which lies at the heart of the entire Adventure, the Major Handling Routine.

A Short Delay

Before we look at that, however, I want to show you a very short section of program, really only one line, which is very important to pacing the running of the program.

In contrast with many Adventure programs which allow a large vocabulary, and look at two or more words at a time, our WEREWOLVES



AND WANDERER only responds to a limited vocabulary. Because of this, it reacts very rapidly to input from the player. In many cases, this

reaction is so fast there is not even time to read what is on the screen before it scrolls off.

Therefore, a delay loop has been included within the program, at line 3520. It runs from 1 to 900 at the moment, which seems ideal to me. You'll find this is called over and over again throughout the program, not only to give you a chance to read information on the screen before it vanishes, but to pace such things as a fight with a monster, or the descent of the elevator.

Your Strength Fades

The game is over if you run out of strength, as was pointed out earlier. Therefore, you have to keep an eye of your strength rating, which diminishes throughout the game. You replenish your strength by eating food, which you can buy with the proceeds of treasure you find lying around the castle.

The first part of the Major Handling Routine subtracts five from your strength (variable name STRENGTH) each time you move.

Line 180 warns you that your strength is getting low if STRENGTH falls below 10. You will discover later that your strength can also fall during a fight. Sometimes you'll see this drops quite drastically if you are wounded. If your strength falls below one (line 190), the program directs you to the Dead End subroutine from line 2290.

When you construct your own programs, you'll discover that — within limits — the more things the player has to manage at a time, the more satisfying the program is to run. You should therefore aim to have something like our STRENGTH problem in your own programs, and a clear penalty, like our Dead End routine, if the problem is not handled satisfactorily.

The Final Reckoning

The final score in WEREWOLVES AND WANDERER is related — as was pointed out earlier — to a number of factors. One of these is TALLY, the time for which you manage to survive. Each time the program traverses the rocky ground of the Major Handling Routine, it runs through line 200, which increments your TALLY by one.

At the very end of the program, whether you've managed to find the exit

or not, your score will be added up. Your score is three times your TALLY, plus five times your STRENGTH, plus two times your WEALTH, plus the value of your uneaten FOOD and thirty times the number of monsters you have killed (variable name MK). It is evident that the number of monsters killed has more effect on your final score than just about any other item.

There is an infinite number of ways that final score can be calculated in an Adventure game. This is done by a simple scoring system in WEREWOLVES AND WANDERER, one you may well wish to modify. So long as your own games have some method of assessing the success or otherwise of a trek through your environment, it doesn't really matter what sort of final tallying system you devise. It can be as elaborate, or simple, as you choose.

Maintaining the Status Quo

As you saw in the sample run before we started outlining this program, the player gets a Status Report each time the program cycles through the Major Handling Routine.

The Status Report is handled by the lines from 210.

Your name is used in line 210 (N$, for "name string") before you are told your STRENGTH and then WEALTH (line 220). If the variable FOOD is greater than zero (line 230) the computer knows you have bought some food which you have not yet eaten, and line 230 prints up a message to this effect. The same goes for the suit of armor (line 240), the axe (270), sword (280), and magic amulet (300). The other lines (those which print up such things as YOU ARE CARRING and the word AND) also use the value of the variables AXE, SWORD, and AMULET to determine when these bridging words should be printed.

Let There Be Light

The presence, or otherwise, of a torch is crucial to many Adventure programs, including this one. The variable LIGHT is set initially to zero, meaning the player has not bought a "flaming torch".

If the player has no torch, he or she can see nothing. It is very dark within our castle. Lines 320 and 330 make up our Light/Dark Routine. If the player does not have a flaming torch (that is, variable LIGHT equals zero)

the routine prints up IT IS TOO DARK TO SEE ANYTHING, and that's about it. If, however, you are carrying a torch (that is, LIGHT equals one) line 330 sends the computer to the vast subroutine from line 990 which describes what you can see in the room.

Later on in the program, LIGHT is brought into play again. In the Pick Up Treasure Routine, the first line checks to see if you have a torch. If you do not, you are told curtly YOU CANNOT SEE WHERE IT IS, and are not given the chance to get it.

Objects in Disarray

You'll recall that the seventh element of a room held in the A array is set equal to the contents of the room. If this element is zero, there is nothing within the room the player can interact with.

The Object Handler depends totally upon this seventh element for its actions. line 340 sets the variable K equal to the value of this element. Line 350 checks the value of K, knowing that if K equals zero, the room is effectively empty. If K is greater than nine (see line 360) the computer knows there is treasure in the room. If the room is not empty (K < 0) and does not contain treasure (K > 9) then the room must have a monster in it, and line 370 breaks the happy news.

If there is a monster in the room (that is, if K is less than 0) the string variables M$ (for "monster string") is assigned to the name of one of the monsters. The variable FF (for "ferocity factor") is also assigned. The higher the FF, the lower the chance you have of winning a battle against the relevant monster.

However, as you'll see when we look at The Battle later on, the weapons you hold, and whether or not you are wearing armour, can modify the ferocity factor in your favour.

The next section of the program (lines 380 through to 440) assign the monster's name to M$, the ferocity factor to FF, and tells you all about it.

You can see how the "major handling routine" contributes another link in the chain which forges a feeling of reality in the Adventure environment. Any player action — such as picking up the torch — which produces a change within the Adventure, helps build mental images which sustain the illusion of the Adventure.

A Limited Vocabulary

WEREWOLVES AND WANDERER recognizes a strictly limited vocabulary. While this diminishes the realism of the Adventure to some extent (if such an arcane environment and such possibilities of interaction could ever be said to be "realistic"), it simplifies the construction of the program, and ensures that it runs as quickly as possible.

The program accepts your commands in lines 450 to 470, prompting you with WHAT DO YOU WANT TO DO?

You can enter the full word (such as NORTH) or just the first letter. As you can see, the last part of line 460 cuts the input down to a single letter anyway, so you may as well just enter the first letter of your command.

Here is the vocabulary the program recognizes:

N—NORTH **S—SOUTH**
E—EAST **W—WEST**
U—UP **D—DOWN**
F—FIGHT **R—RUN**
M—MAGIC (AMULET, used to trigger the amulet, which moves you to a room selected at random within the castle)
I—INVENTORY (to add to your possessions, and used if you want, for example, to buy food or an axe)
Q—QUIT (to terminate the Adventure at any time)
P—PICK UP (treasure)

Once you've entered your command, and it has been reduced (if necessary) to a single letter, the program draws a line across the screen to help organize the information on it.

If you have entered "Q" (for "quit"), the program goes straight to line 120 where your final score is calculated and printed.

As was explained earlier, the computer knows from the contents of the A array which directions are valid for travel from the current room. You'll see here that a different message is triggered fro an attempt to move in each direction. This gives the impression that the comaputer is choosing from a vast set of possible replies, rather than just saying each time "YOU CAN'T GO THAT WAY."

Fight or Flight

When confronted with a monster, you can enter eight "F" (if you want to fight) or "R" (if you want to run). But, because we do not want to encourage cowardice, you'll discover that around 70% of the time you attempt to run, the computer will tell you it is impossible.

Line 560 is used to generate a random number between zero and one if the "R" option is entered, and if this random number is greater than .7, the program jumps to the Unsuccessful Attempt To Run routine. When it hits the routine, the computer tells you NO, YOU MUST STAND AND FIGHT and then changes your order to "F" and then returns to the field of action. If, however, the random number generated is not greater than .7, you are asked which direction you wish to run in.

There is one more possibility. A player may enter "F" when there is no monster in the room. A badly written program may well then obligingly produce a monster out of nowhere. WEREWOLVES AND WANDERER gets around this by checking to see, when an "F" command has been entered, whether the vital seventh element is less than zero, and if it is not, points out THERE IS NOTHING TO FIGHT HERE and then goes back to 440 for a new command:

You'll recall how, at the start of the book, I mentioned that some commercial Adventures, such as *Zork*, allow for whole sentences to be entered by the player. our single-letter inputs seem pretty tame compared to this. But, as you'll see in our final program — THE SATAN BLINDER — which accepts two-word commands, the way the program processes the single-letter input in this program forms the basis of the mechanism used for two, or more, word inputs.

Other Tongues

The routine from lines 590 to 710 acts on the other possible inputs. You'll see (line 600) that if the player has entered "C" (for "consume," indicating a desire to eat), a check is made to see if there is any available food. If the variable FOOD equals zero, you are told YOU HAVE NO FOOD, and the computer returns to 440 for a new command.

The lines 650 through to 700 move you within the castle environment, as was explained earlier.

Making Magic

If you enter "M" (for "magic amulet") you'll be transported somewhere at random within the castle (but not to the entrance or exit). Only one line is needed to achieve this.

If you enter a "C" (for "consume") you are sent to the Eat Food routine, from line 2470, where the computer first checks to see if you have any food. If you do not (that is, the variable FOOD equals zero) the program returns from the subroutine.

If you do have food, the computer tells you how much you have, and then you are asked how much you want to eat (line 2520). Each unit of food gives you five units of strength (line 2560). After a short delay (called by 2570) the program returns.

Picking Up Treasure

The Pick Up Treasure routine from 2340 has two checks which are imposed before you are able to actually take possession of treasure to add to your store of wealth.

First the program checks to see if there is actually any treasure there (line 2360), sending you back to enter a new command with the words THERE IS NO TREASURE TO PICK UP. Once you have passed this hurdle, the computer checks to see if you have a flaming torch. You'll recall we discussed this earlier, when looking at the Light/Dark Routine. Line 2370 was outlined at that time, and this is the line which makes the check.

If you have passed both of these checks, the value of the treasure is added to your wealth (line 2380) and the room is "emptied" (2390) by setting the relevant element to zero so that you cannot simply revisit the same room over and over again and pick up the same treasure several times.

The Quartermaster's Store

One of your real tests within Adventure programs — after the major tests of trying to make sense of the environment, and staying alive — is the management of your resources. You'll recall I suggested that the more you expect the player to do on each game turn, the more satisfactory the program is likely to be.

Although you know, for example, that you need a torch in order to see

anything, and to pick up treasurer, a new player to the game may take quite a while to come to this conclusion. All a new player needs to know is the vocabulary, the fact that if he or she runs out of strength death is on the way, and that the purpose of the quest is to find the exit from the castle.

When you're designing an Adventure program, work out exactly what you're going to tell the player before the start of a game. An Adventure game with a computer is in part like an extended puzzle, in which the player tries to work out the problems set by the programmer. In WEREWOLVES AND WANDERER, as with the programs you are likely to write, there is far more happening than the simple solving of puzzles set by the programmer.

The Inventory/Provisions routine of this program is one of the keys to controlling the game. Once a player realizes that a torch is vital, and that it can be bought by going to the Inventory/Provisions routine, and that possession of an axe, sword or magic amulet can be a great help in trouble, he or she is well on the way to ensuring survival until the map of the castle can be deduced.

The Inventory/Provisions routine is largely self-explanatory. You'll see the program first checks to see how much money you have (by going to the subroutine from 3260) and tells you the state of your finances. It will send you back to the main program (3040) if you have no money left (that is, if the variable WEALTH has a value less than .1).

Next the menu of possibilities is shown, with a code number next to each. You enter a 1 if you want to buy a flaming torch, for instance, 6 for a suit of armor, and 0 to return to the main program. After you enter your chosen number (3120) the computer uses the lines from 3130 through to 3180 to allow you to "make your purchase" by modifying variables and subtracting the cost from your WEALTH. This holds true except for FOOD (item 4 on the menu) which is handled slightly differently.

Do not, whatever you do, try to cheat by buying more than you can afford. The drastic line 3190 prints up YOU HAVE TRIED TO CHEAT ME and everything you own — except for one quarter of your food — is taken away for you.

If you have selected item 4 on the menu, indicating that you want to buy food, the routine from 3210 to 3240 looks after this for you.

The final subroutine (3260 to 3290) tells you how much money (if any) you have, PRINTs four blank lines, then returns for your next menu selection. This routine cycles — as I pointed out earlier — until you enter zero, indicating that you wish "TO CONTINUE ADVENTURE."

The Big Fight

Fights (or melees, as they are often known in Adventure games) are often the high points of a game. The fight consists of three distinct segments, and each segment should be handled as a separate programming task. The three segments, which we will discuss one at a time, are Preparation, The Battle itself, and The Result.

Be Prepared

The fight segment uses INKEY$ to get it underway. (You can, by the way, change nearly all player inputs in WEREWOLVES AND WANDERER into INKEY$ inputs.) Line 740 waits until you are not touching the keyboard, then prints up (750) the message PRESS ANY KEY TO FIGHT. Line 760 holds the action until a key is pressed.

You'll recall from our talk about monsters that the variable FF stands for ferocity factor, and the higher the FF, the lower is your chance of defeating the particular monster. If you are wearing armour (as 770 informs you) your chance of success is increased, because FF is set to three-quarters of its former value.

If you have no weapons (that is, AXE equals zero and SWORD equals zero), the FF is increased by one fifth. It is important in programs like this one to give the player a real reason for spending his or her hard-won wealth on things such as weaponry. Adjusting the chance of success — as fiddling with FF does in this program — gives a specific, and rational, reason to invest in weapons.

If you have both an axe and a sword, lines 820 and 830 allow you to choose from between them, modifying your chances accordingly.

Taking Up Arms

The Battle itself is controlled by lines 880 to 940. This routine actually "stage-manages" the fight, maintains a role as referee, and reports to you during the fight on how it is going. Each time you manage to wound the

monster (line 910) the ferocity factor is reduced to five-sixths of its former value. Your strength is diminished by five each time the monster manages to wound you. Two-thirds of the time the computer comes to line 940, it will go back to line 890 for another round.

The Tumult and the Shouting Dies

Line 950 compares a number generated at random between zero and 15 with the ferocity factor, and if it is higher, gives you the victory, adding one to your MK (Monster Kill count). If you do not win, line 960 tells you the bad news, and halves your remaining strength.

Room Descriptions

Room descriptions add a third dimension to the raw plans of your Adventure environment. You can add as many details as you like to describing each room or cavern your Adventure discovers, or you can keep the descriptions short, and leave it up to the player's imagination to fill in the gaps.

I prefer to steer a middle course, adding some description, but not too much, to trigger the player's imagination but not to stifle it.

This section of the program is pretty simple to understand. The long ON GOSUB line (1010) send the computer to the relevant room, printing up a brief description of the room, and pointing out where the windows and doors are.

You should now refer back to the maps of the various floors, and see how they relate to the descriptions given.

All the subroutines simply print out a description, then return to line 1020 which returns action to the main program.

Special Handling

Two rooms are different. Room 9 is, as you can see from the map, the elevator, and this needs special handling. The delay loop is called twice, and then RO is set to 10 (line 1590), the room you end up in after using the elevator, before the program returns to the main handler.

The End of the Road

The other room needing special handling is room 11, the final exit. When the computer comes to the relevant subroutine, it immediately hits a return. It is put here, within the other room subroutines, just to keep the program clear. Once you've made it to room 11, the program stops cycling, and then announces your success. From here it simply follows on to lines 120 and 130, which we discussed earlier, to print up your score, and end the game.

The examples given should help you develop a number of ideas of your own to incorporate into room descriptions and "special events."

The SATAN BLINDER, the second program in this section, includes some quite difficult "special handling" situations. Although these situations are a little more complex than those outlined here, they use the ideas we have discussed.

