The Amazing

DICK SMITH ELECTRONICS

VZ300

Personal Colour
Computer

Omnibus

Tim Hartnell

The Amazing

DICK SMITH ELECTRONICS PTY LTD

VZ300

Omnibus

Tim Hartnell

The Amazing DICK SMITH ELECTRONICS VZ300 Omnibus

Tim Hartnell

First published in Australia by:
Dick Smith Management Pty. Ltd.,
PO Box 321,
North Ryde, NSW, 2113

Copyright (c) Tim Hartnell, 1986

First printing 1986

Dick Smith Catalogue Number: X-7327

This book was prepared for Dick Smith Management Pty. Ltd. by Interface Publications (Australia) Pty. Ltd., Chelsea House, 34 Camp Street, Chelsea, Vic., 3196. Any queries regarding the contents of the book should be sent to this address.

Special thanks to the staff of Dick Smith Electronics in Frankston, Victoria, for their assistance in preparing this book.

Typeset by Wellset Pty. Ltd., Chelsea Heights, Victoria.

National Library of Australia Card No.
& ISBN 0 949772 38 0

The author, Melbourne-born Tim Hartnell, first met Dick Smith in London in the late seventies while Dick was on a trip to buy the helicopter which he used for his record-breaking world solo flight.

The two kept in touch, and after Tim's return from the UK at the end of 1982, Dick called him one day and asked him to come up to Sydney to see a new product which Dick Smith Electronics had developed. It was the VZ200, the forerunner of the VZ300. Tim was impressed with the machine and its potential, and rang the editor of *Australian Personal Computer* about it. The editor decided the launch of the VZ200 was sufficiently important to change, at the last minute, the proposed cover of the next issue of the magazine, and replace it with a photograph of the VZ200. Tim's review of the computer was the lead story in that issue.

Tim and a couple of authors who had worked with him previously on books published by Tim's company, Interface Publications, then wrote three books on the VZ200, which were distributed exclusively through Dick Smith outlets. The feedback on those books was extremely positive, so it was decided that the VZ300 also deserved dedicated books, such as the one you are reading now.

Tim Hartnell originally left Australia in the middle of 1977 for the UK for what was planned to be a six month's working holiday. He finally stayed for nearly six years, and while in London founded his company, Interface Publications, which now has offices in London, Melbourne and New York. Tim has written fifty or so books (he says he doesn't know the exact number), predominately on computers and related subjects, and they have been translated into eleven languages.

Although he now makes his home in Melbourne, he travels back to the USA and the UK two or three times each year to keep in touch with developments in the computer field. "It gets harder to leave each time," he said recently. "Australia is really the only country I ever want to live in. And with developments like the VZ300, it makes sense professionally, as well as personally."

Contents

Section Five — Sorting and Searching

Section Six — Disk Drives and other Peripherals

Foreword

Your VZ300 is a remarkably flexible computer, and in this book we're going to explore some of that flexibility together.

We start by looking at the sound and graphics potential of your computer. Programs with names like 3-D PRINTER PLOTTER and PATTERN-MASTER only hint at what you can achieve. The sound demonstration programs include ALIEN ATTACK and OBJECT FALLING DOWN STAIRS, to get you ready to annoy everyone in the neighbourhood.

From there we move to the largest section of the book, which shows how you can explore the amazing world of Artificial Intelligence on your VZ300. You'll discover a Noughts and Crosses program which learns as it plays to become a stronger opponent, a reasoning program called SYLLOGY, and a fun-to-beat board game with the name of SNICKERS.

From there, you'll interact with the BLOCKWORLD, as your VZ300 moves coloured blocks around the screen in response to your commands. Translate English into rather strange French with TRANSLATE, and produce reams of poetry using our HANSHAN program.

Once your mind (and VZ300) have recovered from all that intelligence, you can set your computer to work with MINICALC and MORTGAGE.

The fascinating computer language FORTH comes next in this book. We include a complete version of the language, which you can type in so you can run and learn FORTH on your VZ300, without spending a further cent buying an additional language for your computer. FORTH even allows you to define your own words.

A number of searching and sorting techniques follow, all with easy-to-enter programs which allow you to test the speed of the various routines for yourself.

The sixth section of this book examines the wide range of peripherals which you can get for your VZ300, such as joysticks, a disk drive, and a printer.

With all this, I don't think your VZ300 is going to be able to complain that it hasn't got anything to do for many, many months to come.

Good Programming,
Tim Hartnell,
Melbourne, 1986

Section One Graphics and Sound Companion

You'll be amazed at the effects which your VZ300 can produce in the sound and graphics departments. We examine some of the possibilities in this section of the book.

Graphics

We start off with *Pattern-Master 1* which generates an infinite number of randomly-designed paterns. Just press any key when you want to start a new design.

Here's the listing:

```
10 REM PATTERN-MASTER 1
20 REM PRESS ANY KEY TO START
30 REM    A NEW DESIGN
110 MODE(1)
120 COLOR (1+RND(3)),(RND(2)-1)
130 LOZ=RND(3)/.02
140 GF=0
150 SET((60+37*SIN(GF)),(30+27*COS(GF*LOZ)))
180 GF=GF+.01
185 IF INKEY$<>"" THEN RUN
190 GOTO 150
```

From that we move to an even more impressive program, *Pattern-Master 2*. This is a very fine example of how the SET command can be used. If you don't like one design, just press a key and a new one will begin instantly.

Once the screen has filled with a design, a small tune will play, then a new one will begin.

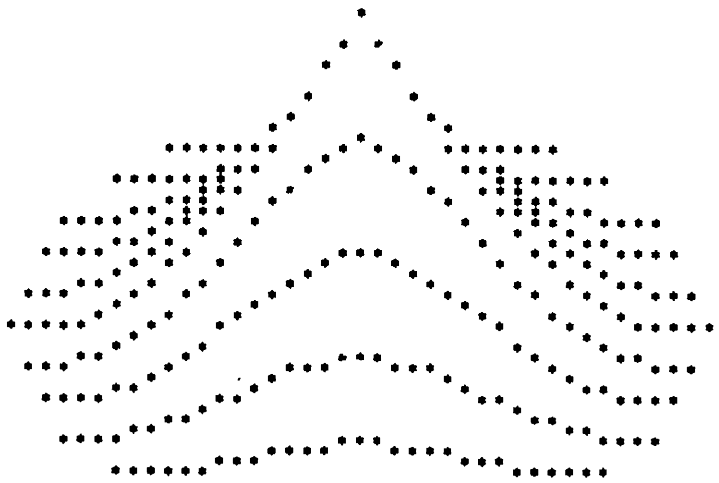
Here's the listing for *Pattern-Master 2:*

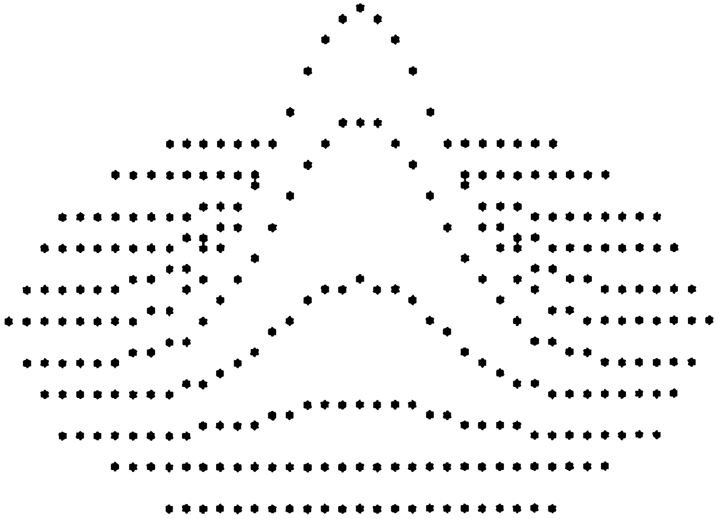
```
10 REM PATTERN MASTER 2
20 MODE(1)
30 Q=RND(2)-1
40 FOR X=1 TO 63 STEP RND(0)
45 COLOR (1+RND(3)),Q
50 Y=RND(31)
60 SET(X,Y):SET(128-X,Y)
70 SET(128-X,64-Y)
80 SET (X,64-Y)
90 IF INKEY$<>"" THEN RUN
100 NEXT X
110 FOR J=1 TO 30 STEP (2+RND(5))
120 SOUND J,RND(3)
130 SOUND J/2+1,RND(3)
140 NEXT J
150 RUN
```

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 210, 225, 245 and 250. Once you've seen these in action, you can try substituting some of your own.