The Listing

Here is the listing of WEREWOLVES AND WANDERER:

```
10 REM WEREWOLVES AND WANDERER
20 GOSUB 2600:REM INTIALISE
30 GOSUB 160
40 IF RO<>11 THEN 30
50 PEN 1:SOUND 5,100:PRINT:PRINT "YOU'VE
 DONE IT!!!":GOSUB 3520:SOUND 5,80:PRINT
 "THAT WAS THE EXIT FROM THE CASTLE!":SO
UND 5,200
60 GOSUB 3520
70 PRINT:PRINT "YOU HAVE SUCCEEDED, ";N$
;"!":SOUND 5,100
80 PRINT:PRINT "YOU MANAGED TO GET OUT O
F THE CASTLE"
90 GOSUB 3520
100 PRINT:PRINT "WELL DONE!"
110 GOSUB 3520:SOUND 5,80
120 PRINT:PRINT "YOUR SCORE IS";
130 PRINT 3*TALLY+5*STRENGTH+2*WEALTH+FO
OD+30*MK:FOR J=1 TO 10:SOUND 5,RND*100+1
0:NEXT J
```

```
140 PRINT:PRINT:PRINT:END
150 REM ----------------
160 REM MAJOR HANDLING ROUTINE
170 PEN 1:STRENGTH=STRENGTH-5
180 IF STRENGTH<10 AND STRENGTH>1 THEN P
RINT "WARNING, ";N$;", YOUR STRENGTH ":P
RINT "IS RUNNING LOW":PRINT
190 IF STRENGTH<1 THEN 2300:REM DEATH
200 TALLY=TALLY+1
210 PRINT N$;", YOUR STRENGTH IS";STRENG
TH
220 IF WEALTH>0 THEN PRINT "YOU HAVE £";
MID$(STR$(WEALTH),2)
230 IF FOOD>0 THEN PRINT "PROVISIONS SAC
K HOLDS";FOOD;"UNITS OF FOOD"
240 IF SUIT=1 THEN PRINT "YOU ARE WEARIN
G ARMOUR"
250 IF AXE=0 AND SWORD=0 AND AMULET=0 TH
EN 320
260 PRINT "YOU ARE CARRYING ";
270 IF AXE=1 THEN PRINT "AN AXE ";
280 IF SWORD=1 THEN PRINT "A SWORD ";
290 IF SWORD+AXE>0 AND AMULET=1 THEN PRI
NT "AND ";
300 IF AMULET=1 THEN PRINT "THE MAGIC AM
ULET"
310 PRINT
320 IF LIGHT=0 THEN PRINT "IT IS TOO DAR
K TO SEE ANYTHING"
330 IF LIGHT=1 THEN GOSUB 990:REM ROOM D
ESCRIPTION
340 K=A(RO,7):M$="":FF=0:REM K IS SET TO
 CONTENTS OF ROOM
350 IF K=0 THEN 450:REM ROOM IS EMPTY
360 IF K>9 THEN PRINT "THERE IS TREASURE
 HERE WORTH £";MID$(STR$(K),2):GOTO 450
370 PRINT:PRINT:PRINT "DANGER...THERE IS
 A MONSTER HERE....":GOSUB 3520
380 IF K=-1 THEN M$="FEROCIOUS WEREWOLF"
:FF=5
390 IF K=-2 THEN M$="FANATICAL FLESHGORG
ER":FF=10
400 IF K=-3 THEN M$="MALOVENTY MALDEMER"
:FF=15
```

```
410 IF K=-4 THEN M$="DEVASTATING ICE-DRA
GON":FF=20
420 IF K=-5 THEN M$="LAZIFFIRUS LOZE":FF
=25
430 IF K=-6 THEN M$="STAGGERING STEVUMPU
S":FF=30
440 IF M$<>"" THEN PRINT:PRINT "IT IS A
";M$:PRINT:PRINT "THE DANGER LEVEL IS";F
F;"!!!"
450 XU=RND*50+50:YU=RND*50+100:PEN 2:FLA
G=0:PRINT:PRINT:PRINT "WHAT DO YOU WANT
TO DO?":SOUND 1,XU:SOUND 5,YU:SOUND 5,XU
:SOUND 5,XU+YU
460 A$="":WHILE A$<"A" OR A$>"Z":A$=INKE
Y$:A$=UPPER$(A$):WEND
470 IF K<0 AND A$<>"F" AND A$<>"R" AND F
LAG=0 THEN 460
480 PEN 1:PRINT:PRINT:PRINT "----------
----------":PRINT:PEN 2
490 IF A$="Q" THEN 120
500 IF A$="N" AND A(RO,1)=0 THEN PRINT "
NO EXIT THAT WAY":GOTO 450
510 IF A$="S" AND A(RO,2)=0 THEN PRINT "
THERE IS NO EXIT SOUTH":GOTO 450
520 IF A$="E" AND A(RO,3)=0 THEN PRINT "
YOU CANNOT GO IN THAT DIRECTION":GOTO 45
0
530 IF A$="W" AND A(RO,4)=0 THEN PRINT "
YOU CANNOT MOVE THROUGH SOLID STONE":GOT
O 450
540 IF A$="U" AND A(RO,5)=0 THEN PRINT "
THERE IS NO WAY UP FROM HERE":GOTO 450
550 IF A$="D" AND A(RO,6)=0 THEN PRINT "
YOU CANNOT DESCEND FROM HERE":GOTO 450
560 IF A$="R" AND RND>0.7 THEN 2420
570 IF A$="R" THEN FLAG=1:PRINT "WHICH W
AY DO YOU WANT TO FLEE?":GOTO 460
580 IF A$="F" AND A(RO,7)>-1 THEN PRINT
"THERE IS NOTHING TO FIGHT HERE":GOTO 44
0
590 IF A$="I" THEN GOSUB 3010:RETURN
600 IF A$="C" AND FOOD=0 THEN PRINT "YOU
 HAVE NO FOOD":GOTO 440
```

```
610 IF A$="P" THEN GOSUB 2350:RETURN
620 IF A$="F" THEN 730
630 IF A$="M" THEN RO=INT(RND*19)+1:IF R
O=6 OR RO=11 THEN 630 ELSE 710
640 IF A$="C" THEN GOSUB 2480:RETURN
650 IF A$="N" THEN RO=A(RO,1)
660 IF A$="S" THEN RO=A(RO,2)
670 IF A$="E" THEN RO=A(RO,3)
680 IF A$="W" THEN RO=A(RO,4)
690 IF A$="U" THEN RO=A(RO,5)
700 IF A$="D" THEN RO=A(RO,6)
710 RETURN
720 REM ------------------
730 REM FIGHT
740 PEN 1:WHILE INKEY$<>"":WEND
750 PRINT "PRESS ANY KEY TO FIGHT"
760 WHILE INKEY$="":SOUND 5,RND*100:WEND
770 IF SUIT=1 THEN PRINT "YOUR AMOUR INC
REASES YOUR CHANCE OF      SUCCESS":FF=3*
(INT(FF/4)):GOSUB 3520
780 :CLS:FOR J=1 TO 6:PEN INT(RND*3+1):P
RINT TAB(3.5*J);"-*=*-*=*-*=*-*=*-":NEXT
 J:PEN 2
790 IF AXE=0 AND SWORD=0 THEN PRINT "YOU
 HAVE NO WEAPONS":PRINT "YOU MUST FIGHT
WITH BARE HANDS":FF=INT(FF+FF/5):GOTO 87
0
800 IF AXE=1 AND SWORD=0 THEN PRINT "YOU
 HAVE ONLY AN AXE TO FIGHT WITH":FF=4*IN
T(FF/5):GOTO 870
810 IF AXE=0 AND SWORD=1 THEN PRINT "YOU
 MUST FIGHT WITH YOUR SWORD":FF=3*INT(FF
/4):GOTO 870
820 PRINT "WHICH WEAPON? 1 - AXE, 2- SWO
RD?"
830 A$="":WHILE A$<"1" OR A$>"2":A$=INKE
Y$:WEND:Z=VAL(A$)
840 IF Z=1 THEN FF=4*INT(FF/5)
850 IF Z=2 THEN FF=3*INT(FF/4)
860 REM ---------------------
870 REM THE BATTLE
880 PRINT:PRINT
890 IF RND>0.5 THEN PRINT M$;" ATTACKS"
ELSE PRINT "YOU ATTACK"
```

```
900 GOSUB 3520
910 IF RND>0.5 THEN PRINT:PRINT "YOU MAN
AGED TO WOUND IT":FF=INT(5*FF/6)
920 GOSUB 3520
930 IF RND>0.5 THEN PRINT:PRINT "THE MON
STER WOUNDS YOU!":STRENGTH=STRENGTH-5
940 IF RND>0.35 THEN SOUND 5,RND*100:GOT
O 890
950 IF RND*16>FF THEN PRINT "AND YOU MAN
AGED TO KILL                    THE ";M$:MK
=MK+1:GOTO 970
960 PRINT:PRINT "THE ";M$;" DEFEATED YOU
":STRENGTH=INT(STRENGTH/2)
970 A(RO,7)=0:K=0:FF=0:M$="":GOSUB 3410:
PRINT:PRINT:GOSUB 3520:PEN 2:RETURN
980 REM -----------------
990 REM ROOM DESCRIPTIONS
1000 PEN 2:PRINT:PRINT "----------------
----------------":PRINT:PRINT:PEN 3
1010 ON RO GOSUB 1040,1100,1170,1230,128
0,1360,1410,1470,1540,1620,1700,1730,179
0,1860,1960,2030,2100,2160,2230
1020 PEN 1:RETURN
1030 REM -----------------
1040 REM ROOM 1
1050 PRINT "YOU ARE IN THE HALLWAY"
1060 PRINT "THERE IS A DOOR TO THE SOUTH
"
1070 PRINT "THROUGH WINDOWS TO THE NORTH
 YOU CAN    SEE A SECRET HERB GARDEN"
1080 RETURN
1090 REM ----------
1100 REM ROOM 2
1110 PRINT "THIS IS THE AUDIENCE CHAMBER
"
1120 PRINT "THERE IS A WINDOW TO THE WES
T. BY       LOOKING TO THE RIGHT"
1130 PRINT "THROUGH IT YOU CAN SEE THE E
NTRANCE     TO THE CASTLE."
1140 PRINT "DOORS LEAVE THIS ROOM TO THE
 NORTH,     EAST AND SOUTH"
1150 RETURN
1160 REM -----------
1170 REM ROOM 3
```

```
1180 PRINT "YOU ARE IN THE GREAT HALL, A
N L-SHAPED"
1190 PRINT "ROOM. THERE ARE DOORS TO THE
 EAST AND"
1200 PRINT "TO THE NORTH. IN THE ALCOVE
IS A DOOR   TO THE WEST"
1210 RETURN
1220 REM ------------
1230 REM ROOM 4
1240 PRINT "THIS IS THE MONARCH'S PRIVAT
E MEETING"
1250 PRINT "ROOM. THERE IS A SINGLE EXIT
 TO THE     SOUTH"
1260 RETURN
1270 REM ------------
1280 REM ROOM 5
1290 PRINT "THIS INNER HALLWAY CONTAINS
A DOOR TO"
1300 PRINT "THE NORTH, AND ONE TO THE WE
ST, AND A"
1310 PRINT "CIRCULAR STAIRWELL PASSES TH
ROUGH THE"
1320 PRINT "ROOM. YOU CAN SEE AN ORNAMEN
TAL LAKE"
1330 PRINT "THROUGH THE WINDOWS TO THE S
OUTH"
1340 RETURN
1350 REM ------------
1360 REM ROOM 6
1370 PRINT "YOU ARE AT THE ENTRANCE TO A
"
1380 PRINT "FORBIDDING-LOOKING STONE CAS
TLE. YOU    ARE FACING EAST"
1390 RETURN
1400 REM ------------
1410 REM ROOM 7
1420 PRINT "THIS IS THE CASLE'S KITCHEN.
 THROUGH"
1430 PRINT "WINDOWS IN THE NORTH WALL YO
U CAN SEE A";
1440 PRINT "SECRET HERB GARDEN. A DOOR L
EAVES THE   KITCHEN TO THE SOUTH"
1450 RETURN
```

```
1460 REM ------------
1470 REM ROOM 8
1480 PRINT "YOU ARE IN THE STORE ROOM, A
MIDST"
1490 PRINT "SPICES, VEGETABLES, AND VAST
 STACKS OF"
1500 PRINT "FLOUR AND OTHER PROVISIONS.
THERE IS A"
1510 PRINT "DOOR TO THE NORTH AND ONE TO
 THE SOUTH"
1520 RETURN
1530 REM -------------
1540 REM ROOM 9
1550 PRINT "YOU HAVE ENTERED THE LIFT...
"
1560 GOSUB 3520
1570 PRINT "IT SLOWLY DESCENDS..."
1580 GOSUB 3520
1590 RO=10
1600 GOTO 1000
1610 REM --------------
1620 REM ROOM 10
1630 PRINT "YOU ARE IN THE REAR VESTIBUL
E"
1640 PRINT "THERE ARE WINDOWS TO THE SOU
TH FROM"
1650 PRINT "WHICH YOU CAN SEE THE ORNAME
NTAL LAKE"
1660 PRINT "THERE IS AN EXIT TO THE EAST
, AND"
1670 PRINT "ONE TO THE NORTH"
1680 RETURN
1690 REM ---------------
1700 REM ROOM 11
1710 RETURN
1720 REM ---------------
1730 REM ROOM 12
1740 PRINT "YOU ARE IN THE DANK, DARK DU
NGEON"
1750 PRINT "THERE IS A SINGLE EXIT, A SM
ALL HALL IN"
1760 PRINT "THE WALL TOWARDS THE WEST"
1770 RETURN
```

```
1780 REM ----------------
1790 REM ROOM 13
1800 PRINT "YOU ARE IN THE PRISON GUARDR
OOM, IN THE";
1810 PRINT "BASEMENT OF THE CASTLE. THE
STAIRWELL"
1820 PRINT "ENDS IN THIS ROOM. THERE IS
ONE OTHER"
1830 PRINT "EXIT, A SMALL HOLE IN THE EA
ST WALL"
1840 RETURN
1850 REM ----------------
1860 REM ROOM 14
1870 PRINT "YOU ARE IN THE MASTER BEDROO
M ON THE"
1880 PRINT "UPPER LEVEL OF THE CASTLE...
"
1890 PRINT "LOOKING DOWN FROM THE WINDOW
 TO THE"
1900 PRINT "WEST YOU CAN SEE THE ENTRANC
E TO THE"
1910 PRINT "CASTLE, WHILE THE SECRET HER
B GARDEN IS";
1920 PRINT "VISIBLE BELOW THE NORTH WIND
OW. THERE"
1930 PRINT "ARE DOORS TO THE EAST AND TO
 THE SOUTH"
1940 RETURN
1950 REM ------------
1960 REM ROOM 15
1970 PRINT "THIS IS THE L-SHAPED UPPER H
ALLWAY."
1980 PRINT "TO THE NORTH IS A DOOR, AND
THERE IS A"
1990 PRINT "STAIRWELL IN THE HALL AS WEL
L. YOU CAN"
2000 PRINT "SEE THE LAKE THROUGH THE SOU
TH WINDOWS"
2010 RETURN
2020 REM -------------
2030 REM ROOM 16
2040 PRINT "THIS ROOM WAS USED AS THE CA
STLE"
```

```
2050 PRINT "TREASURY IN BY-GONE YEARS...
"
2060 PRINT "THERE ARE NO WINDOWS, JUST E
XITS TO THE";
2070 PRINT "NORTH AND TO THE EAST"
2080 RETURN
2090 REM --------------
2100 REM ROOM 17
2110 PRINT "OOOOH...YOU ARE IN THE CHAMB
ERMAIDS'"
2120 PRINT "BEDROOM. THERE IS AN EXIT TO
 THE WEST"
2130 PRINT "AND A DOOR TO THE SOUTH..."
2140 RETURN
2150 REM --------------
2160 REM ROOM 18
2170 PRINT "THIS TINY ROOM ON THE UPPER
EVEL IS THE";
2180 PRINT "DRESSING CHAMBER. THERE IS A
 WINDOW TO"
2190 PRINT "THE NORTH, WITH A VIEW OF TH
E HERB"
2200 PRINT "GARDEN DOWN BELOW. A DOOR LE
AVES TO      THE SOUTH"
2210 RETURN
2220 REM --------------
2230 REM ROOM 19
2240 PRINT "THIS IS THE SMALL ROOM OUTSI
DE THE"
2250 PRINT "CASTLE LIFT WHICH CAN BE ENT
ERED BY A"
2260 PRINT "DOOR TO THE NORTH. ANOTHER D
OOR LEADS"
2270 PRINT "TO THE WEST. YOU CAN SEE THE
 LAKE        THROUGH THE SOUTHERN WINDOWS
"
2280 RETURN
2290 REM ------------
2300 REM DEAD END
2310 PRINT "YOU HAVE DIED OF HUNGER AND
WEAKNESS.."
2320 GOSUB 3520
2330 GOTO 120
2340 REM --------------
```

```
2350 REM PICK UP
2360 IF A(RO,7)<10 THEN PRINT "THERE IS
NO TREASURE TO PICK UP":GOSUB 3520:RETUR
N
2370 IF LIGHT=0 THEN PRINT "YOU CANNOT S
EE WHERE IT IS":GOSUB 3520:RETURN
2380 WEALTH=WEALTH+A(RO,7)
2390 A(RO,7)=0
2400 RETURN
2410 REM --------------
2420 REM UNSUCCESSFUL ATTEMPT TO RUN
2430 PRINT "NO YOU MUST STAND AND FIGHT"
2440 A$="F"
2450 GOSUB 3520
2460 GOTO 590
2470 REM ---------------
2480 REM EAT FOOD
2490 CLS
2500 IF FOOD<1 THEN RETURN
2510 PRINT "YOU HAVE"FOOD"UNITS OF FOOD"
2520 PRINT "HOW MANY DO YOU WANT TO EAT"
;
2530 INPUT Z
2540 IF Z>FOOD THEN 2530
2550 FOOD=INT(FOOD-Z)
2560 STRENGTH=INT(STRENGTH+5*Z)
2570 GOSUB 3410:CLS
2580 RETURN
2590 REM -----------------
2600 REM INTIALISE
2610 MODE 1:BORDER 1:INK 0,1:INK 1,24:IN
K 2,26:INK 3,18:PAPER 0:PEN 2
2620 RANDOMIZE TIME
2630 WEALTH=75:FOOD=0
2640 STRENGTH=100
2650 TALLY=0
2660 MK=0:REM NO. OF MONSTERS KILLED
2670 REM --------------
2680 REM SET UP CASTLE
2690 DIM A(19,7):CHECKSUM=0
2700 FOR B=1 TO 19
2710 FOR C=1 TO 7
```

```
2720 READ A(B,C):CHECKSUM=CHECKSUM+A(B,C
)
2730 NEXT C:NEXT B
2740 IF CHECKSUM<>355 THEN PRINT "ERROR
IN ROOM DATA":END
2750 SOUND 3,90:INPUT "WHAT IS YOUR NAME
, EXPLORER";N$:N$=UPPER$(N$)
2760 CLS
2770 RO=6:REM STARTING POSITION (RO=ROOM
 NUMBER)
2780 SWORD=0
2790 AMULET=0
2800 AXE=0
2810 SUIT=0
2820 LIGHT=0
2830 REM --------------
2840 REM ALLOT TREASURE
2850 FOR J=1 TO 7
2860 M=INT(RND*19)+1
2870 IF M=6 OR M=11 OR A(M,7)<>0 THEN 28
60
2880 A(M,7)=INT(RND*100)+100
2890 NEXT J
2900 REM --------------
2910 REM ALLOT MONSTERS
2920 FOR J=1 TO 6
2930 M=INT(RND*18)+1
2940 IF M=6 OR M=11 OR A(M,7)<>0 THEN 29
30
2950 A(M,7)=-J
2960 NEXT J
2970 A(4,7)=100+INT(RND*100)
2980 A(16,7)=100+INT(RND*100)
2990 RETURN
3000 REM ---------------
3010 REM INVENTORY/PROVISIONS
3020 PRINT "PROVISIONS & INVENTORY"
3030 GOSUB 3260
3040 IF WEALTH<0.1 THEN Z=0:GOTO 3130
3050 PRINT "YOU CAN BUY 1 - FLAMING TORC
H ($15)"
3060 PRINT "              2 - AXE ($10)"
3070 PRINT "              3 - SWORD ($20)"
```

```
3080 PRINT "              4 - FOOD ($2 PER
 UNIT)"
3090 PRINT "              5 - MAGIC AMULET
 ($30)"
3100 PRINT "              6 - SUIT OF ARMO
UR ($50)"
3110 PRINT "              0 - TO CONTINUE
ADVENTURE"
3120 PRINT:PRINT "ENTER NO. OF ITEM REQU
IRED"
3125 Z$="":WHILE Z$<"0" OR Z$>"6":Z$=INK
EY$:WEND:Z=VAL(Z$):SOUND 5,50*Z
3130 IF Z=0 THEN CLS:RETURN
3140 IF Z=1 THEN LIGHT=1:WEALTH=WEALTH-1
5
3150 IF Z=2 THEN AXE=1:WEALTH=WEALTH-10
3160 IF Z=3 THEN SWORD=1:WEALTH=WEALTH-2
0
3170 IF Z=5 THEN AMULET=1:WEALTH=WEALTH-
30
3180 IF Z=6 THEN SUIT=1:WEALTH=WEALTH-50
3190 IF WEALTH<0 THEN PRINT "YOU HAVE TR
IED TO CHEAT ME!":WEALTH=0:SUIT=0:LIGHT=
0:AXE=0:SWORD=0:AMULET=0:FOOD=INT(FOOD/4
):GOSUB 3520
3200 IF Z<>4 THEN 3030
3210 INPUT "HOW MANY UNITS OF FOOD";Q:Q=
INT(ABS(Q))
3220 IF 2*Q>WEALTH THEN PRINT "YOU HAVE'
T GOT ENOUGH MONEY":GOTO 3210
3230 FOOD=FOOD+Q
3240 WEALTH=WEALTH-2*Q
3250 GOTO 3030
3260 IF WEALTH>0 THEN PRINT:PRINT:PRINT
"YOU HAVE $";MID$(STR$(WEALTH),2)
3270 IF WEALTH<=0 THEN PRINT "YOU HAVE N
O MONEY":GOSUB 3520:RETURN
3280 PRINT
3290 RETURN
3300 REM ----------------
3310 DATA 0,2,0,0,0,0,0
3320 DATA 1,3,3,0,0,0,0
3330 DATA 2,0,5,2,0,0,0
3340 DATA 0,5,0,0,0,0,0
```

```
3350 DATA 4,0,0,3,15,13,0
3360 DATA 0,0,1,0,0,0,0
3370 DATA 0,8,0,0,0,0,0
3380 DATA 7,10,0,0,0,0,0
3390 DATA 0,19,0,8,0,8,0
3400 DATA 8,0,11,0,0,0,0
3410 DATA 0,0,10,0,0,0,0
3420 DATA 0,0,0,13,0,0,0
3430 DATA 0,0,12,0,5,0,0
3440 DATA 0,15,17,0,0,0,0
3450 DATA 14,0,0,0,0,5,0
3460 DATA 17,0,19,0,0,0,0
3470 DATA 18,16,0,14,0,0,0
3480 DATA 0,17,0,0,0,0,0
3490 DATA 9,0,16,0,0,0,0
3500 REM ---------------
3510 REM DELAY LOOP
3520 FOR T=1 TO 900:NEXT T
3530 RETURN
```

The Satan Blinder

With our second Adventure, we break new ground. This program takes place in a deserted medieval castle in France. The name of the castle is real, all else is imaginary.

Puzzles

There are a number of ways you can increase the level of interest in an Adventure. One way is to have puzzles which the player must solve before continuing. In this program, the puzzle concerns a dwarf, who is going to make sure that if you stumble into the room where he lives, you will not be able to leave, unless you manage to hit the one thing that will enable you to continue.

There are two locked doors, with keys a long way away from the relevant rooms, and as you've experienced before, you must have the right key for the right lock.

Two-Word Sentences

The main difference between this Adventure and the previous one, is that

in this one the computer is given the facility of understanding English — or at least appearing to do so.

The computer is programmed to recognize and act on two-word sentences. The two words must be a verb first, followed by a noun. The computer, for example, can understand the instruction GO NORTH (or even RUN NORTH, MOVE NORTH, CLIMB NORTH, or WALK NORTH). You'll find it is far more satisfying to "talk" to your computer in this way than it is to communicate simply by touching a key.

As was pointed out at the start of the book, some of the pleasure of Adventure programs comes from trying to work out which words it will understand, and which it will reject.

All the objects you encounter within the chateau can be manipulated — at least to the extent of being picked up or put down. Some objects (such as the MYSTIC SCROLL) can be read and others can be drunk. As you can tell from this, the vocabulary handling is quite extensive.

You have six "attributes". You'll see that these are slowly reduced as you make your moves. You must find the exit from the chateau before any of them is cut right down to zero.

Vocabulary Handling

The most important idea introduced in this program is the vocabulary handling which, I hope you'll agree, adds a completely new dimension to the program. First, the variable KW (which stands for "key word") is set to zero in line 420. If KW becomes equal to one, then the computer has recognized a phrase it can act on. Line 430 accepts the answer to the question "What do you want to do?".

Line 470 sets M equal to the length of the player's input. It cycles through, adding blank spaces, until it is seven characters long. Line 480 then strips off the first three letters. In common with most Adventure programs, only the first three letters of a command must be entered, so WAL will do for WALK, and FIG for FIGHT. Three letters have been found to be sufficient to avoid ambiguity, and it is good to allow the player to simply enter three letters, rather than the whole word, if that is what he or she wants to do.

Line 490 detects the word HELP and gives a cryptic, but useful, reply, then returns to the beginning of the major handling loop in this program.

Line 500 allows the player to QUIT and sends action to the routine which terminates the program, starting a line 2650.

The next section of the program is the most important part of the vocabulary handling routine. Line 520 goes through the player input, character by character, looking for the first space which it knows should occur between words one and two (as between RUN and EAST, or GET and DIAMOND). Having found it, the computer sets C$ equal to the first three letters of the second word, to go with its B$ which it earlier set equal to the first three letters of the first word. If it finds that the first letter of C$ is not a space it knows the player's input was two words, so goes to line 570 to continue processing the input.

Line 530 continues the scan through the input.

The computer comes to line 540 if the scan through the input finishes, and the gap between two words has not been found, or it discovers that the first character of C$ is a space.

The RND in line 540 chooses between two replies to tell the player a new input is needed, and then goes back to get a new input.

Here, now, is the heart of the computer's ability to "understand" the player's wishes. Line 570 checks to see if the player is in room 8 or in room 34, and if not, jumps to line 600. These two rooms are the ones which are locked at the start of the game, and it is important to check to see whether or not they are still locked before the program continues.

Even if the room is locked, it may contain a monster who feels like a fight, and this takes precedence over the need to unlock the door before proceeding. Look carefully at line 580. It allows the following words to be accepted:

STA — for STAB
KIL — for KILL
FIG — for FIGHT
KIC — for KICK
PUN — for PUNCH
SLA — for SLAY
ATT — for ATTACK

Any one of those will be accepted, and acted upon. You can see that a player who does not know, in advance, which words the computer will

accept, should easily find one or two which will allow the machine to carry out his or her wishes.

Line 590 is only acted on if the contents of the room (the seventh element of A array) is a value greater than 98. This means the door is locked. If this is true, and the B$ part of the string does not contain UNL (as n UNLOCK), the computer will just point out that "The doors are locked" and then go back to the start of the major handling routine for a new input.

Line 600 is a repeat of line 580, but this version of it is reached if the player is not in a room with locked (or potentially locked) doors. You'll see, at the end of line 600, that KW (the variable standing, you'll recall, for "key word") is set equal to 1, meaning a word has been recognized. Finding a word meaning "fight" the computer sets KW equal to 1, then goes to the subroutine from line 1510 where the fight takes place.

Line 610 accepts the words for movement, which are GO, MOVE, CLIMB, RUN, and WALK. It goes to the subroutine from 780, where the actual moves between rooms are made.

Line 620 allows the player to pick up things with TAKE, GET, STEAL, and LIFT. The desire to get rid of things is recognized by line 630, which accepts DROP, PUT, THROW, and BREAK (and when it reaches the subroutine from line 1240, prints out a message — such as WITH A CRASH, THE VASE SMASHES AGAINST THE WALL — which is related to the word you chose to use when making your input.

Line 640 is used when you wish to UNLock the door, 650 if you are OPEning something (such as the chests which litter the place), 660 for REAding, 670 for DRInking or SWAllowing, and line 680 for . . . But I'm not going to tell you. It was hinted at a little earlier, and I'm not going to repeat the hint. Far better than you have the pleasure of working it out for yourself.

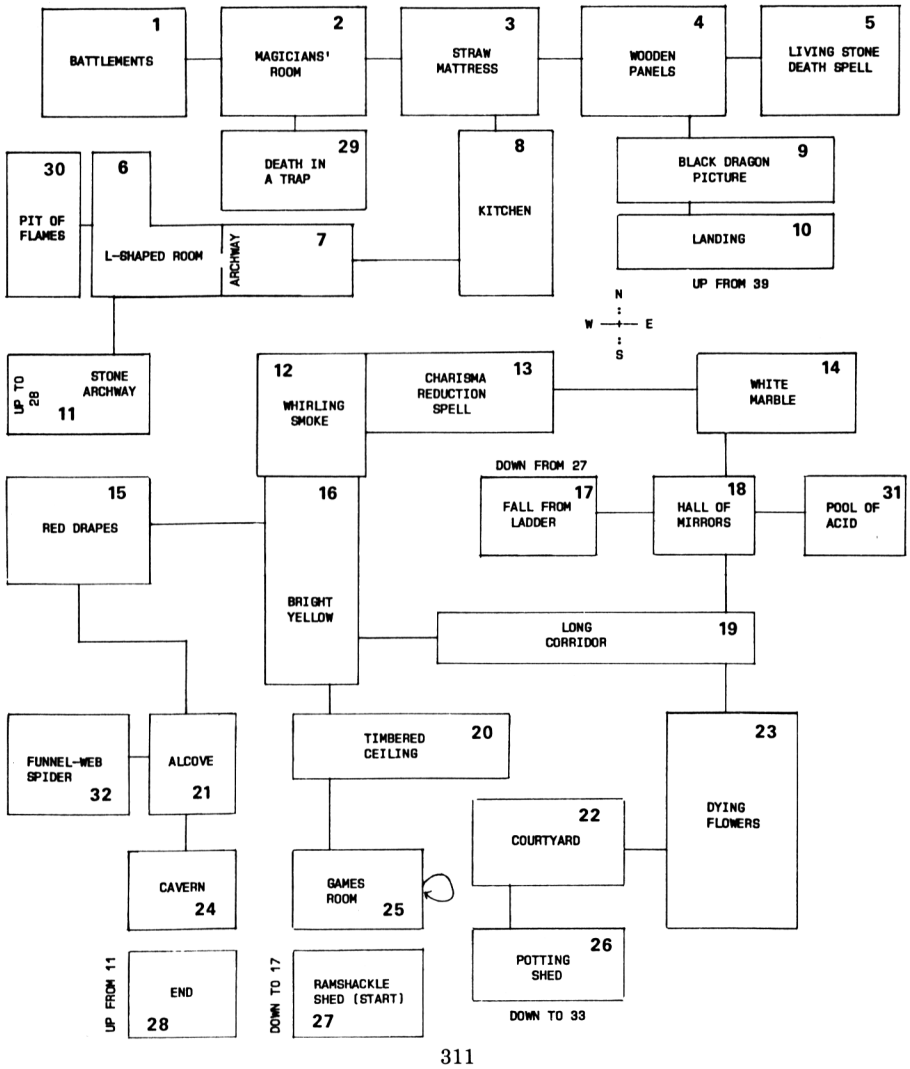
You'll recall that there was a section of code which rejected single word inputs. You'll also recall that the variable KW was to be set to 1 if a word the computer could understand was found. If the computer finds, at line 690, that KW is not equal to zero, then it knows all is probably well. If it does not, the player gets to the next section if he or she has entered a two-word phrase, but it is not one which the computer can understand. All obscene entries, often the input of very frustrated players, end up here. The computer chooses one of the three replies, each of which uses the phrase entered by the player, echoing back any rude words. After a pause

the program returns to the beginning of the major handling routine.

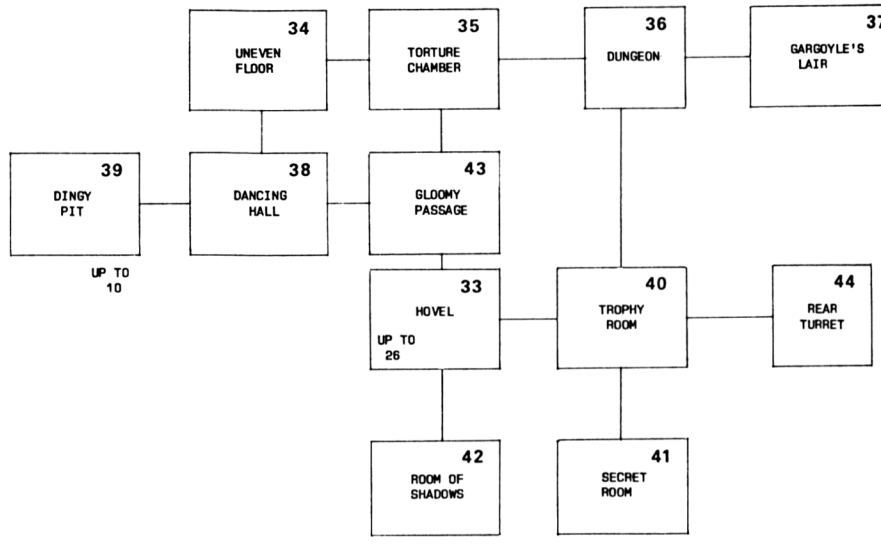
All other parts of this program are simply developments of ideas discussed earlier. You will find that, with the knowledge you've gather so far in your reading, you should have little trouble in working out how each section works.

The map is relatively complicated, as you can see. I suggest you try very hard not to look at it, until you've attempted to map the dungeon yourself.