The printouts show what a few of the sample ones look like.





Here's the listing, so you can see it in action on your own computer:

```
10 REM 3D PRINTER PLOTTER
20 CLS
30 FOR DE=1 TO 4:GOSUB 180
35 SOUND 3*DE,3;DE,3
37 LPRINT:LPRINT:LPRINT:LPRINT
40 FOR CP=-BI TO BI STEP BY
50 AL=0
60 SU=VA*INT(SQR(900-CP*CP)/VA)
70 FOR TI=SU TO -SU STEP -VA
75 GOSUB 210
80 ST=INT(VA*VA+X-CH*TI)
90 IF ST<=AL THEN 120
100 AL=ST
110 LPRINT TAB(ST);"*";
115 PRINT TAB(ST/2);"*";
120 LPRINT " ";:NEXT TI
130 PRINT:LPRINT
140 NEXT CP
150 NEXT DE
160 END
170 REM ----------------------
180 PRINT:PRINT
```

```
190 VA=5:BY=1.5:BI=6*VA:CH=.7
200 RETURN
210 Q=SQR(CP*CP+TI*TI)
220 IF DE<>1 THEN 230
225 X=BI*EXP(-Q*Q/100):RETURN
230 IF DE<>2 THEN 240
235 X=BI*EXP(-COS(Q/16))-BI:RETURN
240 IF DE<>3 THEN 250
245 X=BI-BI*SIN(Q/18):RETURN
250 X=SQR(BI*BI+BYTE/100-Q*Q):RETURN
```

Generating Sound

The SOUND command can be used to add life to your programs. A small amount of sound can do a lot to enhance a program.

The command must always be followed by two numbers (or by variables representing numbers). The first number is the pitch, or frequency, of the note to be played, and the second determines for how long the note will sound.

The pitch is a number between 0 and 31, and the duration is a number between 1 and 9. The pitch value of 0 produces no sound. It can be used as a 'rest', in music terms.

Frequency

Here are the frequencies for the VZ300. The first number is the number you use (as the first one after the SOUND command), and the second is its actual musical value:

0 —	rest	16 —	C4	12 —	G#3	28 —	C5
1 —	A2	17 —	C#4	13 —	A3	29 —	C#5
2 —	A#2	18 —	D4	14 —	A#3	30 —	D5
3 —	B2	19 —	D#4	15 —	B3	31 —	D#5
4 —	C3	20 —	E4				
5 —	C#3	21 —	F4				
6 —	D3	22 —	F#4				
7 —	D#3	23 —	G4				
8 —	E3	24 —	G#4				
9 —	F3	25 —	A4				
10 —	F#3	26 —	A#4				
11 —	G3	27 —	B4				

Duration

Now, here are the relevant note lengths. The first number is the one you have after the comma in the SOUND command, and the second number is its value, taking a quaver to have a value of 1:

Number	Duration
1	1/8
2	1/4
3	3/8
4	1/2
5	3/4
6	1
7	1 1/4
8	2
9	3

A simple sound, repeated over and over again, can be very effective, as this *Motor Boat* demonstrates:

```
10 REM MOTOR BOAT
20 FOR J=1 TO 2:SOUND J,1:NEXT
30 GOTO 20
```

Using the SOUND command within a loop can also be a good way to produce interesting sound, as you'll hear if you run this brief routine in which an *Object Falls Down Stairs:*

```
5 REM OBJECT FALLS DOWN STAIRS
10 FOR J=31 TO 1 STEP -1
20 SOUND J,INT(4-J/13)
30 NEXT
40 SOUND 1,7
```

You can combine more than one SOUND statement at a time within a loop to produce very effective results, as *Alien Attack* demonstrates convincingly:

```
10 REM ALIEN ATTACK
20 FOR Z=1 TO 4:FOR H=1 TO 5
30 SOUND 32-Z,1
```

```
40 SOUND H+Z/2,1
50 SOUND 30-H,1
60 SOUND 31-H-Z,1
70 NEXT:NEXT
```

Or how about this one, which changes your VZ300 into an alarm system:

```
10 REM ALERT!
20 FOR K=1 TO 4:FOR J=1 TO RND(9)
30 SOUND J/2+1,1
40 SOUND J,1:SOUND 30-(J+K),1
50 NEXT:NEXT
60 GOTO 20
70 REM 'CTRL / BREAK' TO STOP
```

The VZ Synthesizer

Finally in this section, we have a program which will allow you to use your VZ300 as a kind of electronic synthesizer, complete with a graphic on-screen representation of a two-octave keyboard.

Once you see it on screen, you'll know instantly which key to press to get which result. You'll note as well that as you press each key, the key changes colour while the note is sounding.

Take care when entering this program. Note that the X's in lines 50, 70 and 90 should be repaced with *inverse spaces*, with the exception of the *third* X in line 90, which stays as an X.

```
10 REM VZ SYNTHESIZER
20 CLS:DU=1:DIM A(26),B(26),C(26)
30 G$=CHR$(128):H$=CHR$(143):I$=H$+H$
35 J$=CHR$(224)+CHR$(224)
40 A$=G$+"  "+G$+"     "+G$+"  "+G$+"  "+G$
50 B$="XX XX XX XX XX XX XX XX"
60 C$="  2  3     5  6  7"
70 D$="QX WX EX RX TX YX UX IX"
80 E$="  S  D     G  H  J     "
90 F$="ZX XX CX VX BX NX MX ,X"
```

```
100 PRINT TAB(4);C$:PRINT TAB(6);A$:PRINT TAB(6);A$
110 PRINT TAB(4);B$:PRINT TAB(4);B$:PRINT TAB(4);D$
115 PRINT
120 PRINT TAB(4);E$:PRINT TAB(6);A$:PRINT TAB(6);A$
130 PRINT TAB(4);B$:PRINT TAB(4);B$:PRINT TAB(4);F$
140 PRINT:PRINT "    PRESS SPACE BAR TO EXIT";
150 FOR J=1 TO 26:READ Z$:A(J)=ASC(Z$):NEXT
160 DATA 2,3,5,6,7,Q,W,E,R,T,Y,U
165 DATA I,S,D,G,H,J,Z,X,C,V,B,N,M,",",
170 FOR J=1 TO 26:READ B(J):NEXT
180 DATA 38,41,47,50,53,132,135,138,141,144
185 DATA 147,150,153,262,265,271
190 DATA 274,277,356,359,362,365,368
195 DATA 371,374,377
200 FOR J=1 TO 26:READ C(J):NEXT
210 DATA 17,19,22,24,26,16,18,20,21,23,25,27,28
220 DATA 5,7,10,12,14,4,6,8,9,11,13,15,16
230 K$=INKEY$:IF K$="" THEN 230
240 IF K$=" " THEN PRINT @448,"":END
250 FOR I=1 TO 26
260 IF K$=CHR$(A(I)) THEN FR=C(I):GOTO 270
265 NEXT I:GOTO 230
270 IF I<6 OR I>13 AND I<19 THEN M$=G$:L$=H$:C=2
280 IF NOT (I<6 OR I>13 AND I<19) THEN L$=I$:M$=J$:C=3
290 COLOR C:PRINT @B(I),L$
300 SOUND FR,DU
310 PRINT @B(I),M$
320 N$=INKEY$:N$=INKEY$
340 K$="":GOTO 230
```

Section Two 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 cannot admire, or even recognise, a particularly effective poem produced by HANSHAN, and it probably isn't proud of its skills in BLOCKWORLD.