FIRST LEVEL



SECOND LEVEL



The Listing

Here, now, is the full listing for THE SATAN BLINDER:

```
10 REM THE SATAN BLINDER
20 GOSUB 6880:REM INITIALISE
30 REM ----------------
40 REM REPORT TO PLAYER
50 GOSUB 7960
60 CLS
70 GOSUB 3410:REM ROOM DESCRIPTION
80 IF A(RO,7)<>0 OR A(RO,9)<>0 OR A(RO,1
0)<>0 THEN GOSUB 6750
90 IF A(RO,7)>98 THEN PRINT "The magic d
oors are locked, preventing  you from ex
ploring further..."
100 IF A(RO,8)<>0 THEN PRINT TAB(3);"Loo
k out!":PRINT "There is a ";M$(A(RO,8));
" here!":IF RND>0.7 AND A(RO,8)<>1 THEN
PRINT "The ";M$(A(RO,8));" attacks!":KW=
1:GOSUB 1510:GOTO 40
```

```
110 IF ST<1 THEN ST=0 ELSE IF RND>0.94 T
HEN ST=ST-1
120 IF CH<1 THEN CH=0 ELSE IF RND>0.94 T
HEN CH=CH-1
130 IF DE<1 THEN DE=0 ELSE IF RND>0.94 T
HEN DE=DE-1
140 IF IT<1 THEN IT=0 ELSE IF RND>0.94 T
HEN IT=IT-1
150 IF WI<1 THEN WI=0 ELSE IF RND>0.94 T
HEN WI=WI-1
160 IF CO<1 THEN CO=0 ELSE IF RND>0.94 T
HEN CO=CO-1
170 IF RND>0.84 AND RO=16 AND A(RO,8)=1
THEN PRINT:PRINT "You hear a whispered v
oice warning you...":PRINT "'Dwarvling,
devling, you must do...'"
180 PEN 3:PRINT:PRINT "Your attributes a
re:"
190 PRINT TAB(4);"Strength  -"ST;TAB(22)
;"Charisma      -"CH
200 PRINT TAB(4);"Dexterity -"DE;TAB(22)
;"Intelligence -"IT
210 PRINT TAB(4);"Wisdom    -"WI;TAB(22)
;"Constitution -"CO:PEN 2
220 IF ST*CH*DE*IT*WI*CO=0 THEN PRINT "Y
ou are exhausted....":PRINT "so this att
empt to become the Satan      Blinder mus
t end...":QU=2:GOTO 2710
230 GOTO 410
240 REM ----------
250 REM INVENTORY
260 PEN 1:FLAG=0
270 FOR J=1 TO 5
280 IF P(J)<>0 THEN FLAG=1
290 NEXT J
300 CASH=0
310 PRINT:PRINT "You are carrying:"
320 FOR J=1 TO 5
330 IF P(J)<>0 THEN PRINT TAB(4);O$(P(J)
):CASH=CASH+V(P(J))
340 NEXT J
350 IF FLAG=0 THEN PRINT TAB(4);"...noth
ing..."
```

```
360 IF CASH>0 THEN PRINT TAB(8);"Total v
alue - $";MID$(STR$(CASH),2)
370 PEN 2:PRINT
380 RETURN
390 REM --------------------
400 REM INPUT HANDLING
410 COUNTER=COUNTER+1
420 KW=0:REM =1 IF WORD UNDERSTOOD
430 PRINT:PEN 1:PRINT "What do you want
to do?":PEN 2:PRINT "                ";:INP
UT A$
440 IF A$="" THEN 430
450 A$=UPPER$(A$)
460 IF LEFT$(A$,3)="INV" THEN GOSUB 260:
GOTO 410
470 M=LEN(A$):IF M<7 THEN A$=A$+" ":GOTO
 470
480 B$=LEFT$(A$,3)
490 IF B$="HEL" THEN PRINT "No help for
mortals in this challenge,  although rea
ding and drinking may       be of some E
lvin aid..":GOTO 410
500 IF B$="QUI" THEN QU=4:GOTO 2650
510 N=1
520 IF MID$(A$,N,1)=" " THEN C$=MID$(A$,
N+1,3):IF LEFT$(C$,1)<>" " THEN 570 ELSE
 540
530 IF N<M THEN N=N+1:GOTO 520
540 IF RND>0.5 THEN 560
550 PRINT TAB(6);"By itself,";LOWER$(A$)
:PRINT TAB(6);"can't be acted ON":GOTO 4
30
560 PRINT LEFT$(A$,1);MID$(LOWER$(A$),2)
;" is just one word.":PRINT "I need two.
.":GOTO 430
570 IF RO<>8 AND RO<>34 THEN 600
580 IF B$="STA" OR B$="KIL" OR B$="FIG"
OR B$="KIC" OR B$="PUN" OR B$="SLA" OR B
$="ATT" THEN KW=1:GOSUB 1510
590 IF A(RO,7)>98 AND B$<>"UNL" THEN PRI
NT TAB (4);"The doors are locked!":GOTO
40
```

```
600 IF B$="STA" OR B$="KIL" OR B$="FIG"
OR B$="KIC" OR B$="PUN" OR B$="SLA" OR B
$="ATT" THEN KW=1:GOSUB 1510
610 IF B$="GO " OR B$="MOV" OR B$="CLI"
OR B$="RUN" OR B$="WAL" THEN KW=1:GOSUB
780:IF KW=9 THEN 410
620 IF B$="TAK" OR B$="GET" OR B$="STE"
OR B$="LIF" THEN KW=1:GOSUB 1030
630 IF B$="DRO" OR B$="PUT" OR B$="BRE"
THEN KW=1:GOSUB 1240
640 IF B$="UNL" THEN KW=1:GOSUB 2480
650 IF B$="OPE" THEN KW=1:GOSUB 2780
660 IF B$="REA" THEN KW=1:GOSUB 2880
670 IF B$="DRI" OR B$="SWA" THEN KW=1:GO
SUB 3000
680 IF B$="BRI" OR B$="PAY" THEN KW=1:GO
SUB 3130
690 IF KW<>0 THEN 740
700 R=INT(RND*3)
710 IF R=0 THEN PRINT "It would not be w
ise to ";LOWER$(A$)
720 IF R=1 THEN PRINT "Only a fool would
 try to ";LOWER$(A$)
730 IF R=2 THEN PRINT "I don't understan
d '";LOWER$(A$)
740 IF KW<>1 THEN 430
750 GOTO 40
760 REM END OF MAJOR HANDLER
770 REM ---------------
780 REM MOVEMENT
790 IF RO<>16 OR A(16,8)<>1 THEN 870
800 R=INT(RND*3)
810 PRINT
820 IF R=0 THEN PRINT "The dwarf refuses
 to let you proceed.."
830 IF R=1 THEN PRINT "The dwarf's face
contorts in rage. 'No, no, no', he shrie
ks. 'I will not let    you go on!!!'"
840 IF R=2 THEN PRINT "'No, no!', says t
he dwarf with a leer.  'You can't contin
ue! I'm not goin' to    let you!!'"
850 GOSUB 7960
860 RETURN
870 C$=LEFT$(C$,1)
```

```
880 PEN 3
890 IF C$="N" AND A(RO,1)=0 THEN PRINT "
You cannot go that way":KW=9:RETURN
900 IF C$="S" AND A(RO,2)=0 THEN PRINT "
There is no exit south":KW=9:RETURN
910 IF C$="E" AND A(RO,3)=0 THEN PRINT "
I see nowhere east to go":KW=9:RETURN
920 IF C$="W" AND A(RO,4)=0 THEN PRINT "
Even you cannot walk through walls":KW=9
:RETURN
930 IF C$="U" AND A(RO,5)=0 THEN PRINT "
There is no way to move up":KW=9:RETURN
940 IF C$="D" AND A(RO,6)=0 THEN PRINT "
You cannot descend from here":KW=9:RETUR
N
950 IF C$="N" THEN RO=A(RO,1)
960 IF C$="S" THEN RO=A(RO,2)
970 IF C$="E" THEN RO=A(RO,3)
980 IF C$="W" THEN RO=A(RO,4)
990 IF C$="U" THEN RO=A(RO,5)
1000 IF C$="D" THEN RO=A(RO,6)
1010 PEN 2:RETURN
1020 REM ------------
1030 REM GET OBJECTS ROUTINE
1040 FLAG=0:N=0
1050 IF A(RO,7)=0 AND A(RO,9)=0 AND A(RO
,10)=0 THEN PRINT "I see nothing to pick
 up":RETURN
1060 FOR J=1 TO 5
1070 IF P(J)<>0 THEN FLAG=FLAG+1
1080 NEXT J
1090 IF FLAG=5 THEN PRINT "You are alrea
dy carrying your":PRINT "maximum of five
 objects...":RETURN
1100 IF C$="CHE" THEN PRINT "It is far t
oo heavy to lift":RETURN
1110 D$="":E$="":F$=""
1120 D$=LEFT$(O$(A(RO,7)),3)
1130 E$=LEFT$(O$(A(RO,9)),3)
1140 F$=LEFT$(O$(A(RO,10)),3)
1150 IF D$=C$ THEN N=A(RO,7):A(RO,7)=0:G
OTO 1180
1160 IF E$=C$ THEN N=A(RO,9):A(RO,9)=0:G
OTO 1180
```

```
1170 IF F$=C$ THEN N=A(RO,10):A(RO,10)=0
1180 J=1
1190 IF P(J)=0 THEN P(J)=N:GOTO 1210
1200 IF J<5 THEN J=J+1:GOTO 1190
1210 PRINT TAB(3);"You now have the ";LO
WER$(O$(N))
1220 RETURN
1230 REM -------------
1240 REM DROP ROUTINE
1250 FLAG=0
1260 FOR J=1 TO 5
1270 IF P(J)<>0 THEN FLAG=1
1280 NEXT J
1290 IF FLAG=0 THEN PRINT "You are not c
arrying anything":RETURN
1300 IF A(R0,7)<>0 AND A(RO,9)<>0 AND A(
RO,10)<>0 THEN PRINT "This room already
holds it maximum       allowed number of
objects":RETURN
1310 D$="":D=0
1320 FOR J=1 TO 18
1330 IF LEFT$(O$(J),3)=C$ THEN D$=O$(J):
D=J
1340 NEXT J
1350 IF D$="" THEN PRINT TAB(3);"How can
 you, when you're                        no
t holding it?":RETURN
1360 FLAG=1
1370 FOR J=1 TO 5
1380 IF P(J)=D THEN P(J)=0:FLAG=0
1390 NEXT J
1400 IF FLAG=1 THEN D$="":GOTO 1350
1410 IF A(RO,7)=0 THEN A(RO,7)=D:FLAG=7:
GOTO 1450
1420 IF A(RO,9)=0 THEN A(RO,9)=D:FLAG=9:
GOTO 1450
1430 IF A(RO,10)=0 THEN A(RO,10)=D:FLAG=
10
1440 D$=LOWER(D$)
1450 IF B$="DRO" THEN PRINT "You have dr
opped the ";D$
1460 IF B$="PUT" THEN PRINT "You have pu
t the ";D$;" down"
```

```
1470 IF B$="THR" THEN PRINT "With a migh
ty heave, you throw the":PRINT "        "
;D$;"away............."
1480 IF B$="BRE" THEN PRINT "You have br
oken the ";D$:A(RO,FLAG)=0
1490 RETURN
1500 REM -------------
1510 REM FIGHT ROUTINE
1520 IF A(RO,8)=1 THEN PRINT "The dwarf
refuses to fight...":GOSUB 7960:PRINT ".
        ..and his magic protects him":RE
TURN
1530 IF A(RO,8)<>0 THEN 1580
1540 R=RND
1550 IF R<0.5 THEN PRINT "There is nothi
ng to fight here"
1560 IF R>=0.5 THEN PRINT "You can't fig
ht empty air!!!"
1570 RETURN
1580 G$=LOWER$(M$(A(RO,8)))
1590 S1=INT(RND*6+RND*6+RND*6)+3
1600 H1=INT(RND*6+RND*6+RND*6)+3
1610 D1=INT(RND*6+RND*6+RND*6)+3
1620 I1=INT(RND*6+RND*6+RND*6)+3
1630 W1=INT(RND*6+RND*6+RND*6)+3
1640 C1=INT(RND*6+RND*6+RND*6)+3
1650 MT=0:HT=0:REM MONSTER TALLY,
                        HUMAN TALLY
1660 FF=S1*(INT(RND*6+1))

1670 PRINT:PRINT "----------------------
----------"
1680 PRINT "The ";G$;"'s danger level is
"FF
1690 GOSUB 7960
1700 FOR J=1 TO 5
1710 T(J)=0
1720 IF P(J)=1 THEN PRINT "Your axe coul
d be handy":T(J)=1
1730 IF P(J)=2 THEN PRINT "Your skill wi
th the sword may stand                you
 in good stead":T(J)=2
```

```
1740 IF P(J)=3 THEN PRINT "    Your dagg
er is useful against                    ";
LOWER$(G$);"s":T(J)=3
1750 IF P(J)=4 THEN PRINT "The mace will
 make short work of it!":T(J)=4
1760 IF P(J)=5 THEN PRINT "Your quarters
taff will give                       it
no quarter...":T(J)=5
1770 IF P(J)=6 THEN PRINT "Swinging your
 Morning Star may inflict    heavy wound
s on the ";G$:T(J)=6
1780 IF P(J)=7 THEN PRINT "A falchion is
 a useful weapon...":T(J)=7
1790 NEXT J
1800 FLAG=0:PRINT
1810 FOR J=1 TO 5
1820 IF T(J)<>0 THEN FLAG=FLAG+1
1830 NEXT J
1840 IF FLAG=0 THEN PRINT "You must figh
t the ";G$:PRINT"with your bare hands":G
OTO 2020
1850 IF FLAG>1 THEN 1920
1860 FOR J=1 TO 5
1870 IF T(J)<>0 THEN FLAG=T(J)
1880 NEXT J
1890 PRINT "You must fight with your ";L
OWER$(O$(FLAG))
1900 FF=INT(FF*2/FLAG)
1910 GOTO 2020
1920 PRINT "Choose your weapon:"
1930 FOR J=1 TO 5
1940 IF P(J)<>0 THEN PRINT J;" - ";O$(J)
1950 NEXT J
1960 PRINT "Enter the number of the weap
on to       select it..."
1970 XX$=INKEY$:IF XX$<"1" OR XX$>"5" TH
EN 1970
1980 J=VAL(XX$)
1990 IF P(J)=0 THEN PRINT "You do not ha
ve the ";LOWER$(O$(J)):PRINT:GOTO 1960
2000 PRINT:PRINT "So you choose to fight
 with the        ";LOWER$(O$(J))
2010 FF=INT(FF*2/J)
```

```
2020 GOSUB 7960
2030 PEN 3:PRINT:PRINT "The ";G$;" has t
he following":PRINT " attributes:"
2040 PRINT "1 - Strength "S1;TAB(19);"2
- Charisma     "H1
2050 PRINT "3 - Dexterity";D1;TAB(19);"4
 - Intelligence"I1
2060 PRINT "5 - Wisdom    "W1;TAB(19);"6
- Constitution"C1
2070 PRINT:PRINT "Your attributes are:"
2080 PRINT "1 - Strength ";ST;TAB(19);"2
 - Charisma     "CH
2090 PRINT "3 - Dexterity";DE;TAB(19);"4
 - Intelligence"IT
2100 PRINT "5 - Wisdom    ";WI;TAB(19);"6
 - Constitution"CO
2110 PEN 2:PRINT:PRINT "     Which attrib
utes will you                fight with?
(Enter 2 numbers,            separated by
 a comma)"
2120 INPUT Z,Q:PEN 1
2130 IF Z<1 OR Q<1 OR Z>6 OR Q>6 OR Z=Q
THEN PRINT "Do not fool around when a ";
G$:PRINT "is in the room with you!":GOTO
 2120
2140 IF Z=1 OR Q=1 THEN MT=MT+S1:HT=HT+S
T
2150 IF Z=2 OR Q=2 THEN MT=MT+H1:HT=HT+C
H
2160 IF Z=3 OR Q=3 THEN MT=MT+D1:HT=HT+D
E
2170 IF Z=4 OR Q=4 THEN MT=MT+I1:HT=HT+I
T
2180 IF Z=5 OR Q=5 THEN MT=MT+W1:HT=HT+W
I
2190 IF Z=6 OR Q=6 THEN MT=MT+C1:HT=HT+C
O
2200 IF HT=MT THEN PRINT TAB(10);"You ar
e evenly matched":GOTO 2240
2210 PRINT "It looks like the odds are i
n"
2220 PRINT "        favour of ";
```

```
2230 IF HT>MT THEN PRINT "you" ELSE PRIN
T "the ";G$
2240 K=INT(RND*7)
2250 GOSUB 7960
2260 PEN 3:PRINT TAB(6);"> The ";G$;" -"
MT
2270 PRINT TAB(LEN(G$)+7);"> You -"HT:PR
INT:PEN 2:IF RND>0.5 THEN PEN 1
2280 IF K=0 THEN PRINT "You struck a spl
endid blow!":MT=MT-2
2290 IF K=1 THEN PRINT "The ";G$;" strik
es out":HT=HT-3:ST=ST-1:CH=CH-1
2300 IF K=2 THEN PRINT "You draw the ";G
$;"'s blood!":MT=MT-1
2310 IF K=3 THEN PRINT "You are wounded!
!":HT=HT-INT(RND*3)-1:D=DE-1
2320 IF K=4 THEN PRINT "The ";G$;" is ti
ring":MT=MT-1
2330 IF K=5 THEN PRINT "You are bleeding
...":HT=HT-1:WI=WI-1:CO=CO-1
2340 IF K=6 THEN PRINT "You wounded the
";G$:MT=MT-2
2350 IF RND>0.25 AND HT>0 AND MT>0 THEN
FOR T=1 TO 1600:NEXT T:GOTO 2240
2360 IF HT>MT THEN PRINT "You have slain
 the ";G$:MK=MK+1:GOTO 2440
2370 PRINT "The ";G$;" got the better":P
RINT "of you that time..."
2380 IF Z=1 OR Q=1 AND RND>0.5 THEN ST=4
*INT(ST/5)
2390 IF Z=2 OR Q=2 AND RND>0.5 THEN CH=3
*INT(CH/4)
2400 IF Z=3 OR Q=3 AND RND>0.5 THEN DE=6
*INT(DE/7)
2410 IF Z=4 OR Q=4 AND RND>0.5 THEN IT=2
*INT(IT/3)
2420 IF Z=5 OR Q=5 AND RND>0.5 THEN WI=5
*INT(WI/6)
2430 IF Z=6 OR Q=6 THEN CO=INT(CO/2)
2440 A(RO,8)=0
2450 PEN 2:GOSUB 7990:GOSUB 7990
2460 RETURN
2470 REM --------------
```

```
2480 REM UNLOCK DOORS
2490 IF RO<>8 AND RO<>34 OR A(RO,7)<99 T
HEN PRINT "There is no locked door in th
is room":RETURN
2500 FLAG=0
2510 T=0
2520 FOR J=1 TO 5
2530 IF P(J)=17 THEN FLAG=1:T=J
2540 IF P(J)=18 THEN FLAG=2:T=J
2550 NEXT J
2560 IF FLAG=0 THEN PRINT "You do not ha
ve a key!"
2570 IF FLAG=2 AND RO=8 OR FLAG=1 AND RO
=34 THEN PRINT "That key does not fit th
is door!":RETURN
2580 PRINT "There is a creak as the key
turns..."
2590 GOSUB 7960
2600 PRINT "....The door is now unlocked
........."
2610 A(RO,7)=0
2620 P(T)=0
2630 RETURN
2640 REM ----------------
2650 REM END OF GAME
2660 SC=0:REM SCORE
2670 IF QU=4 THEN PRINT "I did not imagi
ne you would turn              out to be a
 quitter!":GOTO 2710
2680 PRINT:PRINT "CONGRATULATIONS!":PRIN
T "You have completed the adventure...."
2690 SC=100
2700 PRINT:PRINT
2710 SCI=(SC+2*CASH+4*MK+ST+2*CH+3*DE+4+
IT+5*WI+6*CO)/QU
2720 IF MK>0 THEN PRINT "You killed"MK"m
onsters ";:GOSUB 7990
2730 IF MK>0 AND CASH>0 THEN PRINT "and"
 ELSE PRINT
2740 PRINT "You found $";MID$((STR$(CASH
)),2);" worth of treasure...":GOSUB 7990
2750 PRINT:PRINT "Your score for this ad
venture is"INT(SCI*COUNTER)/10
```

```
2760 GOSUB 7990:PRINT:PRINT:PRINT:END
2770 REM ------------
2780 REM OPEN CHEST
2790 FLAG=0
2800 IF C$<>"CHE" THEN PRINT "That would
 not be wise...":RETURN
2810 IF RO<>13 AND RO<>40 THEN PRINT "Yo
u can't do that, I'm afraid...":RETURN
2820 IF BOX=1 THEN IF RND>0.6 THEN PRINT
 TAB(4);"It holds nothing but dust...":R
ETURN
2830 IF RO=13 OR BOX=1 THEN PRINT "It is
 empty.....":RETURN
2840 IF BOX<>0 THEN RETURN
2850 PRINT "Inside you find a parchment,
 with the   following message on it:":PR
INT:GOSUB 7960:PRINT TAB(7);"Too small t
o grasp,":PRINT TAB(7);"Too slippery to
hold"
2860 PRINT TAB(7);"But a little old man"
:PRINT TAB(7);"Can be bound by gold...":
BOX=1:RETURN
2870 REM ------------
2880 REM READ SCROLL
2890 FLAG=0
2900 FOR J=1 TO 5
2910 IF P(J)=12 THEN FLAG=1
2920 NEXT J
2930 IF FLAG=0 THEN PRINT "You're not ho
lding anything which you
can read...":RETURN
2940 R=INT(RND*3)
2950 IF R=0 THEN PRINT "It says:":PRINT:
PRINT TAB(7);"'If the key fits, insert i
t!'"
2960 IF R=1 THEN PRINT "The scroll reads
:":PRINT:PRINT TAB(7);"'Chesty knows bes
ty!'"
2970 IF R=2 THEN PRINT "It says:":PRINT:
PRINT TAB(7);"'Important: I think, there
fore I ...":PRINT TAB(7);"ulet is very'"
2980 GOSUB 7990:RETURN
2990 REM ------------
3000 REM DRINK POTION
```

```
3010 FLAG=0
3020 FOR J=1 TO 5
3030 IF P(J)=13 THEN FLAG=J
3040 NEXT J
3050 IF FLAG=0 THEN PRINT "I would not a
dvise you to                    drink ";C$
:RETURN
3060 PRINT "You are instantly filled wit
h healing   power and your strength is r
estored":FOR ZZ=1 TO 1500:NEXT ZZ
3070 PRINT:PRINT "      The bottle holdin
g the potion"
3080 PRINT "     magically fades from vi
ew..."
3090 ST=20
3100 P(FLAG)=0
3110 RETURN
3120 REM ----------------
3130 REM BRIBE DWARF
3140 IF A(RO,8)<>1 THEN PRINT "You shoul
dn't try that                    with a
 ";M$(A(RO,8)):RETURN
3150 PRINT "He demands the amulet!!!!!!!
!!!!!!"
3160 FLAG=0
3170 FOR J=1 TO 5
3180 IF P(J)=9 THEN FLAG=J
3190 NEXT J
3200 IF FLAG<>0 THEN 3360
3210 PRINT "You do not have it..."
3220 GOSUB 7960
3230 IF RND>0.5 THEN PRINT "He would acc
ept anything that he                re
ally wants...":GOTO 3330
3240 PRINT "He decides, however, to acce
pt a        'gift' of ";
3250 FOR J=1 TO 5
3260 IF P(J)<>0 THEN FLAG=J
3270 NEXT J
3280 IF FLAG=0 THEN 3320
3290 PRINT "the ";O$(FLAG)
3300 P(FLAG)=0
3310 GOTO 3380
3320 PRINT "anything valuable, but"
```

```
3330 PRINT "have nothing, so he kills yo
u!"
3340 QU=3
3350 GOTO 2690
3360 PRINT "It was lucky for you that yo
u had it!"
3370 P(FLAG)=0
3380 PRINT "The dwarf scampers away, chu
ckling          insanely to himself...":A
(RO,8)=0
3390 RETURN
3400 REM --------------
3410 REM ROOM DESCRIPTIONS
3420 IF RO>11 THEN 4310
3430 ON RO GOSUB 3450,3500,3600,3710,380
0,3910,3980,4030,4100,4180,4250
3440 RETURN
3450 REM 1
3460 PRINT "The wind sweeps by fiercely,
 out here"
3470 PRINT "on the battlements of FoulHo
lme."
3480 PRINT:PRINT "There is only one way
back..."
3490 RETURN
3500 REM 2
3510 PRINT "Take care! This is an eerie
room,"
3520 PRINT "where once foul magicians, s
wept up in"
3530 PRINT "their manic lust for power,
convorted"
3540 PRINT "with little jibbering sprite
s, and"
3550 PRINT "werebeasts with carrion and
death held"
3560 PRINT "loathsomely in their fetid b
reath.."
3570 PRINT:PRINT "Exits lead in three di
rections...and"
3580 PRINT "and evil smell intrudes from
 the south"
3590 RETURN
3600 REM 3
```

```
3610 PRINT "The remains of a stinking, r
otten"
3620 PRINT "mattress lies in one corner.
"
3630 PRINT "You see the mattress has bee
n ripped"
3640 PRINT "apart to find any treasure w
hich was"
3650 PRINT "                hidden in it!"
3660 PRINT
3670 PRINT "Light comes fitfully from a
window to"
3680 PRINT "the north, and around the do
ors to"
3690 PRINT "the south, east and west..."
3700 RETURN
3710 REM 4
3720 PRINT "This wooden-panelled room ma
kes"
3730 PRINT "you feel damp and uncomforta
ble"
3740 IF RND>0.5 THEN PRINT "A bloated ra
t scampers across the floor" ELSE PRINT
"A furtive bat flitters through the
 cobwebs which cover the ceiling"
3750 PRINT "There are three doors leadin
g from this"
3760 PRINT "room; one made of iron. The
hair on"
3770 PRINT "the back of your neck prickl
es. Your"
3780 PRINT "sixth sense warns you to cho
ose very,           very carefully.....
.."
3790 RETURN
3800 REM 5
3810 PRINT "You ignored your intuition..
..":PRINT
3820 GOSUB 7960
3830 PRINT "A SPELL OF LIVING STONE, pri
med"
3840 PRINT "to trap the first intruder h
as been"
```

```
3850 PRINT "set on you...";:GOSUB 7960:P
RINT "with your last"
3860 PRINT "seconds of life you have tim
e"
3870 PRINT "only to feel profound regret
..."
3880 QU=2
3890 SC=50
3900 GOTO 2710
3910 REM 6
3920 PRINT "You are in an L-shaped room.
..":PRINT
3930 PRINT "Heavy parchment lines the wa
lls."
3940 PRINT "You can see through an archw
ay"
3950 PRINT "to the east...but that is no
t"
3960 PRINT "the only exit from this room
"
3970 RETURN
3980 REM 7
3990 PRINT "There is an archway to the w
est,"
4000 PRINT "leading to an L-shaped room.
"
4010 PRINT:PRINT "A door leads in the op
posite direction."
4020 RETURN
4030 REM 8
4040 PRINT "This room, filled with a fou
l stench,"
4050 PRINT "must have been the chateau's
 main"
4060 PRINT "kitchen, as there is much ro
tted food"
4070 PRINT "lying about...at least you h
ope it is"
4080 PRINT "food lying on the ground..."
4090 RETURN
4100 REM 9
4110 PRINT "You find yourself in a small
, cramped"
```

```
4120 PRINT "     room.......which trigge
rs your"
4130 PRINT "          claustrophobia....
"
4140 PRINT:PRINT "There is a fantastical
ly detailed"
4150 PRINT "picture of a fierce bronze d
ragon"
4160 PRINT "painted on the north wall, a
ove the                door..."
4170 RETURN
4180 REM 10
4190 PRINT "A stairwell ends in this roo
m, which"
4200 PRINT "is more of a landing than a
real"
4210 PRINT "room. The door to the north
is"
4220 PRINT "made of heavy iron, which ha
s"
4230 PRINT "rusted over the centuries...
.."
4240 RETURN
4250 REM 11
4260 PRINT "You are in a very long room.
"
4270 PRINT:PRINT "Fresh air blows down s
ome stairs"
4280 PRINT "and rich red drapes cover th
e walls..."
4290 PRINT:PRINT "You can see doors to t
he south and               to the east."
4300 RETURN
4310 IF RO>22 THEN 5190
4320 ON RO-11 GOSUB 4340,4400,4500,4560,
4630,4700,4780,4880,4920,5040,5130
4330 RETURN
4340 REM 12
4350 PRINT "YOu have entered a room fill
ed"
4360 PRINT "with swirling, choking smoke
.":PRINT
4370 PRINT "You must leave quickly to re
main"
```

```
4380 PRINT "healthy enough to continue y
our                          chosen quest...
"
4390 RETURN
4400 REM 13
4410 PRINT "There is a mirror in the cor
ner.."
4420 PRINT:PRINT "You glance at it, and
feel"
4430 PRINT "suddenly very ill....":PRINT
4440 PRINT "You realise the looking-glas
s has"
4450 PRINT "been enfused with a SPELL OF
 CHARISMA                REDUCTION...."
4460 GOSUB 7960
4470 PRINT"Oh dear...oh dear...oh diddum
s..."
4480 CH=CH-1
4490 RETURN
4500 REM 14
4510 PRINT "This room is richly finished
, with"
4520 PRINT "a white marble floor. Strang
e"
4530 PRINT "footprints lead to the two d
oors"
4540 PRINT "from this room. Dare you fol
low them??"
4550 RETURN
4560 REM 15
4570 PRINT "You are in a long, long"
4580 PRINT "hallway, lined on each side"
4590 PRINT "with rich, scarlet drapes...
"
4600 PRINT:PRINT "They are parted halfwa
y down"
4610 PRINT "the east wall, where there i
s a door.."
4620 RETURN
4630 REM 16
4640 PRINT "Someone has spent a long tim
e"
```

```
4650 PRINT "painting this room a bright
yellow..."
4660 PRINT:PRINT "You remember reading t
hat"
4670 PRINT "yellow is the Ancient Oracle
's"
4680 PRINT "Colour of Warning.....
."
4690 RETURN
4700 REM 17
4710 PRINT "As you stumble down the ladd
er,"
4720 PRINT "you fall into the room. The
ladder"
4730 PRINT "crashes down behind you...th
ere"
4740 PRINT "is now no way back...":GOSUB
 7960
4750 PRINT:PRINT "A small door leads eas
t from"
4760 PRINT "this very cramped room..."
4770 RETURN
4780 REM 18
4790 PRINT "You find yourself in a Hall
of"
4800 PRINT "Mirrors...and see yourself"
```

```
4810 PRINT "reflected a hundred times or
"
4820 PRINT "more...Through the brilliant
 glare"
4830 PRINT "you can make out doors in al
l"
4840 PRINT "directions...You notice the"
4850 PRINT "mirrors around the east door
"
4860 PRINT "            are heavily tarni
shed..."
4870 RETURN
4880 REM 19
4890 PRINT "You find yourself in a long
corridor..."
4900 PRINT:PRINT "Your footsteps echo as
 you walk..."
4910 RETURN
4920 REM 20
4930 PRINT "You feel as if you've been w
andering"
4940 PRINT "around this chateau for ever
...":GOSUB 7960
4950 PRINT:PRINT "And you begin to despa
ir of ever..."
4960 PRINT "          .....escaping..."
4970 PRINT:PRINT "Still, you can't get t
oo depressed but"
4980 PRINT "must struggle on. Looking ar
ound, you"
4990 PRINT "see you are in a room which
has a"
5000 PRINT "heavy timbered ceiling, and
white"
5010 PRINT "roughly-finished walls..."
5020 PRINT:PRINT "There are two doors...
"
5030 RETURN
5040 REM 21
5050 PRINT "You are in a small alcove. Y
ou"
5060 PRINT "look around...but can see no
thing in"
```

```
5070 PRINT "the gloom. Perhaps if you wa
it a while"
5080 PRINT "your eyes will adjust to the
"
5090 PRINT "murky dark of this alcove...
"
5100 GOSUB 7960
5110 PRINT "Perhaps not..."
5120 RETURN
5130 REM 22
5140 PRINT "A dried-up fountain stands i
n the"
5150 PRINT "centre of this courtyard, wh
ich"
5160 PRINT "once held beautiful flowers.
..but"
5170 PRINT "they have long since died...
"
5180 RETURN
5190 IF RO>33 THEN 6020
5200 ON RO-22 GOSUB 5220,5270,5310,5380,
5430,5580,5640,5710,5770,5850,5980
5210 RETURN
5220 REM 23
5230 PRINT "The scent of dying flowers f
ills"
5240 PRINT "this brightly-lit room..."
5250 PRINT:PRINT "There are two exits fr
om it...
5260 RETURN
5270 REM 24
5280 PRINT "This is a round stone cavern
 off the"
5290 PRINT "side of the alcove to your n
orth..."
5300 RETURN
5310 REM 25
5320 PRINT "You find yourself in an enor
mous,"
5330 PRINT "circular room, which looks a
s if it"
5340 PRINT "was used as a games room. Ru
bble"
```

```
5350 PRINT "covers the floor, partially
blocking"
5360 PRINT "          the only exit..."
5370 RETURN
5380 REM 26
5390 PRINT "Through the dim mustiness of
"
5400 PRINT "this small potting shed you
can"
5410 PRINT "         see a stairwell..."
5420 RETURN
5430 REM 27 - START
5440 PRINT "And now, if you're sitting c
omfortably,                we'll begin...
":PRINT
5450 GOSUB 7990
5460 PRINT "You find yourself in a small
 wood,       which is over-shadowed by th
e gruesome  presence of Satan's Domain..
."
5470 GOSUB 7990
5480 PRINT:PRINT "You don't know why, bu
t you start to    feel nervous as you wa
lk through the    wood. Your pace quicke
ns. You glance       about nervously..."
5490 PRINT:PRINT "It starts to rain!"
5500 GOSUB 7990
5510 PRINT:PRINT "The rain gets heavier.
 What can you do? Up ahead you see, to y
our relief, a     "
5520 PRINT "small, ramshackle shed. Comi
ng up to     the shed, you see the door i
s hanging    at a crazy angle, with just
the top      hinge still attached..."
5530 PRINT:PRINT "Slowly, oh so slowly,
you push it open  and squeeze into the g
loomy shelter..."
5540 GOSUB 7990
5550 PRINT:PRINT "As your eyes adjust to
 the gloom, you   notice a hole in one c
orner. An old     ladder leads down from
 the hole..."
5560 PRINT:PRINT "Outside the shed, the
```

```
storm continues   to increase. A gust of
 wind catches the door, and it slams shu
t..."
5570 RETURN
5580 REM 28 - END
5590 PRINT "How wonderful! Fresh air, su
nlight..."
5600 GOSUB 7990
5610 PRINT "Bright yellow Melcrium birds
 are         singing...you are free at la
st..."
5620 PRINT:PRINT
5630 GOTO 2680
5640 REM 29
5650 PRINT "The smell came from musty, g
reen bodies rotting in huge, cruel man-t
raps..."
5660 GOSUB 7990
5670 PRINT:PRINT "One springs shut on yo
u..."
5680 PRINT "              ...trapping you
forever"
5690 WU=5.1
5700 GOTO 2690
5710 REM 30
5720 PRINT "You fall into a pit of flame
s...."
5730 IF RND>0.7 THEN 5720
5740 SC=10
5750 QU=3.4
5760 GOTO 2710
5770 REM 31
5780 PRINT "Aaaaaaaah.......you have fal
len into"
5790 GOSUB 7990
5800 PRINT "a pool of acid. Now you know
, too late"
5810 PRINT "why the mirrors were so badl
y                    tarnished...":PRINT
5820 SC=20
5830 QU=3
5840 GOTO 2710
5850 REM 32
```

```
5860 PRINT "It's too bad you chose that
exit from              the alcove..."
5870 GOSUB 7990
5880 PRINT:PRINT "A giant Funnelious Web
eratum Spider      leaps on you...and bef
ore you can"
5890 PRINT "react, bites you on the neck
!":GOSUB 7960
5900 PRINT:PRINT "You have 10 seconds to
 live..."
5910 T=0
5920 GOSUB 7990
5930 T=T+1
5940 IF T<11 THEN PRINT TAB(2*T);11-T:GO
TO 5920
5950 SC=3
5960 QU=5
5970 GOTO 2710
5980 REM 33
5990 PRINT "A stairwell leads into this
room, a"
6000 PRINT "poor and common hovel with m
any doors               and exits..."
6010 RETURN
6020 ON RO-33 GOSUB 6040,6090,6150,6200,
6330,6400,6450,6520,6600,6650,6690
6030 RETURN
6040 REM 34
6050 PRINT "It is jolly hard to see in t
his room,"
6060 PRINT "and you slip slightly on the
 uneven,"
6070 PRINT "          rocky floor..."
6080 RETURN
6090 REM 35
6100 PRINT "Horrors! This room was once
the neat"
6110 PRINT "little 'Ideal Home' torture
chamber of"
6120 PRINT "Satan's Domain...":GOSUB 796
0
6130 PRINT:PRINT "Skeletons lie on the f
loor, still with  chains around their cu
mbling bones..."
```

```
6140 RETURN
6150 REM 36
6160 PRINT "This is fun, indeed. You're
in yet       another room absolutely stuf
fed to the  gills with very unpleasant m
emories..."
6170 GOSUB 7960
6180 PRINT "This foul hole was the Domai
n's dungeon"
6190 RETURN
6200 REM 37
6210 PRINT "Oh no, this is the lair of a
 gruesome,  gooly-toothed, sap-sucking,
marrow-      munching, Gargoyle..."
6220 GOSUB 7960
6230 PRINT:PRINT "It has been held priso
ner here since    the Amstrad was first
invented (or 300  years, whichever is lo
ngest)"
6240 GOSUB 7960
6250 PRINT:PRINT "In its frenzy, she thr
ashes out at you"
6260 GOSUB 7960
6270 PRINT TAB(12);"and..."
6280 GOSUB 7960
6290 PRINT "snaps your neck like an old
toothpick!!"
6300 SC=0
6310 QU=3
6320 GOTO 2710
6330 REM 38
6340 PRINT "This was the Lower Dancing H
all..."
6350 PRINT "With doors to the north, the
 east, and"
6360 PRINT "to the west, you would seem
to be able"
6370 PRINT "to flee any unreasonable dan
ger..."
6380 GOSUB 7960
6390 RETURN
6400 REM 39
6410 PRINT "This is a dingy pit at the f
oot of"
```

```
6420 PRINT "some extremely dubious-looki
ng stairs."
6430 PRINT:PRINT "A door leads to the ea
st..."
6440 RETURN
6450 REM 40
6460 PRINT "Doors open to each compass p
oint from"
6470 PRINT "the Trophy Room of the Domai
n..."
6480 PRINT:PRINT "The heads of strange c
reatures shot by"
6490 PRINT "the ancestral owners are mou
nted high"
6500 PRINT "           up on each wall...
"
6510 RETURN
6520 REM 41
6530 PRINT "You have stumbled on a secre
t room..."
6540 GOSUB 7960
6550 PRINT:PRINT "Down here, eons ago, t
he ancient"
6560 PRINT "Necromancers of Thorin plied
 their"
6570 PRINT "evil craft...and the remnant
 of their"
6580 PRINT "spells hangs heavy on the ai
r..."
6590 RETURN
6600 REM 42
6610 PRINT "Cobwebs brush your face as y
ou make"
6620 PRINT "your way through the gloom o
f this"
6630 PRINT "          Room of Shadows..."
6640 RETURN
6650 REM 43
6660 PRINT "This gloomy, doomy, oh-so-st
oomy"
6670 PRINT "passage lies at the intersec
tion of                three rooms..."
6680 RETURN
```

```
6690 REM 44
6700 PRINT "You are in the rear turret r
oom, below"
6710 PRINT "the extreme western wall of
this"
6720 PRINT "           Satanic Domain..."
6730 RETURN
6740 REM ------------------
6750 REM DESCRIBE OBJECTS
6760 PEN 1:PRINT
6770 IF A(RO,7)>98 AND A(RO,9)=0 AND A(R
O,10)=0 THEN RETURN
6780 PRINT TAB(3);"You can see: ";
6790 IF A(RO,7)>98 THEN 6810
6800 IF A(RO,7)<>0 THEN PRINT O$(A(RO,7)
);" ";
6810 IF A(RO,9)>98 THEN 6830
6820 IF A(RO,9)<>0 THEN PRINT O$(A(RO,9)
);" ";
6830 IF A(RO,10)>98 THEN 6850
6840 IF A(RO,10)<>0 THEN PRINT O$(A(RO,1
0));" ";
6850 PEN 2:PRINT:PRINT
6860 RETURN
6870 REM --------------------
6880 REM Initialise
6890 MODE 1:BORDER 1:INK 0,1:INK 2,26:IN
K 1,24:INK 3,18:PAPER 0:PEN 2
6900 RANDOMIZE TIME
6910 LOCATE 1,20:PRINT "Please stand by.
.."
6920 DEFINT A-Z
6930 DIM A(44,10),P(5),O$(20),V(20),M$(2
0),T(5)
6940 ST=INT(RND*4+RND*4+RND*4)+7
6950 CH=INT(RND*4+RND*4+RND*4)+7
6960 DE=INT(RND*4+RND*4+RND*4)+7
6970 IT=INT(RND*4+RND*4+RND*4)+7
6980 WI=INT(RND*4+RND*4+RND*4)+7
6990 CO=INT(RND*4+RND*4+RND*4)+7
7000 IF ST<10 OR CH<10 OR DE<10 OR IT<10
 OR WI<10 OR CO<10 THEN 6940
7010 CASH=0:REM TREASURE
```

```
7020 RO=27:REM START ROOM
7030 QU=1:REM END OF GAME FLAG
7040 COUNTER=0:REM NUMBER OF MOVES MADE
7050 MK=0:REM MONSTERS KILLED
7060 BOX=0:REM CHEST FLAG
7070 A(8,7)=99:REM LOCKED DOOR ONE
7080 A(34,7)=100:REM LOCKED DOOR TWO
7090 A(40,7)=20:REM CHEST
7100 A(13,7)=19:REM CHEST
7110 A(16,8)=1:REM DWARF
7120 CHECKSUM=0
7130 REM SET UP ROOMS
7140 FOR X=1 TO 44
7150 FOR Y=1 TO 10
7160 READ A(X,Y)
7170 CHECKSUM=CHECKSUM+A(X,Y)
7180 NEXT:NEXT
7190 IF CHECKSUM<>3721 THEN LOCATE 1,20:
PRINT "Error in room DATA!!!!!":END
7200 REM DISTRIBUTE TREASURE
7210 FOR Z=1 TO 20
7220 READ O$(Z),V(Z)
7230 NEXT
7240 FOR Q=1 TO 16
7250 Z=INT(RND*44)+1
7260 IF Z=5 OR Z=17 OR Z=27 OR Z=29 OR Z
=30 OR Z=31 OR Z=32 OR Z=37 THEN 7250
7270 IF A(Z,7)<>0 THEN 7250
7280 A(Z,7)=Q
7290 NEXT
7300 LOCATE 1,20:PRINT "Just a few momen
ts more..."
7310 REM DISTRIBUTE MONSTERS
7320 FOR J=1 TO 20
7330 READ M$(J)
7340 IF J=1 THEN 7390
7350 Z=INT(RND*44)+1
7360 IF Z=5 OR Z=16 OR Z=17 OR Z=27 OR Z
=29 OR Z=30 OR Z=31 OR Z=32 OR Z=37 THEN
 7350
7370 IF A(Z,8)<>0 THEN 7350
7380 A(Z,8)=J
7390 NEXT
```

```
7400 RETURN
7410 REM ROOM DATA
7420 DATA 1,1,2,1,1,1,0,0,0,0
7430 DATA 0,29,3,1,0,0,17,0,0,0
7440 DATA 0,8,4,2,0,0,0,0,0,0
7450 DATA 0,9,5,3,0,0,0,0,0,0
7460 DATA 5,5,5,5,5,5,0,0,0,0
7470 DATA 0,11,7,30,0,0,0,0,0,0
7480 DATA 0,0,8,6,0,0,0,0,0,0
7490 DATA 3,0,0,7,0,0,99,0,0,0
7500 DATA 4,10,0,0,0,0,0,0,0,0
7510 DATA 9,0,0,0,0,39,0,0,0,0
7520 DATA 6,0,0,0,28,0,0,0,0,0
7530 DATA 0,16,13,0,0,0,0,0,0,0
7540 DATA 0,0,14,12,0,0,19,0,0,0
7550 DATA 0,18,0,13,0,0,0,0,0,0
7560 DATA 0,21,16,0,0,0,0,0,0,0
7570 DATA 12,20,19,15,0,0,0,1,0,0
7580 DATA 0,0,18,0,27,0,0,0,0,0
7590 DATA 14,19,31,17,0,0,0,0,0,0
7600 DATA 18,23,0,16,0,0,0,0,0,0
7610 DATA 16,25,0,0,0,0,0,0,0,0
7620 DATA 15,24,0,32,0,0,0,0,0,0
7630 DATA 0,26,23,0,0,0,0,0,0,0
7640 DATA 19,0,0,22,0,0,0,0,0,0
7650 DATA 21,0,0,0,0,0,0,0,0,0
7660 DATA 20,25,25,25,25,25,0,0,0,0
7670 DATA 22,0,0,0,0,33,0,0,0,0
7680 DATA 0,0,0,0,0,17,0,0,0,0
7690 DATA 0,0,0,0,0,11,0,0,0,0
7700 DATA 29,29,29,29,29,29,0,0,0,0
7710 DATA 30,30,30,30,30,30,0,0,0,0
7720 DATA 31,31,31,31,31,31,0,0,0,0
7730 DATA 32,32,32,32,32,32,0,0,0,0
7740 DATA 43,42,40,0,26,0,0,0,0,0
7750 DATA 0,38,35,0,0,0,100,0,0,0
7760 DATA 0,43,36,34,0,0,0,0,0,0
7770 DATA 0,40,37,35,0,0,0,0,0,0
7780 DATA 37,37,37,37,37,37,0,0,0,0
7790 DATA 34,0,43,39,0,0,0,0,0,0
7800 DATA 0,0,38,0,10,0,0,0,0,0
7810 DATA 36,41,44,33,0,0,20,0,0,0
7820 DATA 40,41,41,42,41,41,0,0,0,0
7830 DATA 33,42,41,42,42,42,0,0,0,0
```

```
7840 DATA 35,33,0,38,0,0,0,0,0,0
7850 DATA 0,0,0,40,0,0,18,0,0,0
7860 REM OBJECT/VALUE DATA
7870 DATA AXE,0,SWORD,0,DAGGER,0,MACE,0
7880 DATA QUARTERSTAFF,0,MORNING STAR,0,
FALCHION,0
7890 DATA CRYSTAL BALL,99,AMULET,247,EBO
NY RING,166,GEMS,462, MYSTIC SCROLL,195,
HEALING POTION,231,DILITHIUM CRYSTALS,16
2
7900 DATA COPPER PIECES,27,DIADEM,141,SI
LVER KEY,0,GOLDEN KEY,0,CHEST OF STONE,0
,CHEST MADE OF IRON,0
7910 REM DATA MONSTERS
7920 DATA DWARF,MONOCEROS,PARADRUS,VAMPY
RE,WRNACH,GIOLLA DACKER,KRAKEN,FENRIS WO
LF,CALOPUS,BASILISK,GYGAX
7930 DATA GRIMOIRE,FLYING BUFFALO,BERSER
KOID,WYRM,CROWTHERWOOD,RAGNAROK,FOMORINE
,HAFGYGR,GRENDEL
7940 REM ***********
7950 REM SHORT DELAY
7960 FOR ZZ=1 TO 1500:NEXT ZZ:RETURN
7970 REM ***********
7980 REM LONG DELAY
7990 FOR ZZ=1 TO 4500:NEXT ZZ:RETURN
```

Adding Excitement

Finally, in the section of the *Omnibus,* we'll look at a few ways you can add interest and "playability" to your programs.