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.

AI is in its infancy, and to expect to elicit real awareness and perception from a short BASIC program on a VZ300, 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 (or 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 A1 programs do not spring into the VZ300 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 deliveratedly 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 AI 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.

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 - VZ300 VERSION
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$=""
120 GOSUB 1070:REM PRINT BOARD
130 REM *** MAIN CYCLE ***
140 GOSUB 540:REM MACHINE MOVE
150 GOSUB 1070:REM PRINT BOARD
160 GOSUB 870:REM WIN CHECK
170 IF R$<>"" THEN 240
180 GOSUB 980:REM ACCEPT HUMAN MOVE
```

```
190 GOSUB 1070:REM PRINT BOARD
200 GOSUB 870:REM WIN CHECK
210 IF R$="" THEN 140
220 REM *** END MAIN CYCLE ***
230 REM ***********
240 REM END OF GAME
250 GOSUB 1070:REM PRINT BOARD
260 PRINT:PRINT
270 IF R$="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 4
30
300 REM UPDATE KNOWLEDGE BASE
310 FOR B=1 TO 5
320 FOR J=2 TO 9
330 IF M(J)=D(B) THEN GOSUB 370
340 NEXT J
350 NEXT B
360 GOTO 430
370 REM ** RE-ORDER ELEMENTS OF M ARRAY **
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 RETURN TO CONTINUE"
510 INPUT A$
520 GOTO 30
530 REM ************
540 REM MACHINE MOVE
550 P=ASC("O")
560 X=0
570 J=1
575 IF A(W(J))<>A(W(J+1)) THEN 585
580 IF A(W(J+2))=32 AND A(W(J))=P THEN X=W(J+2):GOTO
 750
```

```
585 IF A(W(J))<>A(W(J+2)) THEN 595
590 IF A(W(J+1))=32 AND A(W(J))=P THEN X=W(J+1):GOTO
 750
595 IF A(W(J+1))<>A(W(J+2)) THEN 610
600 IF A(W(J))=32 AND A(W(J+1))=P THEN X=W(J):GOTO 7
50
610 IF J<21 THEN J=J+3:GOTO 580
620 IF P=ASC("O") THEN P=ASC("X"):GOTO 570
630 REM ** IF NO WIN/BLOCK MOVE FOUND **
640 REM ** THEN THIS NEXT SECTION USED **
650 J=1
660 IF A(M(J))=32 THEN X=M(J):GOTO 750
670 IF J<10 THEN J=J+1:GOTO 660
680 H=0
690 H=H+1
700 X=RND(9):IF A(X)=32 THEN 750
710 IF H<100 THEN 690
720 R$="D":REM IT IS A DRAW
730 RETURN
740 REM *********
750 REM MAKE MOVE
760 A(X)=ASC("O")
770 COUNT=COUNT+1
780 D(COUNT)=X
790 FLAG=0
800 FOR J=1 TO 9
810 IF A(J)=32 THEN FLAG=1
820 NEXT J
830 IF FLAG=0 AND R$="" THEN R$="D"
840 REM IF ALL FULL, R$ NOT ASSIGNED, 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$="W":REM VZ WINS
950 IF A(W(J))=ASC("X") THEN R$="L":REM VZ LOSES
960 RETURN
```

```
970 REM **********
980 REM HUMAN MOVE
990 PRINT:PRINT
1000 PRINT "ENTER YOUR MOVE"
1010 INPUT MOVE
1020 IF MOVE<1 OR MOVE>9 THEN 1010
1030 IF A(MOVE)<>32 THEN 1010
1040 A(MOVE)=ASC("X")
1050 RETURN
1060 REM ***********
1070 REM PRINT BOARD
1080 CLS
1090 PRINT:PRINT:PRINT
1100 PRINT "1 : 2 : 3   ";CHR$(A(1));" : ";CHR$(A(2)
);
1110 PRINT  " : ";CHR$(A(3)):PRINT "---------   ----
-----"
1120 PRINT "4 : 5 : 6   ";CHR$(A(4));" : ";CHR$(A(5)
);
1130 PRINT " : ";CHR$(A(6)):PRINT "---------   -----
----"
1140 PRINT "7 : 8 : 9   ";CHR$(A(7));" : ";CHR$(A(8)
);
1150 PRINT " : ";CHR$(A(9)):PRINT
1160 RETURN
1170 REM **************
1180 REM INITIALISATION
1190 CLS
1200 DIM A(9):REM BOARD
1210 DIM M(10):REM TO HOLD 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)
1270 NEXT J
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)
1340 NEXT J
1350 DATA 2,6,8,4,7,3,1,9,5,2
1360 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=RND(9)
4540 IF A(MVE)=32 THEN A(MVE)=ASC("X"):RETURN
4550 IF H<100 THEN 4520
4560 R$="D"
4570 RETURN
```

To trigger this unintelligent, tireless opponent on your VZ300, simply replace line 180 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 RETURN 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 NO MORE SUBJECT 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:INPUT A$
40 IF A$="?" THEN 910
50 IF A$="" THEN END
60 FLAG=0
70 REM NOTE SPACE BEFORE CLOSE QUOTE IN NEXT LINES
80 IF LEFT$(A$,3)="IS " THEN 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$,N-1):GOTO 1
80
160 IF N<X THEN 140
170 PRINT "I DON'T UNDERSTAND":GOTO 30
180 K=4
190 IF MID$(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$,3):REM REMOV
ES ARTICLE
220 IF LEFT$(C$,3)="AN " THEN C$=MID$(C$,4)
230 IF LEFT$(C$,4)="THE " THEN C$=MID$(C$,5)
240 REM ** STORE INFORMATION **
250 REM CHECK TO SEE IF CAN FIND SUBJECT BEFORE BLAN
K
260 N=0
270 N=N+1
280 IF Z$(1,N)=B$ THEN 320:REM SUBJECT HEADING EXIST
S
290 IF Z$(1,N)="" THEN Z$(1,N)=B$:GOTO 320
300 IF N<25 THEN 270
310 PRINT "NO MORE SUBJECT ROOM"
320 REM REACHES HERE WITH SUBJECT STORED AS HEADING
330 REM ** NOW PUT OBJECT UNDER THIS **
```

```
340 K=0
350 K=K+1
360 IF Z$(K,N)=C$ THEN 400:REM ALREADY STORED
370 IF Z$(K,N)="" THEN Z$(K,N)=C$:GOTO 400
380 IF K<25 THEN 350
390 PRINT "NO MORE OBJECT SPACE"
400 IF FLAG=1 THEN PRINT TAB(6);"> OK":GOTO 30
410 REM ** NOW SWAP OBJECT AND SUBJECT AND SAVE AGAI
N **
420 FLAG=1
430 M$=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"
520 IF LEFT$(A$,3)="AN " THEN A$=MID$(A$,4):REM STRI
P "AN"
530 REM ** GET FIRST WORD - F$ **
540 X=LEN(A$)
550 N=0
560 N=N+1
570 IF MID$(A$,N,1)=" " THEN F$=LEFT$(A$,N-1):GOTO 6
00
580 IF N<X THEN 560
590 PRINT "> DO NOT UNDERSTAND":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)
630 PRINT "(LOOKING FOR ";F$;")"
640 X=0
650 X=X+1
655 IF Z$(1,X)<>F$ THEN 670
660 PRINT "( FOUND AT 1";X;")":GOTO 700
670 IF X<25 THEN 650
680 PRINT "CANNOT FIND SUBJECT":PRINT TAB(2);F$
690 GOTO 30
700 Y=1
710 Y=Y+1
```

```
720 IF Z$(Y,X)=S$ THEN 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 PRINT TAB(6);"> NO":GOTO 30
830 Q=1
840 Q=Q+1
850 IF Z$(Q,M)=S$ THEN PRINT TAB(6);"> YES":GOTO 30
860 IF Q<25 THEN 840
870 IF M<25 THEN 780
880 GOTO 820
900 REM ***********************************
910 REM CHECK CONTENTS OF PARTICULAR FILE
920 INPUT "SUBJECT TO CHECK";H$
930 T=0
940 T=T+1
950 IF Z$(1,T)=H$ THEN 990
960 IF T<25 THEN 940
970 PRINT "I HAVE NO DATA STORED ON ";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 CLS
1080 DIM Z$(25,25)
1090 RETURN
```

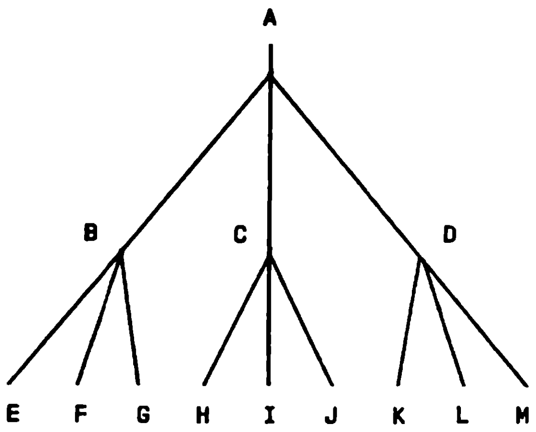
Part Four — Search Trees and Snickers

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 AI 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.

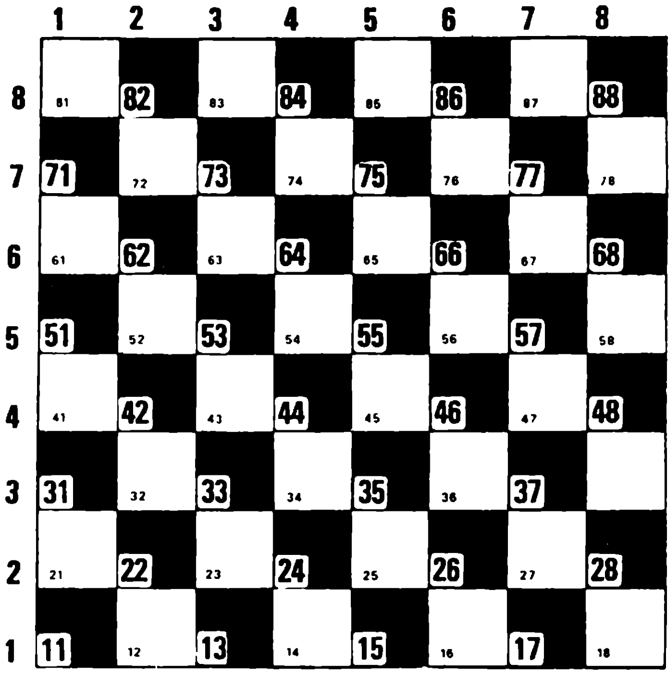
Vanishing Acts

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 VZ300 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):

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

The numbers printed here by the computer refer to those within a master array which holds the board inside your VZ300. 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 VZ300 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 VZ300 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.

And of course, and this where '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 37, or on a game board you have numbered in the same way.

The VZ300 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.

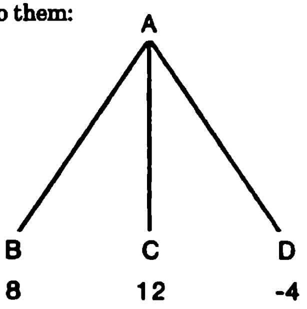
Mini-Maxing

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.

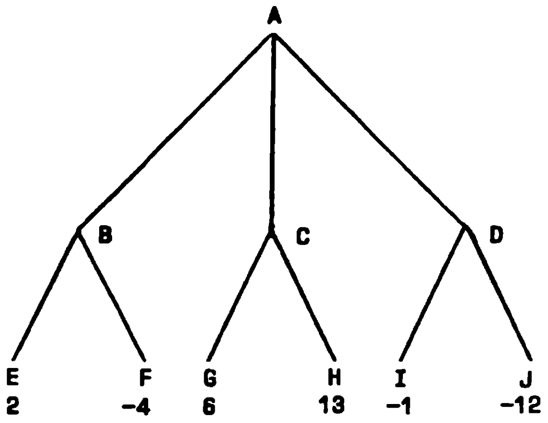
To use this, however, the computer should be able to assign numerical values to the positions it discovers.

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 has failed to find a move, the VZ300 picks locations on the 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 VZ300 chooses at random from these.

If this search fails to find a move, our VZ300 looks to see if it has any 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, except 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.

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.

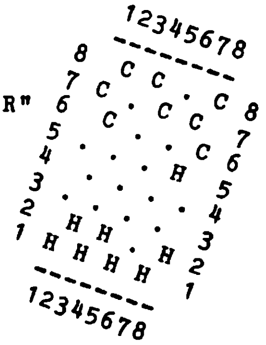
Moving a piece onto the back row carries the same reward as capturing a piece, so the next item in the hierarchy is to make a move onto the back row if that is possible. The routine from 1140 looks after this. The sequence of squares checked in this section means those in the middle squares will move 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 - VZ300 VERSION
20 GOSUB 2070:REM INITIALISE
30 GOSUB 1760:REM PRINT BOARD
40 REM ** MAIN CYCLE STARTS **
```

```
50 GOSUB 190:REM COMPUTER 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 IF HS<5 THEN 50
110 REM ***********
120 REM END OF GAME
130 PRINT:PRINT "THE GAME IS OVER"
140 PRINT
150 IF HS>CS THEN PRINT "YOU HAVE WON"
160 IF CS>HS THEN PRINT "I'M THE WINNER"
170 END
180 REM ***************
190 REM COMPUTER MOVES
200 REM ***************
210 REM SEARCH FOR CAPTURES
220 GSAFE=0
230 CSAFE=0
240 CCAPTURE=0
250 FOR J=1 TO 3
260 G(J)=0:REM EMPTY GOOD, SAFE CAPTURE STORE
270 S(J)=0:REM EMPTY SAFE CAPTURE STORE
280 T(J)=0:REM EMPTY OTHER CAPTURE 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 NO COMPUTER 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:REM NO CAPTUR
ES FOUND
420 REM ** NOW CHOOSE CAPTURE TO MAKE **
430 PRINT:PRINT TAB(8);">> CAPTURE FOUND"
```

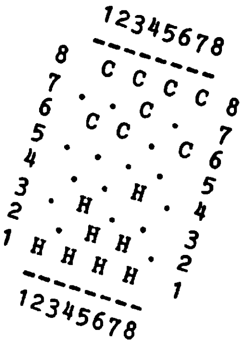