The two most obvious things you should strive for are (a) unpredictability and (b) stability. These may seem to be mutually incompatible, but of course they are not. The unpredictability refers to the unknowns which a player must cope with when traversing the rocky plains of your Adventure. The stability refers to the environment which must (unless you have explicitly arranged for something different, such as a magician who causes the walls to shift every time you turn your head) be sufficiently stable and coherent to allow it to be mapped.

People who try to solve the Adventures you create enjoy making maps as they explore. To allow them to create maps, your Adventures should be relatively stable and coherent. As you've probably gathered, creating an initial map and a Travel Table from it, lies at the heart of Adventure programming. All else is commentary.

Structured Programming

The Adventures in this book were all written *after* an outline for the structure was developed. In essence, the programs consist of a "master loop" at the beginning of the program which calls each of the activities the program must carry out. Each of these activities is in a separate subroutine. The subroutines can be developed, and debugged, one by one. If an error is found, for example, in the way the contents of a room are printed out, it is far easier to go straight to a routine headed "ROOM CONTENTS" than it would be to wade through 20K of program.

Of course, in the real world, things are rarely as neat as they are in theory, Reworking areas of code tends to muddy the work of even the best-intentioned programmer. In the ideal world, an Adventure program might start like this:

```
 10  REM NAME OF ADVENTURE
 20  GOSUB 9000:REM INITIALIZE
 30  GOSUB 8000:REM PRINT PLAYER ATTRIBUTE
 40  GOSUB 1000:REM PRINT ROOM DESCRIPTION
 50  GOSUB 8000:REM CHECK IF MONSTER PRESENT,
                 AND IF SO, PRINT DESCRIPTION
 60  GOSUB 7000:REM CHECK IF TREASURE
                 PRESENT, AND IF SO, PRINT
                 DESCRIPTION
 70  GOSUB 2000:REM ACCEPT PLAYER INPUT
 80  IF player wants to pick something up then GOSUB 2500
 90  IF player wants to fight then GOSUB
     3000. . . . . and so on
150  GOTO 30
```

Working on paper tends to prevent code which really should not be in a program from somehow being patched together to make it work.

As well as helping in the early "get it working" stage, a structured program tends to invite improvements. Once you have your first original Adventure program up and running, you can go back to it some time in the future to try and make the fight sequence more interesting, for example,

by going to that part of the code which covers the Big Clash. You will not have to wade through acres of code, trying to work out just what each line does, and which particular lines control the monster interaction.

These methods were used for both programs in this section. It may prove instructive to follow through the listing of some of the programs and work out which section does what. In many programs, you'll see that I've included lines of hyphens as a REM statement. These break the program into separate modules which should help you follow the program through.

Once you have your program up and running satisfactorily, so the player can move around the environment in accord with your map, and the mechanics of "pick up", "drop," "eat," "trigger magic spell," and the rest are working, you can then start elaborating your masterpiece. The remainder of this final section is devoted to ideas which you can include as embroidery on your basic program.

The Purpose

There must be some reason for the player to be taking part in the Adventure. Write a brief scenario explaining where and why the Adventure is taking place. Give the player a realistic goal (such as to survive till he or she escapes, as in our Adventures), or to find the Golden Horn of Apostraphe and bring it safely back to base, to beat the Monster Hunting Club of America's current record of 342 of the beasties bagged in one session, to rescue the prince stranded at the top of his Castle Tower, or whatever. Make the purpose clear to the player, and make sure a significant percentage of the acts he or she takes part in are related to the achievement of the goal.

The Player Character

Instead of allowing the computer's random number generator to produce the initial character (with x points of intelligence and y of brute strength and only z of magic power), you could perhaps provide for the player to build his or her character from scratch, and then set it out on the Adventure. Running through the same Adventure a number of times with different characters could be very interesting. Of course, you'd have to put limits on the attributes a character could acquire, or you'd have some greedy soul taking the next ten million games worth of strength to be totally beyond defeat. He or she could just blunder wholesale through the environment, crushing all comers.

Experience

You may well wish to provide an opportunity to save a program while it is underway, so a player would not be forced to abort a complex Adventure in mid-flight and then retrace his or her early steps in order to try and solve it. The player's "experience" (attributes, cash, weapons, and the like) would also be saved along with the game in progress.

Quartermastering

Don't settle for just swords and broad axes. Use your imagination to offer a player a host of goodies. Rope often comes in useful. Let a player pass an apparently worthless coil of rope in one room, when six rooms later it is needed to get down a cliff (in the same way the silver key is needed for the first locked door in *The Satan Blinder*) and the player will soon learn that not only weapons and booty should be carried.

Magic

In "real life" role-playing games, combat and magic use are two of the highlights. Because of the infinite variety of magic spells which could be imagined (from causing the ground to fall open beneath an enemy's feet to turning into a canary and singing until everyone goes crazy) it is extremely difficult to program your computer to respond to magic in the same flexible way a Dungeon Master may do. However, we have seen two types of magic in our games (instant, random teleportation and battle victory) and there is no reason at all why you should not add as many additional kinds of magic as you like. Perhaps you might allow your player to choose from a menu of spells at the start of a game, before he or she knew what sort of problems lay ahead. They could have specific effects, which you could spell out in the program, or in notes provided for the player.

Limits

Assign a weight to each object which the player can find, and let the player know that he or she has only limited strength. Soon, the player will have to determine whether it is better to shuck the chainmail armour in order to carry a brace of invisibility spells, or to stick with the hardware and hope the wizardware won't be needed.

Research

Get a few popular history books. Early battle descriptions will give you a host of ideas to convert into terrain, weaponry, enemy reactions, and so on. Every period you read about will suggest at least one Adventure and probably more. The time and place will do much of the work involved in creating the background to the environment.

Handy Extras

As well as offering the player goodies throughout the environment (such as the map we mentioned), you could start with a much bigger arsenal than we have in the program in this book. How about a small group of mercenaries, who can be sacrificed as dragon fodder, or a pack pony to carry your loot?

Alternative Treks

If one of the purposes of the Adventure campaign is to amass as much treasure as possible, you might like to allow for the player to backtrack from time to time to dump his or her treasure in safety at the entrace, before moving back into the dungeon or whatever. A group of bandits, hidden in room 34, will probably take everything our hapless player is carrying, so if some valuables have already been spirited away — out of the main theatre of war — all is not lost.

Time Limits

If you feel some of your Adventures are a little too simple to solve, enhance them by including a real time clock in the program, one which counts down on the screen. This will keep the pressure up, and adds another layer of interest.

Climate

You might like to introduce elements of weather into your environment. A very cold room which means the player must burn something from his or her supplies to survive is one possibility. Another room could be filled with steam, which rusts swords but leaves axes untouched. A slimy floor can produce its own complications.

Other Ideas:

There are a number of other ideas you might like to incorporate into your Adventures, such as the following:

- A monster which does not stay passively in one room, but follows you relentlessly, once you've woken him up.

- A few other "pseudo-players" under the computer's control which appear to be exploring the environment as well. The player can meet these people from time to time, possibly getting information about future dangers from them.

- A "help" command for the player to invoke if he or she feels hopelessly lost. A severe penalty (such as half the player's gold) should be extracted for using this option to ensure it is not used frequently.

- Doors can be locked, impassable, stuck, or traps. Walls can fall in on players, floors give way to a gaping crevasse, and so on. Pictures can slide from walls, hitting our hapless player on the head, and so on. You might like to create an environment just for paranoid players, where every element seems hell-bent on injury.

And so on. I'm sure by now you've got a hundred ideas in your head just waiting to be expressed as Adventure programs. Good Adventuring. The suggestions for further reading which follow may well serve as sources of other ideas to incorporate into your own Adventures.

May the Dreaded Ice-Dragon not molest you, and may all your chests be filled with Elven Gold.

Suggestions for Further Reading

There is an impressive body of support literature for Adventure gaming, as a visit to your local Games shop will demonstrate. The selection which follows is very much a product of my own interest in the field, and it should

not be seen as even an attempt at selecting the "best" publications. However, the list is made up from those works which I've found of interest and value. There are probably over a hundred of equal worth, but at least this list should give you a starting point:

THROUGH DUNGEONS DEEP: A Fantasy Gamers' Handbook — Robert Plamondon (Reston Publishing Company, Inc., Reston, Virgina, 1982)

What is Dungeons and Dragons® — John Butterfield, Philip Parker and David Honigmann (Penguin Books Ltd., Harmondsworth, Middlesex, England, 1982) [*Dungeons and Dragons* is the adventure game which was originated by TSR Hobbies Inc.™ The term is a registered trademark.]

Dicing with Dragons, an Introduction to Role-Playing Games — Ian Livingstone (Routledge & Kegan Paul, London, Melbourne and Henley, 1982)

Fantasy Role-Playing Games — J. Eric Holmes (Hippocrene Books, Inc. New York, 1981)

As well as books, there are a number of game-playing aids which will help you in building the framework environment within which your Adventure will be enacted. Another good source of ideas is rule books for specific games. Among those which may help you are:

MONSTER AND TREASURE ASSORTMENT — TSR Games, POB 756, Lake Geneva, WI53147, 1980, distributed by Random House, Inc. This publication, which comes in a variety of "sets" and "levels" is a great place for getting ideas for monster names and treasure descriptions.

DUNGEON GEOMORPHS — Also produced by TSR Games, the "geomorphs" are a number of rectangular and square map sections, which can be fitted together in an infinite variety of ways to form dungeon and cavern floor plans.

BOOK O, AN INTRODUCTION TO TRAVELLER — Loren K. Wiseman (Games Designers' Workshop Inc. POB 1646, Bloomington, IL 61701). This is an example of the rule book I mentioned. *Traveller* is a science-fiction Adventure in the far future, and BOOK O is an introduction to the concepts of role-playing with specific attention to Traveller. As Book O points out, the "total amount of *Traveller* material available is staggering and growing all the time" (p. 11). Despite this, BOOK 0, BOOK 1, BOOK 2, and BOOK 3 are all your really need to play the game, and BOOK O is

enough, by itself, to give you a taste for laying out Adventure programs in deepest space. Leaf through a copy at your Games shop, and you'll see how many great ideas it contains which you can apply to your own programs. Note that Traveller is a registered trademark.

DUNGEONS & DRAGONS® boxed game sets are a good way to learn some of the possibilities of Adventure gaming. You can start (and stay, if you like) with the BASIC SET WITH INTRODUCTORY MODULE. This set typically contains two books, and a set of six dice, with various numbers of sides. The books are BASIC RULES, which set out the concepts of role-playing, and explain how characters are "rolled up," how their personalities are derived, how fights are sorted out, and many aspects of the Dungeon or Game Master's role.

As well as dice and BASIC RULES, the set contains a campaign background, THE KEEP ON THE BORDERLANDS. This comes with a great deal of information, including a series of maps, room and player information, and further details on fight resolution. I feel that the BASIC RULES boxed set is probably the best source of ideas you can get your hands on. It is also a very good way to achieve greater understanding of how role-playing games are developed and controlled.

WARNING: These sources of ideas are suggested, of course, only for your own use, to produce games for your own entertainment. You cannot incorporate copyright material into programs for public sale.

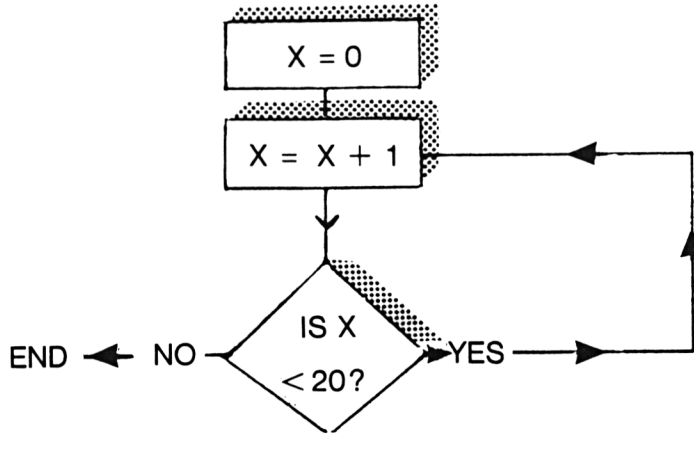
Section Seven
Structured Programming

There will come a point in your development as a programmer when you'll have mastered the use of much of the Amstrad's BASIC language, and can now concentrate on writing better programs; programs which work after relatively little debugging, which are easy for others to understand and operate, and which are written logically and elegantly.

Your Amstrad programs will be more likely to run first time if they are planned out carefully before you start entering code (i.e. program) into your computer.

A good way to start is to use a diagram which is often a 'flow chart'. A flow chart is a series of boxes and other shapes, joined by lines, which show the flow of action and decision-making within the computer while the program is running.

The shapes are not too important, and I suggest you stick to just two: a rectangle for most actions the computer must carry out, with a diamond shape each time the computer has to make a decision. The corners of the diamond can be used — as you can see in the diagram — to cater for the alternatives facing the computer.



The diagram shows the flow chart of a program which sets the variable X equal to zero, then adds one to it. The value of X is checked. If X is found to be less than 20, the program goes back to add one to X again. This continues until the value of X equals 20.

A flow chart is 'universal'. The same flow chart can be used as the basis of a program written on a computer furnished with a completely different language from the one in which you originally intended to write the program.

A flow chart models the flow of action and logic within a program, and is therefore very useful for picking up potential bugs at the earliest stages. You may, for example, find that one condition the program will test will never be fulfilled, possibly leading to the program being trapped in an infinite loop. Other parts of the code may be bypassed completely, because the condition which triggers entry into that part of the code will never be met.

Once you've devised a flow chart for your program, and you've run through it mentally a few times, so that the most obvious bugs are removed, you should reduce the flow chart to a series of subroutine calls. Although it seems pretty silly to do this for a simple program like our "SET X EQUAL TO ZERO, ADD ONE, CHECK IF LESS THAN 20" program, this method comes into its own with more complex programs.

Programming in Modules

As you've seen in many programs in this book, I'm in favour of you starting your program with a series of subroutine calls, with each action of the program being looked after by a separate subroutine. Then, if the steps within a program have to be performed several times in a particular sequence, the series of subroutine calls can be cycled through — over and over again — until a particular condition is met which signals the end of the run.

You'll recognize how useful this approach to programming can be when you get to the debugging stage of your program. If there is a bug, it is likely to be within a single subroutine, so it will be relatively easy to pin down the subroutine which contains the bug, rather than having to work right through the program trying to track it down.

Working with subroutine 'modules' in this way allows you to *test* sections of the program in isolation, even before the entire program is finished. I'll

try to make this statement clear by showing you the first part of a typical program to play Draughts:

```
10 ' Draughts
20 GOSUB 9000: ' Initialise
30 GOSUB 8000: ' Print Board
40 GOSUB 7000: ' Accept Player Move
50 GOSUB 8000: ' Print Board
60 GOSUB 5000: ' Amstrad Makes Move
70 IF (human NOT won) AND (computer NOT
     won) THEN 30
80 IF (computer has won) THEN
     PRINT "I win"
90 IF (human has won) THEN
     PRINT "You win"
100 END
```

You could have quite a bit of this program running, and tested (such as the initialisation routine, printing the board and accepting the player's move), before you even turned your attention to how on earth you were going to get the Amstrad to make its move.

You would then know — for example — that you would not need to waste any extra thought on whether or not an error in the board-printing routine was the cause of odd output. Having tested the board subroutine and the player move routine, you'd know that the error must be in the subroutine between lines 5000 and 6999, the subroutine in which the computer makes its move.

All you need to do is put a single PRINT statement, such as "THIS IS COMPUTER MAKING A MOVE" followed by a RETURN for incomplete subroutines, knowing that the program will accept that, and demonstrate the direction the program flow is following, even if whole sections of code have not yet been written. In general, I'd advise you to use this system of using a 'master loop' of subroutine calls within which you will 'hold' the entire program.

It took me a while to learn this lesson. Although I had read suggestions along the lines of 'work out exactly what you're going to enter *before* you start at the computer' in several books, I tended to just jump right in without much prior thought.

Although I worked out rough flow charts, and had an idea what sort of

display organisation I wanted, I certainly did not write much program out on paper before starting at the computer. Then, I once found myself stuck for a two-week period without my Amstrad and ideas for several programs just itching to be written. I had to write them out in an exercise book.

The relative ease with which the programs were debugged when they were eventually entered into the computer, and the complexity of the programs I wrote in this way (including my first chess program) convinced me that this was the way I would work from then on. It is amazing how much cleaner a program can be if all the rough working out is done on paper, rather than on the computer screen.

When working on programs, I tend to write the major 'call subroutines loop' first of all, but without line numbers, so the program contains lines like GOSUB PRINT BOARD and GOSUB INITIALISE. Next, I write each module (or subroutine) on a separate sheet of paper. Then, when the major subroutines have been written I shuffle them into an order which seems most logical.

All this, of course, occurs before any line numbers are written in. The subroutine modules are put in an order which ensures that the program structure is as logical as possible. I use arrows to indicate the destination of GOTO's within a module, and *names* for subroutines, as suggested in the major loop. Later on, when the program has started to assume a firm shape, the lines are numbered (I always work in tens, starting at line 10) and the relevant GOTO and GOSUB destingations are added.

All programs have an 'end condition' at which point computation stops. It is worth putting a test for this end condition before the WEND which sends the program back to start cycling through the major subroutine loop. This ensures that the cycle will continue until a particular condition is met, at which point the program will jump over the WEND and continue on to the lines which signal the end of the program.

Explicit Input Prompts

It is very useful, when writing a program, to keep in mind how the program will appear to a stranger when it is run by him or her for the first time. If there is an *input prompt* required, it is far more useful if the program prints up something like "HOW MANY HOURS HAS THE EMPLOYEE WORKED THIS WEEK?" instead of just "INSERT HOURS?" or the almost useless lone question mark.

The same suggestion applies to print output. It is far better that your program is written so that it prints out THE NUMBER OF HOURS WORKED ON FULL PAY THIS WEEK IS 27, rather than HOURS, FULL: 27 or an unsupported 27. Of course, providing explicit input prompts and output PRINT statements consumes memory as well as typing time when entering the program, but the contribution they make to the final program means the trouble involved is well worth it.

REM Statements

While exact PRINT statements and input prompts will help a person running a program make sense of it, REM statements can help make the program clear to those who are examining the listing for the first time. REM statements (which are, as you know, ignored by the computer when a program is run) should be used to help illuminate the flow of logic within the program, and what is happening in certain places within it. This is especially important in parts of the program where decisions are made, or calculations are carried out.

Not only can REM statements be used to explain different parts of the code, but they can also be used to provide visual 'breaks', so that the various blocks of code which carry out certain tasks are visually separated from the rest of the program. A blank REM statement (the word REM standing alone in a program line) can be used for this. A row of hyphens is an effective alternative.

Variables

It is worth considering the use of explicit names for variables, using either the word in full (such as HOURS as a variable name in a payroll program for hours worked) or an abbreviated version of this (such as HR) which has a fairly obvious meaning. You'll discover that this makes it easy, when working on a program, to keep track of your variables. This will help you with the initial debugging and later on as well if you need to improve or extend the original program.

Explicit variable names also help make your code more 'transparent' so other programmers can work out what the various parts of the program are intended to do. You'll also fine it a great help to yourself when you return to the program at a later date. It is surprising how code which seemed incredibly clear in terms of its purpose when you entered it, becomes exceedingly dense when you return to it after a long break.

Checking Input

Any input entered by the user should be checked by the program before it is accepted to ensure that incorrect data does not cause the program to crash at some future point. Whether you want string, or numeric input, it is often wise to allow for string input, which is first checked to see whether the entered material is acceptable and then, if necessary, the string can be changed into a number.

For example, if the user needs to enter a number between one and nine, a string can be accepted and then checked to ensure that it is not less than "1" and more than "9" before being changed into a number with a command like VAL.

As well as ensuring that the program will reject invalid input, you should check the program to see all the inputs which it does accept produce sensible answers when processed later on in the program. For example, make sure that your program does not accept zero as a possible number if the computer must later divide by this number.

Similarly, if numbers are to be processed by a function, and then the result of this processing used for division, you must check that an apparently valid input does not turn into zero as a result of evaluation by the function.

If the information entered by the user is rejected, and a new input is requested, the program should ideally point out why the original input was not suitable, or spell out again exactly what is required (such as "ENTER A NUMBER BETWEEN 1 AND 4"). You risk making users angry if input which appears valid is continually rejected without apparent reason.

Documentation

The written material which accompanies a program is often called documentation. It is useful for a program to be supported by some documentation, however sketchy.

The written information should explain, of course, what the program does, then go on to outline the flow of action within the program. The documentation should alert the user as to the kind of actions which will be required from him or her when running the program, and give an indication of the kinds of user input and reation which will be accepted.

The format of the final output should also be discussed. A list of variable names can be included.