```
440 FOR T=1 TO 1000:NEXT T
450 IF GSAFE<>0 THEN 500
460 IF CSAFE<>0 THEN 670
470 REM ** CHOOSE FROM GENERAL CAPTURES **
480 MOVE=T(RND(CCAPTURE))
490 GOTO 540
500 REM ** CHOOSE FROM GOOD SAFE **
510 REM ** SELECT FROM STORED MOVES **
520 MOVE=G(RND(GSAFE))
530 REM ** MAKE MOVE **
540 START=INT(MOVE/100)
550 ED=MOVE-100*START
560 A(START)=B
570 A(START-ED)=B
580 A(START-2*ED)=C
590 CS=CS+1
600 REM ** CHECK IF LANDING ON BACK ROW **
610 IF START-2*ED>18 THEN RETURN
620 A(START-2*ED)=B
630 CS=CS+1
640 PRINT "I CAPTURED AND LANDED ON"
645 PRINT START-2*ED;"ON BACK ROW"
650 FOR T=1 TO 2000:NEXT T
660 RETURN
670 REM ** SAFE CAPTURE **
680 MOVE=S(RND(CSAFE))
690 GOTO 540
700 REM ** CHECK PROPOSED CAPTURE FOR SAFETY **
710 REM CHECK SQUARE BELOW
720 PRINT J+K;"TO";Y;"CAPTURING ON";X
730 FOR T=1 TO 900:NEXT T
740 IF A(Z)=H THEN 920:REM STORE AS A NON-SAFE CAPTU
RE
750 REM CHECK IN OTHER DIRECTION
760 IF A(Y+M)=H AND A(Y-M)=B THEN 920
770 REM CHECK IF PIECE EXPOSED
780 IF A(J+K+M)=C AND A(J+K+2*M)=H THEN 920
790 REM ** IF REACHED THIS POINT THEN CAPTURE IS 'SA
FE' **
800 REM ** STORE THIS MOVE **
810 CSAFE=CSAFE+1
820 S(CSAFE)=100*(J+K)+20+M:REM TO RECREATE MOVE
```

```
830 REM A 'GOOD SAFE' CAPTURE
840 CHECK=GSAFE
850 IF Y+2*M<1 THEN RETURN
855 IF A(Y+M)<>H THEN 870
860 IF A(Y-(20+M))<>B AND A(Y+2*M)=B THEN GSAFE=GSAF
E+1
870 IF CHECK=GSAFE THEN RETURN:REM NOT 'GOOD SAFE'
880 REM ** STORE GOOD SAFE MOVE **
890 PRINT "I AM CONSIDERING";J+K;"TO";M+20+J+K
900 G(GSAFE)=100*(J+K)+20+M
910 RETURN
920 REM ** STORE NON-SAFE CAPTURE **
930 CCAPTURE=CCAPTURE+1
940 PRINT "I AM CONSIDERING";J+K;"TO";M+20+J+K
950 T(CCAPTURE)=100*(J+K)+20+M
960 RETURN
970 REM ****************************
980 REM ** MOVE TO PROTECT PIECE UNDER THREAT **
990 MOVE=0
1000 J=80
1010 K=1
1020 Q=J+K
1030 IF A(Q)<>C THEN 1110
1035 IF A(Q+9)<>B THEN 1050
1040 IF A(Q-9)=H AND A(Q+18)=C THEN MOVE=100*(Q+18)+
Q+9
1050 REM RANDOM REJECTION OF MOVE
1055 IF MOVE=0 OR A(Q-9)<>H THEN 1065
1060 IF A(Q+20)=B AND RND(2)=1 THEN 1510
1065 IF A(Q+9)<>B OR A(Q-9)<>H THEN 1075
1070 IF A(Q+20)=C THEN MOVE=100*(Q+20)+Q+9:GOTO 1510

1075 IF A(Q+11)<>B OR A(Q-11)<>H THEN 1085
1080 IF A(Q+22)=C THEN MOVE=100*(Q+22)+Q+11
1085 IF KOVE=0 OR A(Q+2)<>H THEN 1095
1090 IF A(Q+22)=B AND RND(2)=1 THEN 1510
1095 IF A(Q+11)<>B OR A(Q-11)<>H THEN 1110
1100 IF A(Q+20)=C THEN MOVE=100*(Q+20)+Q+11:GOTO 151
0
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
1150 MOVE=0
```

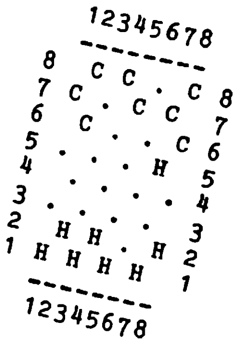
```
1160 REM UNDESIRABLE MOVES FIRST
1170 IF A(22)=C AND A(11)=B THEN MOVE=22
1180 IF A(28)=C AND A(17)=B THEN MOVE=28
1190 IF A(22)=C AND A(13)=B THEN MOVE=22
1200 IF A(26)=C AND A(17)=B THEN MOVE=26
1210 IF A(26)=C AND A(15)=B THEN MOVE=26
1220 IF A(24)=C AND A(15)=B THEN MOVE=24
1230 IF A(24)=C AND A(13)=B THEN MOVE=24
1240 IF MOVE=0 THEN 1310
1250 PRINT:PRINT "TO BACK ROW FROM";MOVE
1260 FOR T=1 TO 2000:NEXT T
1270 A(MOVE)=B
1280 CS=CS+1
1290 RETURN
1300 REM ******************************
1310 REM ** SAFE, NON-CAPTURE MOVES **
1320 CMOVE=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 AND 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 CMOVE=0 THEN 1630
1490 REM ** MAKE MOVE **
1500 MOVE=T(INT(RND(1)*CMOVE)+1)
1510 START=INT(MOVE/100)
1520 ED=MOVE-100*START
1530 A(START)=B
1540 A(ED)=C
1550 RETURN
1560 REM ** STORE MOVES **
1570 CMOVE=CMOVE+1
1580 PRINT "CONSIDERING";J+K;"TO";X
```

```
1590 FOR T=1 TO 800:NEXT T
1600 T(CMOVE)=100*(J+K)+X
1610 RETURN
1620 REM ************************
1630 REM RANDOM NON-CAPTURE MOVE
1640 PRINT "LOOKING FOR RANDOM, LEGAL MOVE"
1650 L=0
1660 L=L+1
1670 J=10*INT(RND(1)*8+1)
1680 K=INT(RND(1)*8+1)
1690 IF A(J+K)=C THEN 1720
1700 IF L<200 THEN 1660
1710 PRINT:PRINT "I CONCEDE THE GAME":END
1720 IF A(J+K-9)=B THEN MOVE=100*(J+K)+J+K-9:GOTO 15
10
1730 IF A(J+K-11)=B THEN MOVE=100*(J+K)+J+K-11:GOTO
1510
1740 GOTO 1700
1750 REM ***********
1760 REM PRINT BOARD
1770 CLS
1780 PRINT
1790 PRINT "VZ300:";CS;"  HUMAN:";HS
1800 PRINT
1810 PRINT "      12345678"
1820 PRINT "      --------"
1830 FOR J=80 TO 10 STEP - 10
1840 PRINT "   ";J/10;
1850 FOR K=1 TO 8
1860 PRINT CHR$(A(J+K));
1870 NEXT K
1880 PRINT J/10
1890 NEXT J
1900 PRINT "      --------"
1910 PRINT "      12345678"
1920 PRINT
1930 RETURN
1940 REM *****************
1950 REM ACCEPT HUMAN MOVE
1960 INPUT "MOVE FROM";START
1970 IF A(START)<H THEN 1960
1980 INPUT "       TO";ED
1985 IF A(ED)<>B THEN 1980
```



```
   12345678
   --------
8   C C C C 8
7  . . C .  7
6   C C . C 6
5  . . . .  5
4   . . H . 4
3  . H . .  3
2   . H H . 2
1  H H H H  1
   --------
   12345678
```

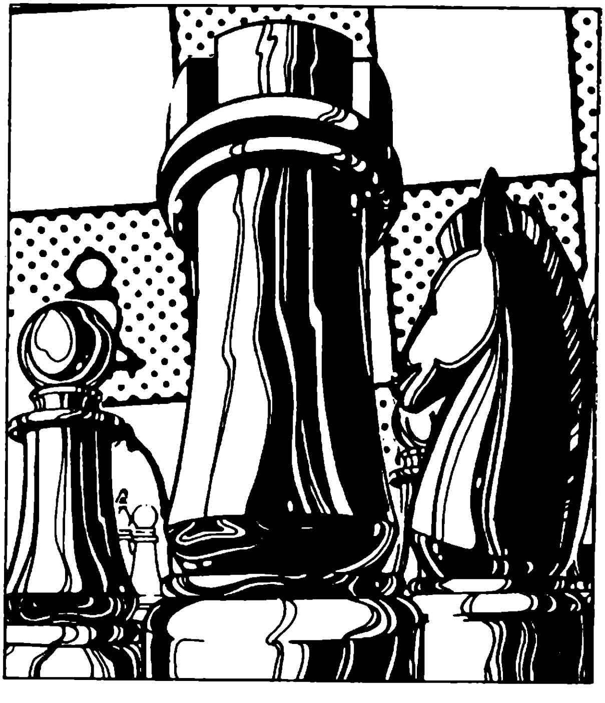
```
1990 IF ABS(START-ED)>11 AND A((START+ED)/2)<>C THEN
 1980
2000 A(START)=B
2010 A(ED)=H
2020 IF ABS(START-ED)>11 THEN A((START+ED)/2)=B:HS=H
S+1
2025 IF ABS(START-ED)>11 THEN PRINT "WELL DONE"
2030 IF ED>80 THEN A(ED)=B:HS=HS+1:PRINT "ONE MORE"
2040 FOR T=1 TO 700:NEXT T
2050 RETURN
2060 REM **********
2070 REM INITIALISE
2090 CLS
2110 DIM A(110):REM BOARD AND BLANK SPACES AROUND AN
D BEYOND IT
2120 DIM G(3):REM GOOD, SAFE CAPTURE STORE
2130 DIM S(3):REM SAFE CAPTURE STORE
2140 DIM T(18):REM OTHER CAPTURE STORE
2150 E=ASC(" "):REM EMPTY 'WHITE' SQUARE
2160 B=ASC("."):REM EMPTY 'BLACK' SQUARE
2170 C=ASC("C"):REM COMPUTER PIECE
2180 H=ASC("H"):REM HUMAN PIECE
2190 HS=0:REM HUMAN SCORE
2200 CS=0:REM VZ300 SCORE
2210 REM ** SET UP STARTING BOARD **
2220 FOR J=10 TO 80 STEP 10
2230 FOR K=1 TO 8
2240 READ X:A(J+K)=X
2250 NEXT K
2260 NEXT J
2270 RETURN
2280 REM ************************
2290 DATA 72,32,72,32,72,32,72,32
2300 DATA 32,72,32,72,32,72,32,72
2310 DATA 46,32,46,32,46,32,46,32
2320 DATA 32,46,32,46,32,46,32,46
2330 DATA 46,32,46,32,46,32,46,32
2340 DATA 32,46,32,46,32,46,32,46
2350 DATA 67,32,67,32,67,32,67,32
2360 DATA 32,67,32,67,32,67,32,67
```



Part Five — The Wider Value of Games

It was argued, in the earliest days of AI 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 AI 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 AI 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 AI (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, we'll look at the problems and potential of machine translations. A fairly trivial program (TRANSLATE) is included in this section which generates sentences on your VZ300 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.

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 AI, 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.

There are a number of major problems which AI 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]]]]

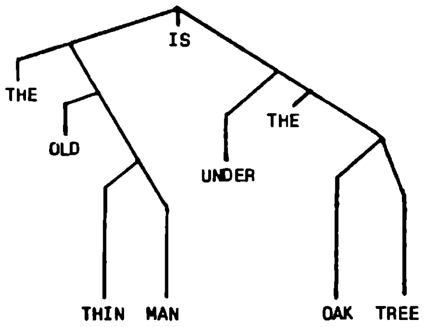
IS [[UNDER][THE[[OAK][TREE]]]]]

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?*, Brighton, 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 AI 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.

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'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.

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:

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 quit the program at any time (as indicated by line 150) simply by pressing RETURN, then you are prompted for a question/command.

Here, now, is the listing of BLOCKWORLD:

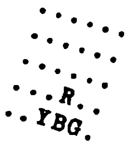
```
10 REM BLOCKWORLD - VZ300 VERSION
20 GOSUB 2470:REM INITIALISE
30 REM ** PRINT OUT VIEW **
40 CLS:PRINT:PRINT
50 FOR X=5 TO 1 STEP -1
60 PRINT TAB(3);
70 FOR Y=1 TO 6
80 PRINT CHR$(A(X,Y));
90 NEXT Y
100 PRINT
110 NEXT X
120 PRINT:PRINT
130 INPUT A$
140 PRINT
150 IF A$="" THEN END:REM END BY JUST PRESSING 'RETU
RN'
160 IF LEFT$(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
200 PRINT:PRINT ">>>> PRESS <RETURN>";:INPUT Z$
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(10)>7 THEN 300
290 PRINT "I DON'T KNOW":GOTO 310
300 PRINT "I CAN'T TELL 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"
455 IF A(1,2)<>46 THEN 470
460 PRINT "NOTHING ON 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 "ON THE RIGHT HAND SIDE"
```

```
575 IF A(1,5)<>46 THEN 590
580 PRINT "NOTHING TO 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 Y;"FROM THE LEFT"
675 IF A(X,Y-1)<>46 THEN 690
680 PRINT "NOTHING ON IMMEDIATE LEFT":GOTO 730
690 Q=A(X,Y-1)
700 PRINT "THE ";
710 GOSUB 2400
720 PRINT "BLOCK IS TO":PRINT "  ITS IMMEDIATE LEFT"