If there are ways in which the program can be developed, extended or improved, suggestions along these lines can be added to the documentation. Written references to any material which will help in understanding the algorithms used, or for giving suggested areas for program development, should also be included.

In many ways, it is reasonable to assume that the job of programming is not finished once the program is done. Without documentation, the job is only three-quarters complete. Documentation finishes the task, adding a professional stamp to your work which allows the program you've written to be used most effectively.

Possibly the only time extensive documentation is not really required is when the program is 'menu-driven'. A program which uses REM statements extensively may not need very much in the way of documentation, especially if you can include a variable list, as a series of REM statements, at the end of the program.

Generally, however, you'll find it better to document a program externally, rather than rely on the REM statements, or the menu choices, to do the job for you. It is worth trying to write your program documentation so that it would make sense to someone who has not seen the program running.

This person should be able to get a very good idea, just from reading the documentation, of what the program does and how it does it; how it interacts with the user both in terms of accepting information and in presenting the results of its computations to the users; and how the program is organised as a whole.

Documentation for a major program should start with an introduction which quickly explains what is going on, and tells how to use the program. The later parts of the documentation can then discuss the program in greater detail. It is not good practice to force the user to wade through a vast amount of information in order to dig out the vital facts he or she needs to get the program running.

Section Eight
Utility and Practical Programs

Many computers are launched without significant software support from the manufacturer. Commercial software comapnies watch the launch of a new computer, trying to gauge when there will be enough machines in the marketplace to justify spending time developing programs, whether they be games or business application programs. In the meantime, while waiting for programs which suit your needs to be developed, you can either adapt existing published programs from books or magazines, or write your own material from scrach. It is likely you may well start by adapting programs before moving on to writing your own.

If you know your business is unusual, and that a specific program would be very useful, it may well be worth the trouble and expense of hiring a free-lance programmer to create a program for you or modify a program which is currently available. Otherwise, books and magazines will be among your program sources.

There are a number of things to keep in mind when you decide you'd like to buy business software for your Amstrad. You may be lucky enough to find exactly the program you need, which simply has to be loaded in and then run. However, a program which is tightly locked to your present method of doing business may prevent you from changing and developing your method of operation if the need arises.

Despite any claims you see in the advertising of programs, it is improbable that exactly the right program for your present and future needs exists ready for purchase off the shelf. You must be prepared to work on the program yourself to some extent.

Several companies are developing programs which are open enough to be tailored for a number of applications but are still written tightly enough to be of real use. You'll find these advertised and reviewed in the computing magazines.

What Can You Buy?

The following programs are among those which are available for small business use:

— payroll
— inventory
— accounts receivable
— accounts payable
— general ledger accounting
— wordprocessing

You may well find you can buy a package which includes more than one of these programs in a single integrated form. Many of the spread sheet accounting programs have developed versions which can perform a number of related tasks, and you may find such a program is suitable for your needs. To help you evaluate the use of your computer in business, our program *Minicalc,* which follows this discussion, not only performs a useful forecasting task by itself, but also will give you hands-on experience in working with such programs.

The whole field of budgetary control and forecasting is growing and is one of the areas where computers find most ready acceptance, because their value is so easy to appreciate. Using percentages and ratios for the analysis of financial statements is a growing field, and a program to assist with this may prove very valuable.

Minicalc

The *Minicalc* program, which can be very useful for extrapolating trends, allows you a permanent hard copy of its output. You can, however, use it so the results are just shown on the screen. You are given a choice at the start of each RUN. *Minicalc* offers one of the facilities provided by spread sheet programs.

If you have any stream of data which represents returns of events occurring in sequence, and which appear to indicate a fairly steady development, you'll find applications for this program.

You could, for example, plot the cost of running a car over a two year period and, assuming you kept the same car (and did not do something radical to it like having an accident or replacing the motor), predict with some certainty the running costs for the following year. Car repairs tend to

follow a trend which could be characterised by slowly rising costs, partly due to inflation and partly to the increasing age of the car.

Similarly, the number of rejects on a production line, with constantly improving quality control earlier in the production process, should lead to a gradually decreasing rejection rate. Entering known figures for rejects into *Minicalc* could provide you with an indication of the reject rate for three, six and nine months ahead, assuming your quality control improvement continues. You may well find that such things as the number of person hours lost due to industrial accidents in your plant shows a downward trend. *Minicalc* is ideal for producing a forward projection of this trend.

Many relationships can be extrapolated with this program, and so long as you do not run the projection too far into the future (watch out for absurd output with 'excess extrapolation'), you should find the information of value.

An example of excess extrapolation would be to enter the growth pattern in passenger use of a privately-owned bus service, until it exceeded the number of people in the area served by the buses, or exceeded the number of people in the area who did not have easy access to alternative means of transport.

To suggest that because your company showed a groww improvement in output of five per cent per month for the last six months that this growth pattern will continue month after month for five years is ludicrous. This would certainly be placing too much relience on a relatively short period of data collection.

Despite these cautionary examples, you'll still find *Minicalc* a valuable planning tool, especially if you use it to project for time periods which are similar to the time periods over which your entered data has been collected. That is, if you have sales figures from a particular territory for 12 months and you'd like to see how the next 12 months shape up, assuming gross factors remain much the same over the coming year as pertained during the year for which data is available, you could use *Minicalc* with some confidence. To project the next decade's figures from a single 12 months' return would not be wise.

However, even this long range forecast could be of benefit in highlighting, for example, the residual deterioration in sales from a certain territory. While a one per cent drop per month in sales over a six month period might

not seem too critical and could no doubt be blamed on external factors, projecting this for a further five years could highlight the seriousness of the problem.

For example, entering six months' sales figures into the program (assuming the figures were 100 units, 99, 98, 97, 96 and 95) would show an average deterioration of 1.04%. Projecting this trend would show figures of 84 after 12 months, 74 after 24 months and 65 after 36 months — a fall-off of more than a third!

On the other side of the coin, the output of a growing trend can be a very encouraging source of good news. Assuming, for example, you projected future days lost through strike action, after you have followed a year-long process of improving management/worker relations, and entered figures for the last four quarters of 145 hours, 136, 122 and 104 lost, you'd find that if the trend continued over the next four quarters you'd only lose 91 man hours, 80, 71 and 62 respectively. Even if you doubt the reliability of a straight line projection of this type, you will probably agree that at the very least it gives additional information with which to make management decisions and, even if limited, this can be of value.

Although the program listing and output refers to time periods called 'months', it can obviously be altered or taken to refer to any time period you desire — from nanoseconds to years.

```
10 '      MINICALC
20 '
30 CLS
40 INK 1,0:INK 0,23
50 PRINT:PRINT TAB(16);"MINICALC"
60 PRINT TAB(16);"========"
70 GOSUB 870
80 PRINT "ENTER NUMBER OF MONTHS FOR W
HICH FIGURESARE AVAILABLE"
90 INPUT M:IF M<2 THEN 90
100 TT=0
110 CLS
120 DIM A(M),B(M)
```

```
130 IF Z=1 THEN P=8 ELSE P=0
140 PRINT #P,"RECORDED FIGURES:"
150 FOR A=1 TO M
160 PRINT "ENTER FIGURE FOR MONTH";A
170 INPUT A(A)
180 PRINT #P,"MONTH:";A;A(A)
190 TT=TT+A(A)
200 NEXT
210 AV=TT/M
220 FOR B=2 TO M
230 B(B)=(100-(A(B-1)*100/A(B)))
240 NEXT
250 CLS
260 PRINT
270 PRINT #P," ":PRINT #P,"___________
____________________________":PRINT #P,
" "
280 PRINT £P,"DIFFERENCE BETWEEN MONTH
S:"
290 FOR A=2 TO M
300 PRINT #P,"MONTHS ";A-1;" - ";A;INT
(100*B(A))/100;"%"
310 NEXT
320 FOR T=1 TO 2500:NEXT
330 TT=0
340 FOR A=2 TO M
350 TT=TT+B(A)
360 NEXT
370 AVE=INT(TT*100/(M-1))/100
380 CLS
390 PRINT £P," ":PRINT #P,"___________
____________________________":PRINT #P,
" "
```

```
400 AVE=INT(100*AVE)/100
410 PRINT #P,"AVERAGE CHANGE: ";AVE;"%
"
420 FOR T=1 TO 2500:NEXT
430 CLS
440 PRINT £P," ":PRINT #P,"___________
____________________________":PRINT #P,
" "
450 PRINT "NOW HERE'S A PROJECTION OF
CHANGE:"
460 IF Z=1 THEN PRINT #8,"          PR
OJECTION OF CHANGE"
470 FOR T=1 TO 1500:NEXT
480 PRINT:PRINT "HOW MANY MONTHS PROJE
CTION WOULD YOU    LIKE"
490 INPUT NU
500 CLS
510 PRINT "FINAL MONTH:";A(M)
520 PRINT "AVERAGE PER MONTH:";AV
530 IF Z<>1 THEN 560
540 PRINT #8,"AVERAGE PER MONTH";AV
550 PRINT #8,"FINAL MONTH:";A(M)
560 FOR T=1 TO 1500:NEXT
570 PRINT "DO YOU WANT A PROJECTION BA
SED UPON"
580 PRINT "  1  - THE FINAL MONTH...OR
"
590 PRINT "  2  - THE AVERAGE PER MONT
H"
600 A$=INKEY$
610 IF A$<>"1" AND A$<>"2" THEN 600
```

```
620 D=VAL(A$)
630 CLS
640 PRINT #P," "
650 E=(A(M) AND D=1)+(AV AND D=2)
660 PRINT #P,"MONTH 1 - RECORDED ";A(M
)
670 FOR A=2 TO NU
680 E=E+AVE*E/100
690 PRINT #P,"MONTH:";A;" - ";INT(E)
700 FOR T=1 TO 500:NEXT
710 NEXT
720 PRINT #P," ":PRINT #P,"___________
___________________________":PRINT #P,
" "
730 FOR T=1 TO 1500:NEXT
740 PRINT:PRINT
750 PRINT "        1 - PROJECTION AGAIN
"
760 PRINT "        2 - OUTPUT AGAIN"
770 PRINT "        3 - START AGAIN"
780 PRINT "        4 - TO END"
790 IF INKEY$<>"" THEN 790
800 A$=INKEY$
810 IF A$="" THEN 800
820 IF A$="1" THEN 430
830 IF A$="2" THEN 250
840 IF A$="3" THEN RUN
850 IF A$="4" THEN CLS:PRINT:PRINT:PRI
NT "O.K.":END
860 GOTO 790
870 PRINT:PRINT
880 PRINT "ENTER 1 IF YOU WANT A COPY
ON YOUR      PRINTER, 2 IF NOT"
```

```
890 A$=INKEY$
900 IF A$<>"1" AND A$<>"2" THEN 890
910 Z=VAL(A$)
920 PRINT #P," ":PRINT #P,"___________
____________________________":PRINT #P,
" "
930 CLS
940 RETURN
```

Here is a sample output of the program:

```
RECORDED FIGURES:
MONTH:  1   2902
MONTH:  2   2897
MONTH:  3   2876
MONTH:  4   2680
MONTH:  5   2640
MONTH:  6   2612
MONTH:  7   3200
MONTH:  8   3253
MONTH:  9   2800
MONTH:  10   2830
MONTH:  11   2720
MONTH:  12   2650

---------------------------------------

DIFFERENCE BETWEEN MONTHS:
MONTHS   1   -   2 -0.18 %
MONTHS   2   -   3 -0.74 %
MONTHS   3   -   4 -7.32 %
```

```
MONTHS  4  -  5 -1.52 %
MONTHS  5  -  6 -1.08 %
MONTHS  6  -  7  18.37 %
MONTHS  7  -  8  1.62 %
MONTHS  8  -  9 -16.18 %
MONTHS  9  -  10  1.06 %
MONTHS  10  -  11 -4.05 %
MONTHS  11  -  12 -2.65 %

----------------------------------------

AVERAGE CHANGE: -1.15 %

----------------------------------------

          PROJECTION OF CHANGE
AVERAGE PER MONTH 2838.33333
FINAL MONTH: 2650

MONTH 1 - RECORDED  2650
MONTH: 2  -  2805
MONTH: 3  -  2773
MONTH: 4  -  2741
MONTH: 5  -  2709
MONTH: 6  -  2678
MONTH: 7  -  2647
MONTH: 8  -  2617
MONTH: 9  -  2587
MONTH: 10  -  2557
MONTH: 11  -  2528
MONTH: 12  -  2498

----------------------------------------
```

Personal Finance

Our next program — *Personal Finance* — will not only prove useful in its own right, but your experience working with it should aid you in evaluating business software for your own needs. In fact, you may well be able to modify it to work as a budgetary control program for a small business, or learn enough from starting and running it to go on to develop a financial planning program of your own.

The program is designed to aid you both to blance your cheque book, and to project future recurrent costs against your income. If you have, for example, to pay $399.50 per month on your housing loan, $250 per month on payments on your boat, and $22.56 for newspaper deliveries, you can enter these costs, plus current balance, and get the Amstrad to subtract the total cost from the balance. The program will accept deposits from you, and keep you informed of the running balance. It's designed to lead you step-by-step through looking at your bank account balance.

When you first run the program, you'll be presented with a menu of five choices:

Key Action

1 — To start again.
2 — Deposit.
3 — Modify schedule.
4 — Stop.
5 — Print data.

When you run the program, press the '1' key as the first option. 'Enter number of items to be paid for each month' is the next instruction you'll be given. It is best to be generous here, perhaps entering a higher number than the maximum number of items you think you'll need because although you can leave some items blank, you cannot add additional items once the program is running (although you can easily change an item — even a blank one — into another item).

```
10 '   PERSONAL FINANCE
20 '
```

```
30 CLS
40 BAL=0
50 GOTO 700
60 PRINT "ENTER NUMBER OF ITEMS TO BE
PAID FOR    EACH MONTH"
70 INPUT NUM
80 CLS
90 DIM A$(NUM),A(NUM)
100 FOR A=1 TO NUM
110 PRINT "ENTER NAME OF ITEM NUMBER";
A
120 INPUT A$(A)
130 IF LEN(A$(A))<2 THEN 120
150 INPUT "EACH MONTH";A(A)
160 FOR Z=1 TO A
170 PRINT A$(Z);" £";A(Z)
180 NEXT Z
190 NEXT A
200 PRINT:PRINT:PRINT:PRINT
210 PRINT "ENTER X IF THIS IS CORRECT"
220 PRINT " ''   C TO CHANGE ONE"
230 PRINT " ''   A TO SEE THE LIST AGA
IN"
240 PRINT:PRINT
250 B$=INKEY$
260 IF B$<>"X" AND B$<>"C" AND B$<>"A"
 THEN 250
270 IF B$="X" THEN 420
280 IF B$="A" THEN PRINT:FOR Z=1 TO NU
M:PRINT A$(Z);" £";A(Z):NEXT:GOTO 200
290 PRINT "ENTER NUMBER OF ITEM YOU WI
SH TO CHANGE"
300 INPUT B
```

```
310 PRINT:PRINT
320 PRINT "ENTER NEW NAME FOR ITEM NUM
BER";B
330 INPUT A$(B)
340 PRINT:PRINT
350 PRINT "AND HOW MUCH IS ";A$(B)
360 INPUT "EACH MONTH";A(B)
370 CLS
380 FOR A=1 TO NUM
390 PRINT A$(A);" - £";A(A)
400 NEXT
410 GOTO 200
420 CLS
430 IF BAL<>0 THEN 460
440 PRINT "ENTER LAST KNOWN BALANCE"
450 INPUT BAL
460 CLS
470 PRINT "£";BAL
480 PRINT:PRINT "ENTER DEPOSITS ONE BY
 ONE"
490 PRINT "      AND END WITH 'E'"
500 INPUT Q$
510 IF Q$="" THEN 500
520 IF Q$="E" THEN 550
530 BAL=BAL+VAL(Q$)
540 GOTO 470
550 D=BAL
560 PRINT "ENTER 1 TO SUBTRACT ALL OF"
570 PRINT "PAYMENTS OR 2 FOR THE MENU"
580 C$=INKEY$
590 IF C$<>"1" AND C$<>"2" THEN 580
600 C=VAL(C$)
```

```
610 IF C=2 THEN 730
620 SD=0
630 FOR A=1 TO NUM
640 SD=SD+A(A)
650 NEXT
660 PRINT:PRINT
670 PRINT "COST SCHEDULE TOTALS £";SD
680 PRINT "BALANCE BEFORE CURRENT COST
S £";BAL
690 BAL=BAL-SD
700 PRINT "CURRENT BALANCE £";BAL
710 IF BAL<0 THEN PRINT:PRINT "OVERDRA
WN!"
720 FOR T=1 TO 500:NEXT
730 PRINT:PRINT:PRINT
740 PRINT "1 - TO START AGAIN"
750 PRINT "2 - DEPOSIT"
760 PRINT "3 - MODIFY SCHEDULE"
770 PRINT "4 - TO END
780 PRINT "5 - DATA TO PRINTER"
790 IF INKEY$<>"" THEN 790
800 C$=INKEY$
810 IF C$<>"1" AND C$<>"2" AND C$<>"3"
 AND C$<>"4" AND C$<>"5" THEN 800
820 PRINT:PRINT:PRINT
830 C=VAL(C$)
840 IF C=1 THEN CLEAR:GOTO 60
850 IF C=2 THEN 470
860 IF C=3 THEN 370
870 IF C=4 THEN PRINT:PRINT "THANK YOU
":END
880 IF C=5 THEN 900
890 GOTO 790
```

```
900 ' PRINTER OUTPUT
910 WIDTH 40
920 PRINT #8,TAB(12);"PERSONAL FINANCE
"
930 PRINT #8,TAB(12);"================
"
940 PRINT #8," ":PRINT #8," "
950 FOR T=1 TO NUM
960 PRINT #8,A$(T);TAB(30);"£";A(T)
970 NEXT T
980 PRINT #8," "
990 PRINT #8,"BALANCE BEFORE CURRENT C
OSTS  £";D
1000 PRINT #8,"COST SCHEDULE TOTALS  £
";SD
1010 PRINT #8,"CURRENT BALANCE AFTER D
EDUCTIONS £";D-SD
1020 GOTO 700
```

You'll find that, once the program is underway, it leads you neatly, step-by-step, through the required operating procedure. Once you've read through the sample run and worked out what is going on, enter the program and try running it in its present form. Then you can try your hand at modifying it to make it more useful for your own needs.

```
RENT                    £ 120
HEATING                 £ 43.4
ELECTRICITY             £ 29.9
CLOTHES                 £ 24
H.P. PAYMENTS           £ 17.65
TRAVELLING              £ 32
MISCELLANEOUS           £ 15
ENTERTAINMENT           £ 39
```

```
BALANCE BEFORE CURRENT COSTS  £ 449.91
COST SCHEDULE TOTALS  £ 320.95
CURRENT BALANCE AFTER DEDUCTIONS £ 128.96
```

Other Applications

Here are a couple of small programs that use a central formula to work out various financial calculations.

The first program calculates the percentage change in a value. This has obvious uses for selling items second-hand, share prices and so on.

```
10 '    PERCENTAGE CHANGE
20 '
30 INPUT "CURRENT WORTH";C
40 INPUT "PERCENTAGE CHANGE";P
50 PRINT "INCREASE [I] OR"
60 INPUT "DECREASE [D]";D$
70 IF D$="D" THEN P=-P
80 N=(1+P/100)*C
90 PRINT:PRINT
100 PRINT "THE NEW FIGURE IS";N
```

Straight line depreciation is a concept often found in business life. It refers to the method of writing off an item on the balance sheet over a number of years. This eay-to-use program calculates the depreciation over a specified time period; though the computer uses years, the time scale can be easily changed if needs demand.

```
30 Y=0
40 CLS
50 INPUT "ENTER PURCHASE PRICE";P
60 PRINT
70 PRINT "PURCHASE PRICE IS  £";P
80 PRINT
```

```
90 INPUT "ENTER FINANCIAL LIFE OF ASSE
T (IN YEARS)";L
100 PRINT
110 PRINT "FINANCIAL LIFE OF ASSET IS"
;L;"YEARS"
120 PRINT
130 D=(INT(P*100/L))/100
140 PRINT "IT DEPRECIATES ";D;" POUNDS
 A YEAR"
150 PRINT
160 INPUT "ENTER FIRST YEAR OF USE (AS
 IN 1984)   ";Y
170 PRINT:PRINT
180 PRINT " YEAR";TAB(12);"WORTH"
190 PRINT " ====================="
200 PRINT Y;TAB(11);"£";FIX(P)
210 P=P-D
220 IF P<1 THEN END
230 Y=Y+1
240 GOTO 200
```

Repayments on Mortgage

Probably the biggest sum of money you will ever borrow will be used to buy your house. The formula used takes into account that the early repayments are almost entirely repaying interest, while the later ones are repaying more principal than interest.

This program will tell you what repayments should be on a housing loan, and also how much you will pay back altogether. This final figure is, however, somewhat depressing.

```
10 '  HOUSING LOAN
20 '
30 INPUT "PRINCIPAL ";P
```

```
40 INPUT "TIME ";T
50 INPUT "ANNUAL INTEREST RATE";R
60 R=R/100
70 REP=((1+R)^T)*R*P/(((1+R)^T)-1)
80 REP=INT(REP*100)/100
90 PRINT:PRINT:PRINT
100 PRINT "ANNUAL REPAYMENT IS £";REP
110 PRINT:PRINT "MONTHLY REPAYMENT IS
 £";INT(REP*8.33)/100
120 PRINT:PRINT "TOTAL TO BE REPAID IS
 £";REP*T
```

Section Nine
Crackers and Corkers

This section of the book is made up of lots of odds and ends. Although they didn't fit easily into any other section, they struck me as interesting and important enough to be included, even if it meant we had a bit of a muddle.

Keys

To disenable the BREAK key, include a pair of lines like these:

```
10 ON BREAK GOSUB 9500

9500 RUN
```

This ensures that on the second push of the BREAK key, the program simply runs again.