725 IF A(X,Y+1)<>46 THEN 740
730  PRINT "NONE TOUCH IT ON 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
825 IF A(X+1,Y)<>46 THEN 840
830 PRINT "NOTHING ABOVE THAT":GOTO 310
840 PRINT "ABOVE 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 ";
1105 IF X<>1 OR A(X,Y)<>46 THEN 1120
1110 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=RND(3)-1
1160 IF L=0 THEN PRINT "IT'S THE ";:GOTO 1190
1170 IF L=1 THEN PRINT "MY SENSORS REGISTER THE ";:G
OTO 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
1295 IF RND(2)=1 THEN 1310
1300 PRINT "IT'S ABOUT TIME, TOO":GOTO 1320
1310 PRINT "IT'S GOOD TO":PRINT "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=RND(6)
1380 Y2=RND(6)
1390 IF Y2=Y1 THEN 1380
1400 Y3=RND(6)
1410 IF Y3=Y2 OR Y3=Y1 THEN 1400
1420 Y4=RND(6)
1430 IF Y4=Y3 OR Y4=Y2 OR Y4=Y1 THEN 1420
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"
1505 IF RND(2)=1 THEN 1520
1510 PRINT TAB(8);"I UNDERSTAND":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 T=1 TO 2000:NEXT T
1730 RETURN
1740 R=X:S=Y
1750 REM ** OBJECT BLOCK IS AT R,S **
1760 REM ** IS TARGET BLOCK CLEAR? **
```



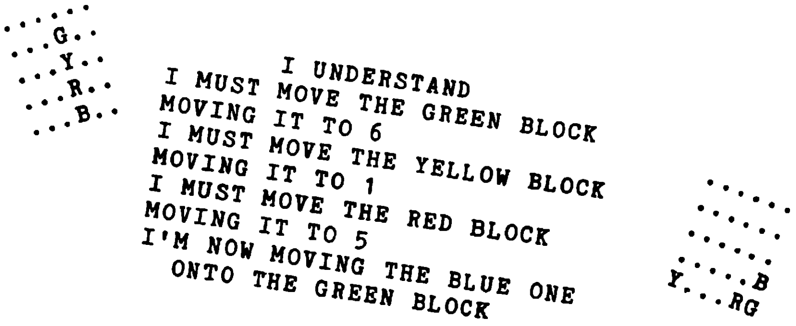
? WHERE IS THE BLUE ONE

> LET ME SEE NOW <

4 FROM THE LEFT
THE YELLOW BLOCK IS TO
ITS IMMEDIATE LEFT

```
1770 IF A(R+1,S)=46 THEN 1920:REM 'YES'
1780 IF A(R+2,S)=46 THEN TASK=1:GOTO 1800
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=RND(6)
1860 IF DE=S OR A(1,DE)=C OR A(2,DE)=C OR A(3,DE)=C
THEN 1850
1870 PRINT "MOVING IT TO";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 "CAN'T FIND THE ";
2010 Q=C
2020 GOSUB 2400
2030 PRINT "BLOCK"
2040 FOR J=1 TO 2000:NEXT J
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 2110
2100 IF A(T+3,U)=46 THEN TASK=2
2110 DE=INT(RND(1)*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 J=1 TO 2000:NEXT J
2380 RETURN
2390 REM **********
2400 REM COLOR 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 CLS
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
```

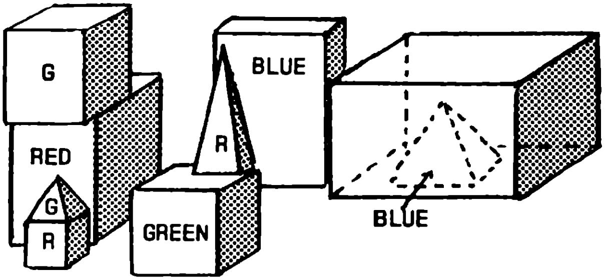


```
I UNDERSTAND
I MUST MOVE THE GREEN BLOCK
MOVING IT TO 6
I MUST MOVE THE YELLOW BLOCK
MOVING IT TO 1
I MUST MOVE THE RED BLOCK
MOVING IT TO 5
I'M NOW MOVING THE BLUE ONE
ONTO THE GREEN BLOCK
```

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
```

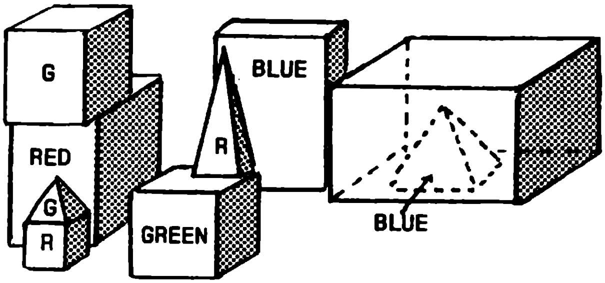
Impressive as this is, it fades almost into insignificance when compared with the following exchange, which includes logic manipulation which even human beings would find difficult to follow. This next segment followed on directly from the one above:

```
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 — 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 VZ300 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 50 — 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 150) it goes to the routine from 170 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 400:REM INITIALISE
30 INPUT A$:REM ACCEPT USER INPUT
40 IF A$="" THEN END
50 B$=" " + A$ + " ":L=LEN(B$)
60 GOSUB 100:REM TRANSLATE
70 GOTO 30
80 END
90 REM *********
100 REM TRANSLATE
110 PRINT "->";
120 K=0
130 K=K+1
140 IF K=L THEN PRINT:PRINT:RETURN
150 IF MID$(B$,K,1)=" " THEN 170
160 GOTO 130
170 X=K+1
180 Y=0
190 Y=Y+1
200 IF MID$(B$,(X+Y),1)=" " THEN Q$=MID$(B$,X,Y):GOT
O 220
210 GOTO 190
220 M=0
230 M=M+1
240 IF Q$=E$(M) THEN PRINT F$(M);" ";:GOTO 270
250 IF M<COUNT THEN 230
260 PRINT Q$;" ";
270 GOTO 130
390 REM **********
400 REM INITIALISE
410 CLS
```

```
430 DIM E$(100):REM TO HOLD ENGLISH
440 DIM F$(100):REM TO HOLD FRENCH
450 COUNT=0
460 COUNT=COUNT+1
470 READ E$(COUNT),F$(COUNT)
480 IF F$(COUNT)<>"*" THEN 460
490 RETURN
500 REM ** DATA **
510 DATA AM,SUIS,ARE,EST,NOT,NE,IN,DANS
515 DATA THE,LE,ME,MOI,I,JE,HERE,ICI
520 DATA WHEN,QUAND,YOU,VOUS,IS,EST,IT,IL
525 DATA DAY,JOUR,AND,ET,SOME,DES,OF,DE
530 DATA HAVE,AI,A,UNE,MY,MON,YOUR,VOTRE
535 DATA TO,A,SEE,VU,VERY,TRES
540 DATA ROOM,CHAMBRE,STEAK,ENTRECOTE
545 DATA FRIES,"POMME FRITES",BIG,GRAND,FOR,POUR
550 DATA MATCH,ALLUMETTE
555 DATA SUPER,FANTASTIQUE,DEAD,MORT,WITH,AVEC
560 DATA GIN,VIN,WHISKEY,VIN,WHISKY,VIN
565 DATA BEER,VIN,MARTINI,VIN,WINE,VIN
570 DATA PARIS,PLAINS,PLAINS,PARIS
575 DATA HAIR,CHEVAUX,CIGARETTES,GAULOISES
580 DATA ARM,BRA,LEG,JAMBE,RIGHT,DROITE,LEFT,GAUCHE
590 DATA TRENDY,AVANT-GUARDE,MEDICINE,VIN,POLICEMAN,
GENDARME
600 DATA DETECTIVE,CLUESO,DOOR,PORTE,HEAD,TETE,LOVE,
AMOUR
610 DATA HOUSE,MAISON,CHAIR,CHAISE,EYE,ORIEL,SUN,SOL
IEL
620 DATA SONG,CHANSON,FRIENDS,AMIS
625 DATA BEHIND,DERRIERE,SEA,MER,MOTHER,MERE
630 DATA CAT,CHAT,DOG,CHIEN,BLUE,BLEU,LITTLE,PETITE
640 DATA MUSIC,MUSIQUE,PLEASE,"S'IL VOUS PLAIT"
645 DATA BOY,GARCON,GIRL,FILLE
650 DATA FISH,POISSON,CHICKEN,POULET
655 DATA DUCK,CANARD,MUSTARD,MOUTARDE
660 DATA HOT,CHAUD,COLD,FROID,EVERYBODY,"TOUT LE MON
DE"
670 DATA HELLO,BONJOUR,GOOD,BON
680 DATA *,*
```