Opening Frames

Use this program to get a line of enlarged text (including user-defined graphics if you like) at the top of the screen for an opening frame. Put your message within the quotes in line 50:

```
10 REM LARGE WRITING
20 REM FOR TITLE AT TOP OF SCREEN
30 DEFINT A-Z
40 MODE 1:INK 0,0:BORDER 0:INK 1,0:INK 2
,24:INK 3,6
50 PEN 1:LOCATE 1,25:PRINT "Your message
 here":REM Up to 20 characters
60 FOR A=0 TO 300 STEP 2
70 FOR B=0 TO 16 STEP 2
80 IF TEST(A,B)=0 THEN 130
90 PLOT 10+2*A,365+2*B,2
```

```
100 PLOT 10+2*A,367+2*B,3
110 PLOT 12+2*A,365+2*B,2
120 PLOT 12+2*A,367+2*B,3
130 NEXT B
140 NEXT A
150 LOCATE 1,25:PRINT SPC(39):INK 1,26
160 LOCATE 1,22
170 GOTO 170
```

Here is a sample loading screen, which you can modify to appear on the screen while the major program is loading:

```
10 REM Sample Loading Screen
20 GOSUB 140
30 LOCATE 1,6
40 PEN 1:PRINT "      Interface Publicati
ons Ltd"
50 PEN 2:PRINT:PRINT "              prese
nts"
60 PEN 3:PRINT:PRINT "         THE WORLD
THINKER"
70 PEN 2:PRINT:PRINT "           An adven
ture set                    in deepest Sh
epherds Bush"
80 PRINT:PRINT:PRINT
90 PEN 1:PRINT:PRINT "     Copyright Tim
 Hartnell 1986"
100 LOCATE 1,25:PEN 3:PRINT"    Loading
now..."
110 CHAIN"!filename"
120 END
130 REM ----------------------
140 REM Initialise
150 MODE 1
160 INK 0,0:INK 1,24:INK 2,6:INK 3,1,2:B
ORDER 0
170 RETURN
```

Frame

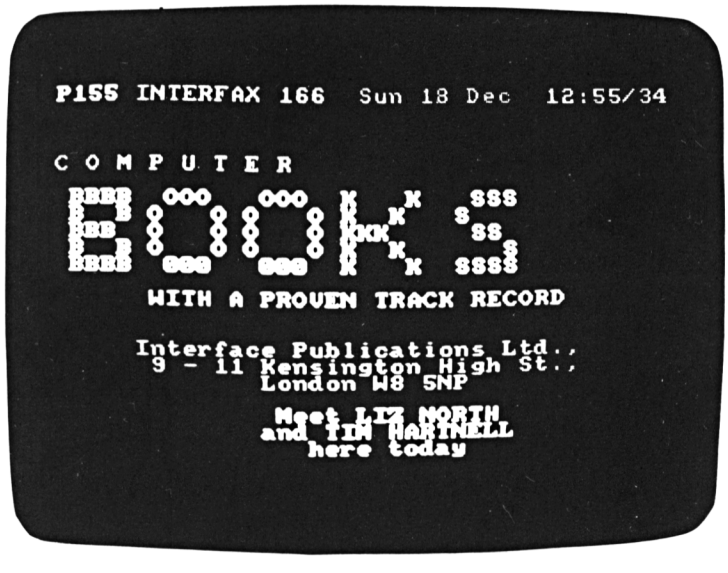
Use this next program to produce a 'frame' on the screen for displaying the score:

```
10 REM LINE FRAME ON SCREEN
20 REM   FOR SCORE DISPLAY
```

```
30 MODE 1:INK 0,0:BORDER 0:INK 1,6:INK 2
,24:INK 3,2
40 MOVE 0,0
50 DRAW 0,399,2
60 DRAW 639,399,2
70 DRAW 638,0
80 DRAW 0,0
90 MOVE 0,320
100 DRAW 639,320,3
110 GOTO 110
```

D.I.Y. Teletext

This program allows your Amstrad to produce imitiation teletext screens, somewhat like those shown on *Ceefax* or *Oracle.* This program, in its present form, was used at the Interface Publications stand at the London



Book Fair. BBC Scotland were so intrigued with it, they filmed the program in action for their television show *Imprint.* The four 'pictures' should, of course, be changed to your own screens:

```
10 REM D.I.Y. TELETEXT
20 GOSUB 1600:REM INITIALISE
30 CLS
40 LOCATE 1,1:PRINT "P152"
50 WHILE NOT CEEFAXING
60 LOCATE 6,1
```

```
70 PEN 3:PRINT "INTERFAX";PAGENUMBER;" "
;date$;"  ";
80 h$=STR$(hr):m$=STR$(mn):s$=STR$(sc)
90 h$=MID$(h$,2)
100 m$=MID$(m$,2):IF mn<10 THEN m$="0"+m
$
110 s$=MID$(s$,2):IF sc<10 THEN s$="0"+s
$
120 PEN 2:PRINT h$;":";m$;"/";s$;" "
130 SC=INT((TIME-M)/300)
140 IF sc=10 OR sc=24 OR sc=39 OR sc=52
THEN GOSUB 280
150 IF SC>59 THEN MN=MN+1:M=TIME
160 IF MN>59 THEN MN=0:HR=HR+1
170 IF HR>13 THEN HR=1
180 pagenumber=pagenumber+1
190 IF pagenumber=200 THEN pagenumber=70
0
200 IF pagenumber=720 THEN pagenumber=10
0
210 WEND

220 REM ----------------------
230 REM CHANGE COLOUR AT RANDOM
240 col=INT(RND*3)+1
250 PEN col
260 RETURN
270 END
280 REM pages
290 count=count+1
300 IF count=5 THEN count=1
310 flag=0:IF RND>0.8 THEN flag=count:co
unt=2
320 IF SC>59 THEN MN=MN+1:M=TIME
330 ON count GOSUB 380,670,980,1320
340 IF flag<>0 THEN count=flag-1
350 IF SC>59 THEN MN=MN+1:M=TIME
360 RETURN
370 REM ****************************
380 REM picture one
390 LOCATE 1,1:PEN 1:PRINT "P";MID$(STR$
(pagenumber),2)
400 LOCATE 1,2
```

```
410 PRINT "

          "
420 PEN 3:PRINT "C O M P U T E R

                  "
430 PEN 1
440 PRINT " BBBB  OOO   OOO  K   K   SSS
          "
450 PRINT " B  B O   O O   O K  K   S
          "
460 PRINT " BBB  O   O O   O KKK     SS
          "
470 PRINT " B  B O   O O   O K  K      S
          "
480 PRINT " BBBB  000   000  K   K  SSSS
          "
490 PRINT "
           "
500 PEN 2
510 PRINT "      WITH A PROVEN TRACK REC
ORD       "
520 PRINT "
          "
530 PRINT "
           "
540 PEN 3
550 PRINT "     Interface Publications L
td.,      "
560 PRINT "      9 - 11 Kensington High
St.,      "
570 PRINT "             London W8 5NP
          "
580 PRINT "
          "
590 PEN 1
600 PRINT "              Meet LIZ NORTH
          "
610 PRINT "              and TIM HARTNELL
          "
620 PRINT "                 here today
          "
```

```
630 PRINT "
          "
640 PRINT "
          "
650 RETURN
660 REM ****************************
670 REM picture two
680 LOCATE 1,1:PEN 1:PRINT "P";MID$(STR$
(pagenumber),2)
690 GOSUB 240
700 LOCATE 1,2
710 PRINT "
          "
720 PRINT "
          "
730 PRINT "New titles from Interface Pub
lications "
740 PRINT "
          "
750 GOSUB 240
760 PRINT "
       "
770 IF RND>0.5 THEN 790
780 IF RND>0.5 THEN PRINT "40 Best Machi
ne Code Routines for the   BBC Micro - I
an Hutt - #7.95" ELSE PRINT "How to Prog
ram your AMSTRAD like a      Professiona
l - #6.95                   ":GOTO 810
790 IF RND>0.5 THEN PRINT "MEGABASIC FOR
 THE COMMODORE 64 - #4.99" ELSE PRINT "A
dvanced Z80 Machine Code Programming "
800 GOSUB 240
810 PRINT "

          "
820 IF RND>0.5 THEN 840
830 IF RND>0.5 THEN PRINT "How to Progra
m your ATARI ST like a     Professional
- M. Fairbanks - #6.95":GOTO 850
840 IF RND>0.5 THEN PRINT "SPECTRUM+ LOG
O - Tim Hartnell - #2.99  " ELSE PRINT "
Creating Arcade Games on the
Commodore 64 - R Woolford - #7.95       "
```

```
850 PRINT "

          "
860 IF RND>0.5 THEN PRINT "THE AMSTRAD A
DVENTURE CREATOR - #2.99" ELSE PRINT "Re
plicating Reality: Exploring           Co
mputer Simulations                    "
870 GOSUB 240
880 PRINT "

          "
890 GOSUB 240
900 PRINT "EXPLORING EXPERT SYSTEMS ON Y
OUR      "
910 PRINT " MICROCOMPUTER - #7.95
          "
920 PRINT "

          "
930 PRINT "

          "
940 GOSUB 240
950 IF RND>0.6 THEN PRINT " Computer boo
ks with a track record..." ELSE PRINT "-
-------------------------------------"
960 RETURN
970 REM ****************************
980 REM picture three
990 LOCATE 1,1:PEN 1:PRINT "P";MID$(STR$
(pagenumber),2)
1000 LOCATE 1,2
1010 PRINT "
          "
1020 PRINT "
          "
1030 GOSUB 240
1040 PRINT "Books published by Interface
          "
1050 PRINT " are distributed in the UK
          "
1060 PRINT "  by W.H.S. DISTRIBUTORS
          "
```

```
1070 PRINT "
           "
1080 PRINT "
           "
1090 PRINT "
           "
1100 PRINT "  S P E C T R U M  . . . .
. . .      "
1110 GOSUB 240
1120 PRINT "
           "
1130 PRINT "    C O M M O D O R E   6 4
. .        "
1140 GOSUB 240
1150 PRINT "
           "
1160 GOSUB 240
1170 PRINT "      A M S T R A D . . . .
.          "
1180 GOSUB 240
1190 PRINT "
           "
1200 GOSUB 240
1210 PRINT "        B B C   M I C R O .
           "
1220 GOSUB 240
1230 PRINT "
           "
1240 GOSUB 240
1250 PRINT "          A T A R I   S T
           "
1260 PRINT "
           "
1270 GOSUB 240
1280 PRINT "            M S X . . .
           "
1290 PRINT "
           "
1300 RETURN
1310 REM ****************************
1320 REM picture four
1330 LOCATE 1,1:PEN 1:PRINT "P";MID$(STR
$(pagenumber),2)
```

```
1340 LOCATE 1,2
1350 PRINT "
           "
1360 PEN 2
1370 PRINT "######################
 ##########"
1380 PRINT "######################  ";:PE
N 1:PRINT " IP";:PEN 2:PRINT "   #######
##"
1390 PRINT "                     ####
####        "
1400 PRINT "                       ###     #
##          "
1410 PRINT "                       ###     #
##          "
1420 PRINT "                        ###   ##
#           "
1430 PRINT "                        ###   ##
#           "
1440 PRINT "                         ######
           "
1450 PRINT "                         ######
           "
1460 PRINT "                          ####
            "
1470 PRINT "                          ####
           "
1480 PEN 1:PRINT "  I N T E R F A C E
 ";:PEN 2:PRINT " ##              "
1490 PRINT "                            ##
           "
1500 PRINT "
           "
1510 PEN 1:PRINT "  P U B L I C A T I O
N S   L T D     "
1520 PRINT "
           "
1530 PRINT "
           "
1540 PEN 3
1550 PRINT "   W E ' R E
           "
1560 PRINT "               T H E
           "
```

```
1570 PRINT "                              E X P E
 R T S  ! "
1580 PRINT "----------------------------
---------    ----------------------------
---------  "

1590 RETURN
1600 REM INITIALISE
1610 BORDER 0
1620 INK 0,0:INK 1,26:INK 3,18:INK 2,24
1630 MODE 1
1640 INPUT "TIME IN HOURS";HR
1650 INPUT "TIME IN MINUTES";MN
1660 M=TIME
1670 INPUT "ENTER DAY DATE AS Fri 29 Mar
";date$
1680 PAGENUMBER=100
1690 RETURN
```

Generating Sound

The SOUND command is followed by up to seven numbers, as follows:

a — channel status
b — tone period
c — duration
d — volume
e — volume envelope
f — tone envelope
g — noise period

Only a and b above are compulsory; they have no default values.

There are three sound channels, which can be designated as X, Y and Z. The a number above must be between 1 and 255. A value of 1 will send SOUND to channel X, 2 will send it to Y and 4 will send it Z. To get it to X and Y at the same time, use 3; and 5 to X and Z at the same time. A value of 7 will get it to all three.

The tone period, which is also compulsory, must be in the range 0 to 4059. This determines the pitch of the note to be played. This chart may help.

NOTE	FREQUENCY	PERIOD
Middle C	261.262	478
C#	277.183	451
D	293.665	426
D#	311.127	402
E	329.628	379
F	349.228	358
F#	369.994	338
G	391.995	319
G#	415.305	301
Inter. A	440.000	284
A#	466.164	268
B	493.883	253
C	523.251	239
C#	554.365	225
D	587.330	213

D#	622.254	201
E	659.255	190
F	698.457	179
F#	739.989	169
G	783.991	159
G#	830.609	150
A	880.000	142
A#	932.328	134
B	987.767	127
C	1046.502	119
C#	1108.731	113
D	1174.659	106
D#	1244.508	100
E	1318.510	95

The duration, which defaults to 20, must be in the range -32768 to 32767. A positive number gives a duration measured in 1/100ths of a second; when set at 0, the length of time the note sounds depends on the volume envelope which has been defined; when set to a negative number, the unsigned value of this number determines how many times the volume envelope which has been defined should be repeated.

The volume ranges from 0 to 7 if a volume envelope has not been specified, and from 0 to 15 if one has been set. The default value is 4, with 0 meaning no sound at all.

The next value is the volume envelope, which needs to be in the range 0 to 15, defaulting to 0. It uses a pre-defined envelope (created with the ENV command).

The tone envelope comes next, and this too is an integer in the range 0 to 15, with a default of 0.

Finally, we have the noise period, which has the same range, 0 to 15, with 0 as the default.

To show the SOUND command in action, try the following on your computer. We start with a simple outburst of musical indignation:

```
10 REM OUTBURST
20 SOUND 2,91,49,14,0,1,1
```

Now, a single loop, and the earth is under attack:

```
10 REM ALIEN ATTACK
20 FOR J=1 TO 250
30 SOUND 132,RND*100,20,14
40 NEXT J
```

It's air raid time:

```
10 REM Musical siren
20 RANDOMIZE TIME
30 FOR j=1 TO 10
40 IF RND>0.2 THEN SOUND 1,284
50 IF RND>0.3 THEN SOUND 2,239
60 IF RND>0.2 THEN SOUND 4,190
70 IF RND>0.2 THEN SOUND 2,230
80 IF RND>0.2 THEN SOUND 1,282
90 SOUND 4,190
100 NEXT j
```

Beware the dreaded Undertoad:

```
10 REM Alert! Alert!
20 FOR j=1 TO 15
30 SOUND 1,480,5
40 SOUND 5,9,12,0
50 SOUND 1,522,5
60 NEXT j
```

The motor boat is up and off:

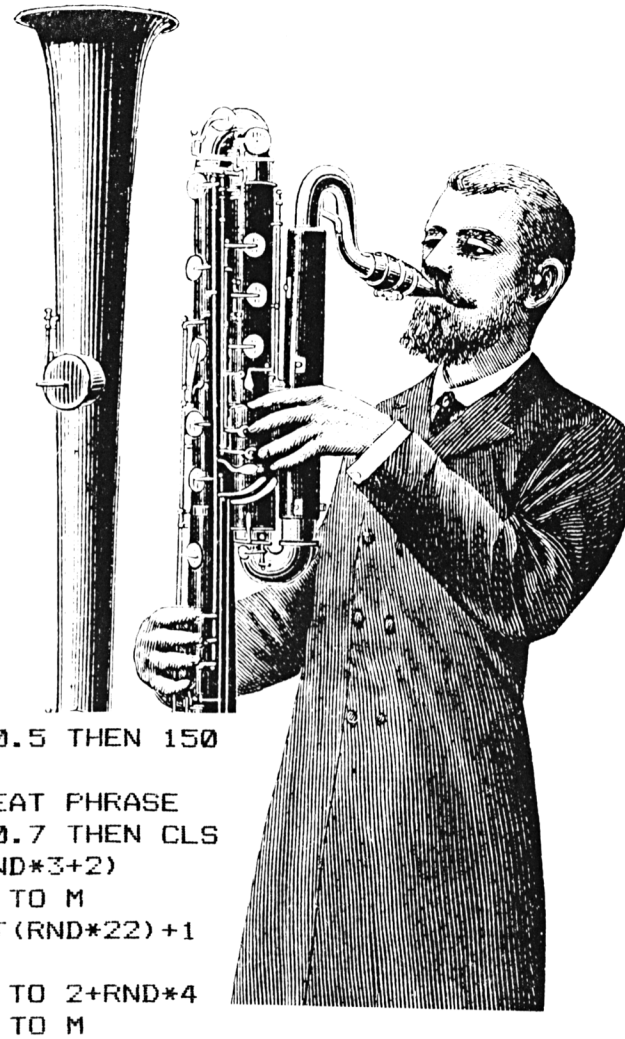
```
10 REM MOTOR BOAT
20 WHILE NOT FINISHED
30 SOUND 1,997.4,4,15,0,0,1
40 SOUND 2,3289,5,15,0,0,1
50 WEND
```

Down we go, and explode:

```
10 REM     Falling Sound
20 REM followed by explosion
30 FOR q=100 TO 200
40 SOUND 1,q,1
50 NEXT q
60 FOR q=1 TO 60
70 SOUND 1,RND(100),1,15,0,0,31
80 NEXT q
```

Finally, the Phantom Composer comes into its own:

```
10 REM THE PHANTOM COMPOSER
20 DIM A(22)
30 FOR J=1 TO 22
40 READ A(J)
50 NEXT J
60 Y=100
70 REM ------------
80 MODE 1:INK 0,0:BORDER 0:INK 1,6:INK 2
,24:INK 3,2
90 WHILE NOT FINISHED
100 NOTE=INT(RND*22)+1:IF RND>0.8 THEN N
OTE=3
110 IF RND>0.89 THEN GOSUB 200
120 DRAW RND*A(NOTE)+300,A(NOTE)/2,INT(R
ND*3)+1
130 DRAW A(NOTE)/2,RND*A(NOTE)+300,INT(R
ND*3)+1
140 DUR=120*INT(RND*4+1)
150 SOUND 132,A(NOTE),DUR,14
160 IF RND>0.9 THEN Y=INT(10*RND)*30+50
170 FOR K=1 TO Y:NEXT K
```



```
180 IF RND>0.5 THEN 150
190 WEND
200 REM REPEAT PHRASE
210 IF RND>0.7 THEN CLS
220 M=INT(RND*3+2)
230 FOR J=1 TO M
240 B(J)=INT(RND*22)+1
250 NEXT J
260 FOR Z=1 TO 2+RND*4
270 FOR J=1 TO M
280 SOUND 132,A(B(J)),DUR,14
290 DRAW RND*A(NOTE)+100,A(NOTE),INT(RND
*3)+1
300 FOR K=1 TO Y:NEXT K
310 NEXT J
320 FOR K=1 TO Y:NEXT K
330 NEXT Z
340 FOR K=1 TO Y:NEXT K
350 RETURN
360 DATA 0,0,478,379,319,239,239,478,190
,159,119,119,478,379,319,239,478,190,159
,119,358,426
```

Appendices

A — BASIC Keywords
B — Inks and Colours
C — ROM Routines
D — Error Messages
E — Key Numbering
F — User-Defined Graphics Forms

Appendix A
Basic Keywords

Here are the Amstrad's BASIC keywords. You can't use them for variables:

ABS, AFTER, AND, ASC, ATN, AUTO, BIN$, BORDER, CALL, CAT, CHAIN, CHR$, CINT, CLEAR, CLG, CLOSEIN, CLOSEOUT, CLS, CONT, COS, CREAL, DATA, DEF, DEFINT, DEFREAL, DEFSTR, DEG, DELETE, DI, DIM, DRAW, DRAWR, EDIT, EI ELSE, END, ENT, ENV, EOF, ERASE, ERL, ERR, ERROR, EVERY, EXP, FIX, FN, FOR, FRE, GOSUB, GOTO, HEX$, HIMEM, IF, INK, INKEY, INKEY$, INTP, INPUT, INSTR, INT, JOY, KEY, LEFT$, LEN, LET, LINE, LIST, LOAD, LOCATE, LOG, LOGI0, LOWER$, MAX, MEMORY, MERGE, MID$, MIN, MOD, MODE, MOVE, MOVER, NEXT, NEW, NOT, ON, ON BREAK, ON ERROR GOTO, ON, SQ, OPENIN, OPENOUT, OR, ORIGIN, OUT, PAPER, PEEK, PEN, PI, PLOT, PLOTR, POKE, POS, PRINT, RAD, RANDOMIZE, READ, RELEASE, REM, REMAIN, RENUM, RESTORE, RESUME, RETURN, RIGHT$, RND, ROUND, RUN, SAVE, SGN, SIN, SOUND, SPACE$, SP, SPEED, SQ, SQR, STEP, STOP, STR$, STRING$, SWAP, SYMBOL, TAB, TAG, TAGOFF, TAN, TEST, TESTR, THEN, TIME, TO, TROFF, TRON, UNT, UPPER$, USING, VAL, VPOS, WAIT, WEND, WHILE, WIDTH, WINDOW, WRITE, XOR, XPOS, YPOS and ZONE.

The above words are supported on both the 464 and the 664. In addition to these, the 664 supports the following:

AUTO, CLEAR INPUT, COPYCHR$, CUROSR, DEC$, DERR, DRAW and DRAWR (on the 664, this pair of commands, and the two pairs following, cater for an additional ink mode parameter, which allows control of the overplotting rules), MOVE and MOVER, PLOT and PLOTR, FILL, FRAME, GRAPHICS, MASK, ON BREAK CONT, PEN (provides an additional background mode parameter on the 664), PRINT USING, DIR, DRIVE, ERA and REN (these take normal string expressions on the 664, whereas a string must be passed by an address on the 464).

Appendix B
Inks and Colours

Grey Scale	Colour
0	Black
1	Blue
2	Bright blue
3	Red
4	Magenta
5	Mauve
6	Bright red
7	Purple
8	Bright magenta
9	Green
10	Cyan
11	Sky blue
12	Yellow
13	White
14	Pastel blue
15	Orange
16	Pink
17	Pastel magenta
18	Bright green
19	Sea green
20	Bright cyan
21	Lime
22	Pastel green
23	Pastel cyan
24	Bright yellow
25	Pastel yellow
26	Bright white

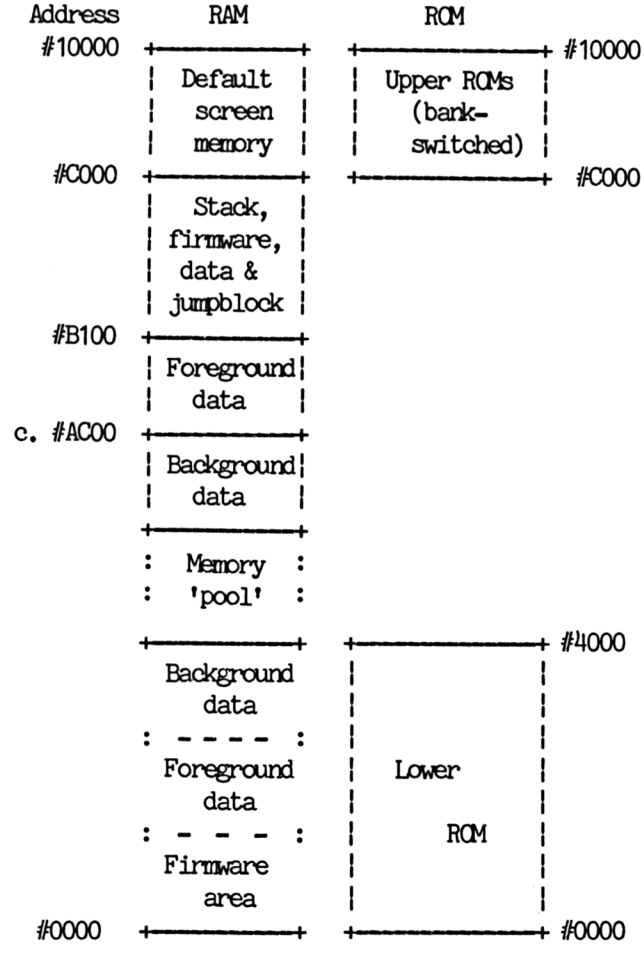
Default Colour Settings

Ink	Colour	Colour numbers
Border	Blue	1/1
0	Blue	1/1
1	Bright yellow	24/24
2	Bright cyan	20/20
3	Bright red	6/6
4	Bright white	26/26
5	Black	0/0
6	Bright blue	2/2
7	Bright magenta	8/8
8	Cyan	10/10
9	Yellow	12/12
10	Pastel blue	14/14
11	Pink	16/16
12	Bright green	18/18
13	Pastel green	22/22
14	Blue/Bright yellow (flashing)	1/24
15	Sky blue/Pink (flashing)	11/16

Appendix C
CPC-464 ROM Routines

The heart of your computer is the Z80A chip, which is rated at 4MHz, but runs at around 3.3 MHz. It communicates with a gate array, which looks after most of the system. The monitor screen is controlled by a 6845 chip (CRTC, Cathode Ray Tube Controller), and the four channel sound (three music, one noise) is an AY-3-8912. The rest of the system is under the control of an 8255.

Memory Map



The Main Firmware Jumpblock

Initialisation

0: KM INITIALISE #BB00 - initialises the key manager; variables, buffers and indirections initialised; previous state lost

1: KM RESET #BB03 - resets the key manager

Characters

2: KM WAIT CHA #BB06 - waits for next character from the keyboard

3: KM READ CHAR #BB09 - tests if a character is available from the keyboard

4: KM CHAR RETURN #BB0C - returns a single character to the keyboard for next call of KM READ CHAR or KM WAIT CHAR

5: KM SET EXPAND #BB0F - sets an expansion string

6: KM GET EXPAND #BB12 - gets character from expansion string

7: KM EXP BUFFER #BB15 - allocates buffer for expansion strings

Keys

8: KM WAIT KEY #BB18 - this tries to get a key from the key buffer, waiting if no key is immediately available

9: KM READ KEY #BB1B - as above, this tries to get a key from the buffer, but does not wait if not available

10: KM TEST KEY #BB1E - tests if key is pressed

11: KM GET STATE #BB21 - tests to see if CAPS LOCK or SHIFT LOCK currently on

12: KM GET JOYSTICK #BB24 - interrogates key state map for current joystick(s) state

Translation Tables

13: KM SET TRANSLATE #BB27 - sets entry in normal key translation table

14: KM GET TRANSLATE #BB2A - gets entry from normal key translation table

15: KM SET SHIFT #BB2D - sets entry in shifted key translation table

16: KM GET SHIFT #BB30 - gets entry from shifted key translation table

17: KM SET CONTROL #BB33 - sets entry in control key translation table

18: KM GET CONTROL #BB36 - gets entry from control key translation table

Repeating

19: KM SET REPEAT #BB39 - sets whether a key may repeat or not

20: KM GET REPEAT #BB3C - asks if a key is allowed to repeat

21: KM SET DELAY #BB3F - sets start-up delay, and repeat speed

22: KM GET DELAT #BB42 - gets start-up delay, and repeat speed

Breaks

23: KM ARM BREAKS #BB45 - arms the break mechanism so next call of KM BREAK EVENT will trigger break event

24: KM DISARM BREAKS #BB48 - prevents break events from being generated

25: KM BREAK EVENT #BB4B - generates break event if the break mechanism is armed

The Text VDU

Initialisation

26: TXT INITIALISE #BB4E - initialises the text VDU

27: TXT RESET #BB51 - reinitialises text VDU indirections and control code table

28: TXT VDU ENABLE #BB54 - allows characters to be placed on screen; cursor 'blob' is also enabled

29: TXT VDU DISABLE #BB57 - prevents characters being placed on screen

Characters

30: TXT OUTPUT #BB5A - outputs character or control code to text VDU

31: TXT WR CHAR #BB5D - writes character to screen

32: TXT RD CHAR #BB60 - reads character from screen

33: TXT SET GRAPHIC #BB63 - toggles graphics VDU write character option

Windows

34: TXT WIN ENABLE #BB66 - sets size of current text window

35: TXT GET WINDOW #BB69 - gets size of current window

36: TXT CLEAR WINDOW #BB6C - clears current window

Cursor

37: TXT SET COLUMN #BB6F - sets cursor horizontal position

38: TXT SET ROW #BB72 - sets cursor vertical position

39: TXT SET CURSOR #BB75 - sets cursor position

40: TXT GET CURSOR #BB78 - query current cursor position

41: TXT CUR ENABLE #BB7B - allows cursor 'blob' to be placed on screen; user

42: TXT CUR DISABLE #BB7E - disallows above

43: TXT CUR ON #BB81 - allows cursor 'blob' to be placed on screen; system

44: TXT CUR OFF #BB84 - disallows above

45: TXT VALIDATE #BB87 - checks if cursor is within current window

46: TXT PLACE CURSOR #BB8A - puts cursor 'blob' on the screen

47: TXT REMOVE CURSOR #BB8D - takes cursor 'blob' off the screen

48: TXT SET PEN #BB90 - sets ink for writing characters

49: TXT GET PEN #BB93 - gets ink for above

50: TXT SET PAPER #BB96 - sets ink for writing text background

51: TXT GET PAPER #BB99 - gets ink for above

52: TXT INVERSE #BB9C - swaps current pen and paper inks

53: TXT SET BACK #BB9F - sets character write mode to opaque or transparent for currently selected stream

54: TXT GET BACK #BBA2 - gets character write mode for current stream

Matrices

55: TXT GET MATRIX #BBA5 - gets address of a character matrix

56: TXT SET MATRIX #BBA8 - sets a character matrix

57: TXT SET M TABLE #BBAB - sets user-defined matrix table address

58: TXT GET M TABLE #BBAE - gets user-defined matrix table address

Control Codes

59: TXT GET CONTROLS #BBB1 - fetches address of control code table

Streams

60: TXT STR SELECT #BBB4 - selects text VDU stream

61: TXT SWAP STREAMS #BBB7 - swaps state of two streams

Entries to Graphics VDU

Initialisation

62: GRA INITIALISE #BBBA - initialises graphics VDU

63: GRA RESET #BBBD - resets graphics VDU

Current Position

64: GRA MOVE ABSOLUTE #BBC0 - move to absolute position

65: GRA MOVE RELATIVE #BBC3 - move to relative position

66: GRA ASK CURSOR #BBC6 - get current graphics position

67: GRA SET ORIGIN #BBC9 - sets location of user origin and moves current position there

68: GRA GET ORIGIN #BBCC - gets origin of user coordinates

Window

69: GRA WIN WIDTH #BBCF - sets right and left edges of graphics window

70: GRA WIN HEIGHT #BBD2 - sets top and bottom of the graphics window

71: GRA GET W WIDTH #BBD5 - gets left and right edges of graphics window

72: GRA GET W HEIGHT #BBD8 - gets top and bottom edges of graphics window

73: GRA CLEAR WINDOW #BBDB - clears graphics window

Inks

74: GRA SET PEN #BBDE - sets graphics plotting ink

75: GRA GET PEN #BBE1 - gets current graphics plotting ink

76: GRA SET PAPER #BBE4 - sets graphics background ink

77: GRA GET PAPER #BBE7 - gets current graphics background ink

Plotting

78: GRA PLOT ABSOLUTE #BBEA - plots point at absolute position

79: GRA PLOT RELATIVE #BBED - plots point relative to current position

Testing

80: GRA TEST ABSOLUTE #BBF0 - tests point at absolute position

81: GRA TEST RELATIVE #BBF3 - tests point relative to current position

Line Drawing

82: GRA LINE ABSOLUTE #BBF6 - draws line to absolute position

83: GRA LINE RELATIVE #BBF9 - draws line relative to present position

Character Drawing

84: GRA WR CHAR #BBFC - puts character on screen at current cursor position

Entries to Screen Pack

Initialisation

85: SCR INITIALISE #BBFF - initialises screen pack

86: SCR RESET #BC02 - resets screen pack

Screen Hardware

87: SCR SET OFFSET #BC05 - sets offset of start of the screen

88: SCR SET BASE #BC08 - sets area of RAM to use for screen memory

89: SCR GET LOCATION #BC0B - fetches current base and offset settings

Mode

90: SCR SET MODE #BC0E - sets screen into new mode

91: SCR GET MODE #BC11 - queries current screen mode

92: SCR CLEAR #BC14 - clears screen to ink zero

93: SCR CHAR LIMITS #BC17 - queries screen size in characters

Screen Addresses

94: SCR CHAR POSITION #BC1A - converts physical coordinates to screen position

95: SCR DOT POSITION #BC1D - converts base coordinates to screen position

96: SCR NEXT BYTE #BC20 - steps screen address right one byte

97: SCR PREV BYTE #BC23 - steps screen address left one byte

98: SCR NEXT LINE #BC26 - steps screen address down one line

99: SCR PREV LINE #BC29 - steps screen address up one line

Inks

100: SCR INK ENCODE #BC2X - encodes an ink to cover all pixels in a byte

101: SCR INK DECODE #BC2f - decodes encoded ink

102: SCR SET INK #BC32 - sets two colours (may be the same) in which to display an ink: same - steady
different - flashing

103: SCR GET INK #BC35 - queries for two colours used to display an ink on screen

104: SCR SET BORDER #BC38 - sets colours for border

105: SCR GET BORDER #BC3B - gets colours for border

106: SCR SET FLASHING #BC3E - sets flash periods

107: SCR GET FLASHING #BC41 - queries current flash periods

Miscellaneous

108: SCR FILL BOX #BC44 - fills character area of screen with ink

109: SCR FLOOD BOX #BC47 - fills byte area of screen

110: SCR CHAR INVERT #BC4A - inverts character position

111: SCR HW ROLL #BC4D - moves whole screen up or down one character

112: SCR SW ROLL #BC50 - moves area of screen up or down one character

113: SCR UNPACK #BC53 - expands character matrix for current screen mode

114: SCR REPACK #BC56 - compresses character matrix to standard form

115: SCR ACCESS #BC59 - sets screen write mode for graphics VDU

116: SCR PIXELS #BC5C - writes pixel to screen, ignoring write mode

Line Drawing

117: SCR HORIZONTAL #BC5F - plots horizontal line

118: SCR VERTICAL #BC62 - plots vertical line

Entries to Cassette Manager

Initialisation

119: CAS INITIALISE #BC65 - initialises cassette manager

120: CAS SET SPEED #BC68 - sets write speed

121: CAS NOISY #BC6B - turns prompt messages on or off

Motor Control

122: CAS START MOTOR #BC6E - starts motor

123: CAS STOP MOTOR #BC71 - stops motor

124: CAS RESTORE MOTOR #BC74 - restores previous motor state

Reading Files

125: CAS IN OPEN #BC77 - opens file for output

126: CAS IN CLOSE #BC7A - closes input file properly

127: CAS IN ABANDON #BC7D - close input file immediately

128: CAS IN CHAR #BC80 - read character from input file

129: CAS IN DIRECT #BC83 - read input file into store in one go

130: CAS RETURN #BC86 - puts last character read by CAS IN CHAR back into read buffer

131: CAS TEST EOF #BC89 - check to see if end of input file reached

Writing Files

132: CAS OUT OPEN #BC8C - opens file for output

133: CAS OUT CLOSE #BC8F - closes output file properly

134: CAS OUT ABANDON #BC92 - closes output file immediately

135: CAS OUT CHAR #BC95 - writes character to output file

136: CAS OUT DIRECT #BC98 - writes output file directly from store

Cataloguing

137: CAS CATALOG #BC9B - generates catalogue from tape

Records

138: CAS WRITE #BC9E - writes record to tape

139: CAS READ #BCA1 - reads record from tape

140: CAS CHECK #BCA4 - compares record on tape with contents of store

Entries to Sound Manager

Initialisation

141: SOUND RESET #BCA7 - resets sound manager

Sound Queues

142: SOUND QUEUE #BCAA - adds a sound to queue

143: SOUND CHECK #BCAD - checks if space in sound queue

144: SOUND ARM EVENT #BCB0 - sets up event to be when sound queue is empty

Sounds

145: SOUND RELEASE #BCB3 - releases held sounds on a number of channels

146: SOUND HOLD #BCB6 - stops all sounds immediately

147: SOUND CONTINUE #BCB9 - starts them again

Envelopes

148: SOUND AMPL ENVELOPE #BCBC - sets up amplitude envelope

149: SOUND TONE ENVELOPE #BCBF - sets up tone envelope

150: SOUND A ADDRESS #BCC2 - gets address of an amplitude envelope

151: SOUND T ADDRESS #BCC5 - gets address of tone envelope

Entries to the Kernel

Initialisation

152: KL CHOKE OFF #BCC8 - resets kernel

153: KL ROM WALK #BCCB - finds and initialises all background ROMs

154: KL INIT BACK #BCCE - initialises specific background ROM

155: KL LOG EXIT #BCD1 - introduces RSX to the firmware

156: KL FIND COMMAND #BCD4 - searches all RSXs, background and foreground ROMS to process a command

Frame Flyback List

157: KL NEW FRAME FLY #BCD7 - initialises and puts block on frame flyback list

158: KL ADD FRAME FLY #BCDA - puts block on frame flyback list

159: KL DEL FRAME FLY #BCDD - takes block off frame flyback list

Fast Tick List

160: KL NEW FAST TICKER #BCE0 - initialises and puts block on fast ticker list

161: KL ADD FAST TICKER #BCE3 - puts block on fast ticker list

162: KL DEL FAST TICKER #BCE6 - takes block off fast ticker list

Tick List

163: KL ADD TICKER #BCE9 - puts block on tick list

164: KL DEL TICKER #BECE - removes block from tick list

Events

165: KL INIT EVENT #BCEF - initialises event block

166: KL EVENT #BCF2 - 'kicks' event block

167: KL SYNC RESET #BCF5 - clears synchronous event queue

168: KL DEL SYNCHRONOUS #BCF8 - takes event out of event queue

169: KL NEXT SYNC #BCFB - gets next queue event

170: KL DO SYNC #BCFE - performs event routine

171: KL DONE SYNC #BD01 - finishes processing an event

172: KL EVENT DISABLE #BD04 - disables normal synchronous events

173: KL EVENT ENABLE #BD07 - enables normal synchronous events

174: KL DISARM EVENT #BD0A - prevents event from occurring

175: KL TIME PLEASE #BD0D - queries for elapsed time

176: KL TIME SET #BD10 - sets elapsed timer

Entries to the Machine Pack

Programs

177: MC BOOT PROGRAM #BD13 - loads and runs a program

178: MC START PROGRAM #BD16 - runs foreground program

Screen

179: MC WAIT FLYBACK #BD19 - waits for frame flyback

180: MC SET MODE #BD1C - sets screen mode

181: MC SCREEN OFFSET #BD1F - sets screen offset

182: MC CLEAR INKS #BD22 - sets all inks to one colour

183: MC SET INKS #BD25 - sets colours for all inks

Printer

184: MC RESET PRINTER #BD28 - resets printer indirection

185: MC PRINT CHAR #BD2B - try to send character to the Centronics port

186: MC BUSY PRINTER #BD2E - tests to see if Centronics port is busy

187: MC SEND PRINTER #BD31 - sends character to the Centronics port

Sound Chip

188: MC SOUND REGISTER #BD34 - sends data to a sound chip register

Entries to Jumper

Initialisation

189: JUMP RESTORE #BD37 - restores the standard jumpblock

Disc BIOS Jumpblocks

There are two jumpblocks, the standard CP/M one, and the extended jumpblock, which starts at #BE80. Details of this jumpblock are as follows:

0: SET MESSAGE	#BE80 -	enables/disables disc error messages
1: SETUP DISC	#BE83 -	resets various disc parameters
2: SELECT FORMAT	#BE86 -	selects AMSTRAD format
3: READ SECTOR	#BE89 -	reads physical sector from disc
4: WRITE SECTOR	#BE8C -	writes physical sector to disc
5: FORMAT TRACK	#BE8F -	formats entire track
6: MOVE TRACK	#BE92 -	unverified move to specified track
7: GET DR STATUS	#BE95 -	return status for specified drive
8: SET RETRY COUNT	#BE98 -	sets number of retries on error
9: ENTER FIRMWARE	#BE9B -	calls firmware routine
10: SET REG SAVE	#BE9E -	sets/clears saving of alternate and IY registers

11: SET SIO #BEA1 - resets and initializes serial interface

12: SET CMND BUFFER #BEA4 - initializes the initial command buffer

13: DO IN STATUS #BEA7 - tests if special character I/O device 0 has character available

14: DO IN #BEAA - fetches character

15: DO OUT STATUS #BEAD - tests if output ready

16: DO OUT #BEB0 - outputs character

17: D1 IN STATUS #BEB3 - tests if special character I/O device 1 has character available

18: D1 IN #BEB6 - fetches character

19: D1 OUT STATUS #BEB9 - tests if output ready

20: D1 OUT #BEBC - outputs character

External Disc Commands

|CPM - shuts down BASIC and AMSDOS and cold boots CP/M

|DISC.IN - redirects tape input firmware entries to disc counterparts

|DISC.OUT - redirects tape output firmware entries to disc counterparts

|DISC - command redirects tape input and output firmware entries to disc counterparts

|TAPE.IN - restores tape input firmware entries to pre-AMSDOS initialization state

|TAPE.OUT - restores tape output firmware entries

|TAPE - restores tape firmware entries

|A - sets default drive to drive A

|B - sets default drive to drive B

|DRIVE - sets current default drive

|USER - sets default user number

|DIR - displays disc directory

|ERA - erases files

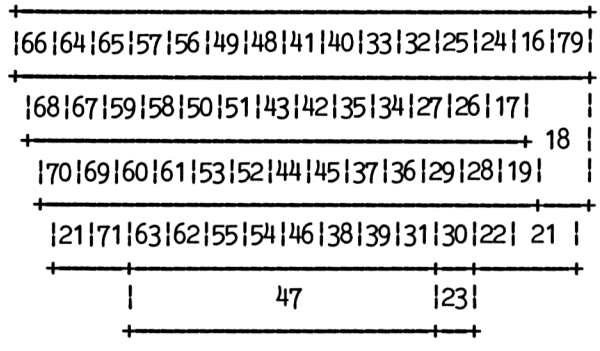
|REN - renames a file

Appendix D
Coping with Error Messages

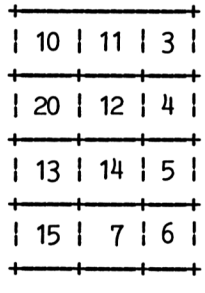
1 — Unexpected NEXT. Check FOR/NEXT loops
2 — Syntax Error. Illegal line in BASIC
3 — Unexpected RETURN, Check GOSUB calls
4 — DATA exhausted. Do you need RESTORE. Are there insufficient items of DATA?
5 — Improper Argument. Invalid parameters
6 — Overflow. Number too big
7 — Memory full. Program too big. Too many nested constructions
8 — Line does not exist. Call to a line which is not there
9 — Subscript out of range. Check size of dimensioned arrays
10 — Array already dimensioned. The same array is DIMmed more than once in the program
11 — Division by zero. Check intermediate results of calculations
12 — Invalid direct command. Must be part of a program
13 — Type mismatch. You've given Arnold a string when it expected a number, of vice versa
14 — String space full
15 — String too long. Cut down to 255 characters
16 — String expression too complex. Cut it down to smaller strings
17 — Cannot CONTinue
18 — Unknown user function. You haven't DEF FNed when you should have
19 — RESUME missing. ON ERROR GOTO routine incomplete
20 — Unexpected RESUME. You can only use it in an ON ERROR GOTO
21 — Direct command found. Line without line number found on cassette
22 — Operand missing. Check numeric expressions in program
23 — Line too long. Cut down longest line(s)
24 — EOF met. File reading error. Check file parameters
25 — File type error. As for 24.
26 — NEXT missing. Count your FORs
27 — File already open
28 — Unknown command
29 — WEND missing. Count your WHILEs and WENDs
30 — Unexpected WEND. Check positions of WHILEs

E — Key Numbering

Main Keyboard

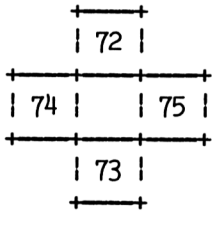


Function/Numeric Keypad

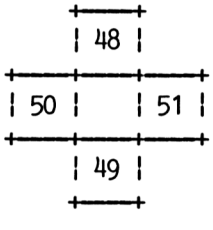


Joysticks:

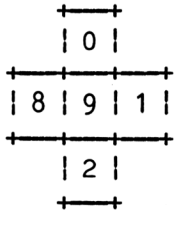
Joystick 0



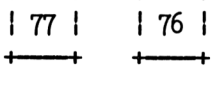
Joystick 1

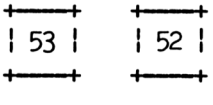


Cursor Keys

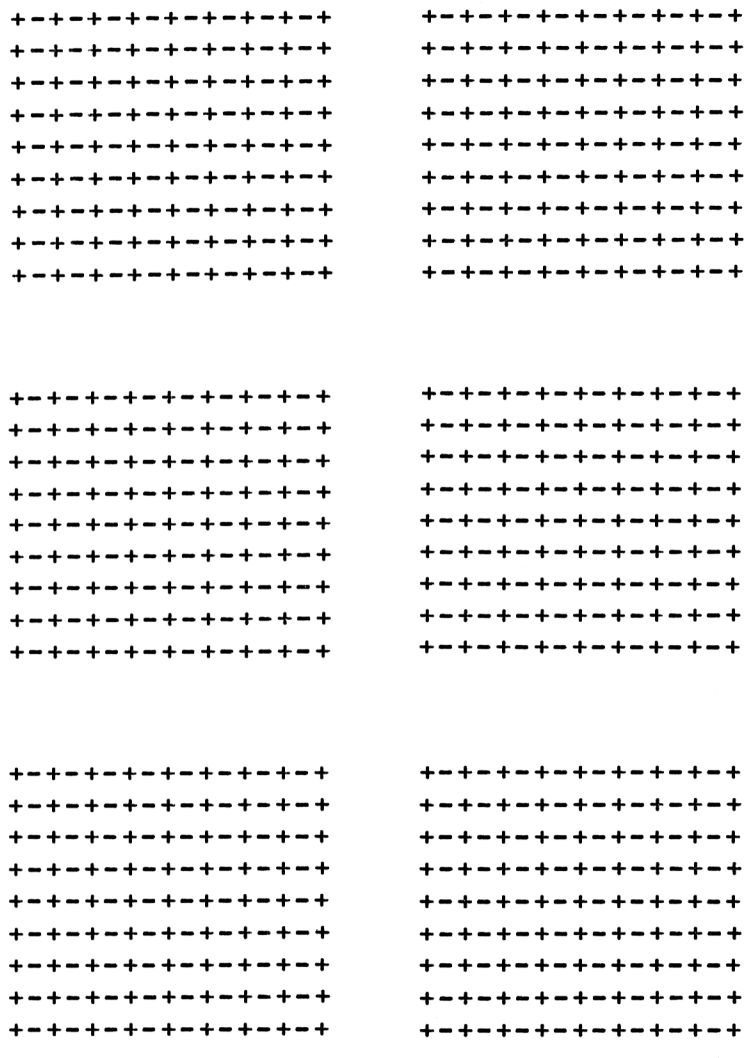


Fire:





F — User-Defined Graphics Forms



User-defined graphics forms may be photocopied without restriction.

Here it is. The ultimate programming resource for your Amstrad computer. A bumper collection of ideas, tricks, techniques and programs for you and your machine.

The major sections include:

EXPLORING ARTIFICIAL INTELLIGENCE — detailed history; eight programs to demonstrate AI including BLOCKWORLD and X-SPURT.

CREATING ADVENTURE PROGRAMS — complete "how to do it" instructions, plus major adventure programs listed in full.

THE AMSTRAD GRAPHICS COMPANION — huge selection of data for user-defined graphics, plus generous collection of dramatic demonstrations.

THE AMSTRAD DISC COMPANION — make the best use of your disc drive and CP/M

LOGO-K — a complete LOGO graphics implementation plus tutorial; uses standard primitives, so programs run with other Logo programs, including Dr. Logo.

EXPLORATION, ENTERTAINMENT and UTILITIES — a host of worthwhile games and utility programs, ready to run

THE AMAZING AMSTRAD OMNIBUS is designed to give you and your Amstrad months and months of ideas and entertainment.

ISBN 0-907563-82-1

£7.95

THE AMAZING AMSTRAD OMNIBUS

Martin Fairbanks

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/

AMSTRAD
DJR