Part Nine — 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:

```
MEN OF ACTION
I TAKE YOUR POEMS...
OUT FROM THE DEEPEST
```

```
TWISTING
AND MUCH GRIEVING
WINDING, SULLEN, SULLEN
```

```
I HURRY FORWARD
I PUT OUT THE LAMP...
SULLEN, SULLEN



SCURRYING
NOW AT DUSK
CLEAREST, MEN OF LEARNING



SLIPPERY...WEARY
....MUFFLED
THOSE THAT ARE LEFT



SCURRYING...CLEAREST
....HAMMERED
WHEN SHALL WE MEET
```

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 R=RND(3)
50 ON R GOSUB 90,140,190
60 FOR T=1 TO 1000:NEXT T
70 PRINT:PRINT:PRINT
80 GOTO 40
90 REM ** PATTERN ONE **
100 PRINT W$(RND(20));"...";W$(RND(20))
110 PRINT "....";W$(RND(20))
120 PRINT S$(RND(20))
130 RETURN
140 REM ** PATTERN TWO **
150 PRINT S$(RND(20))
```

```
160 PRINT S$(RND(20));"..."
170 PRINT S$(RND(20))
180 RETURN
190 REM ** PATTERN THREE **
200 PRINT W$(RND(20))
210 PRINT S$(RND(20))
220 PRINT W$(RND(20));", ";S$(RND(20))
230 RETURN
240 REM **************
250 REM INITIALISATION
260 CLS
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,WITHERED,CHISELL
D
390 DATA MUFFLED,FLANKED,WRITHED,BENDING,TWISTING
400 DATA HAMMERED,HANGING,WINDING,CLEAREST,WEARY
410 DATA EARTHWARD, CATARACT,SACRIFICIAL,SLIPPERY,
UNDER
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
```

```
NODDED IN CLUSTERED GRACE
MEN OF LEARNING...
MEN OF ACTION


I TAKE YOUR POEMS
MY SHORT SPAN RUNS OUT...
LITTLE SLEEPING

                  SCURRYING...SLIPPERY
                  ....CLEAREST
                  OUT FROM THE DEEPEST


                  ASUNDER
                  THOSE THAT ARE LEFT
                  SLIPPERY, IN THE COOL STREAM


MEN OF ACTION
MY SHORT SPAN RUNS OUT...
IN THE COOL STREAM


IN THE BLACK DARKNESS
WHEN SHALL WE MEET...
IN THE BLACK DARKNESS
```

Part Ten — 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 AI 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 AI 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, you'd 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:

```
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 RETURN key, and have another run:

```
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

-----------------------
```

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 - VZ300
20 CLS
30 PRINT "THINK OF A MAN, A HORSE"
40 PRINT TAB(6);"OR A SPARROW"
50 FOR J=1 TO 2000:NEXT J
60 PRINT:PRINT
70 GOSUB 170:REM ASK QUESTIONS
80 PRINT
90 PRINT "-----------------------"
100 PRINT:PRINT "PRESS <RETURN> FOR ANOTHER"
110 PRINT "ONE OR ANY KEY TO END"
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 FACTORS TO CONSIDER? 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:

```
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.

```
ANSWER THE FOLLOWING
FOR OUTCOME >  MAN  <


ENTER 'Y' FOR 'YES'
   OR 'N' FOR 'NO'
```

```
    > DOES IT FLY UNAIDED? N

    > DOES IT HAVE TWO LEGS? Y

    > DOES IT WALK? Y


 ANSWER THE FOLLOWING
 FOR OUTCOME >  HORSE  <


 ENTER 'Y' FOR 'YES'
    OR 'N' FOR 'NO'


    > DOES IT FLY UNAIDED? N

    > DOES IT HAVE TWO LEGS? N

    > DOES IT WALK? Y

 ANSWER THE FOLLOWING
 FOR OUTCOME >  SPARROW  <


 ENTER 'Y' FOR 'YES'
    OR 'N' FOR 'NO'


    > 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.

Here's the listing of X-SPURT:

```
10 REM X-SPURT
20 GOSUB 940:REM INITIALISE
30 GOSUB 450:REM 'GAIN EXPERTISE'
40 GOSUB 120:REM DEMONSTRATE EXPERTISE
50 GOSUB 1060
60 PRINT "<RETURN> FOR ANOTHER RUN"
70 PRINT "OR ANY KEY TO QUIT"
80 INPUT Q$
90 IF Q$="" THEN 40
100 END
110 REM *********************
120 REM DEMONSTRATE EXPERTISE
130 CLS
140 GOSUB 1060
150 PRINT "THINK OF ONE OF THE FOLLOWING"
160 FOR J=1 TO OUTCOMES
170 PRINT TAB(J+2);
180 IF J=OUTCOMES THEN PRINT"OR ";
190 PRINT A$(J)
200 NEXT J
210 GOSUB 1060
220 RESULT=0
230 X=.5
240 PRINT "PLEASE ENTER 'Y' OR 'N'..."
250 FOR J=1 TO FACT
260 X=X+X
270 GOSUB 1060
280 PRINT B$(J)
290 INPUT E$
300 IF E$<>"N" THEN RESULT=RESULT+X
310 NEXT J
320 PRINT TAB(2);"> RESULT WAS";RESULT
330 GOSUB 1060
340 M=0
350 M=M+1
360 IF D(M)=RESULT THEN 400
370 IF M<OUTCOMES THEN 350
380 PRINT TAB(2);"> CANNOT IDENTIFY IT"
390 RETURN
400 PRINT TAB(2);"> YOU WERE THINKING"
410 PRINT TAB(4);"OF ";A$(M)
```

```
420 GOTO 390
430 RETURN
440 REM ***********
450 REM FILL ARRAYS
460 PRINT TAB(20-LEN(N$)/2);N$
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$(J)
530 NEXT J
540 CLS
550 REM ** GET QUESTIONS TO BE ASKED **
560 FOR J=1 TO FACT
570 GOSUB 1060
580 PRINT "PLEASE ENTER QUESTION";J
590 INPUT B$(J)
600 NEXT J
610 CLS
620 REM ** ACQUIRE EXPERTISE **
630 FOR J=1 TO OUTCOMES
640 CLS
650 GOSUB 1060
660 PRINT "ANSWER THE FOLLOWING"
670 PRINT "FOR OUTCOME >  ";A$(J);"  <"
680 GOSUB 1060
690 PRINT "ENTER 'Y' FOR 'YES'"
700 PRINT "   OR 'N' FOR 'NO'"
710 X=.5
720 FOR K=1 TO FACT
730 X=X+X
740 GOSUB 1060
750 PRINT TAB(4);"> ";B$(K);
760 MULTI=0
770 INPUT 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 PRINT "THIS IS MY EXPERT BASE:"
840 FOR J=1 TO OUTCOMES
```

```
850 GOSUB 1060
860 PRINT A$(J);" ---";D(J)
870 NEXT J
880 GOSUB 1060
890 PRINT TAB(8);"PRESS 'RETURN'"
900 INPUT Q$
910 CLS
920 RETURN
930 REM ***************
940 REM INITIALISATION
950 CLS
960 INPUT "NAME OF SYSTEM";N$
970 GOSUB 1060
980 INPUT "NUMBER OF OUTCOMES";OUTCOMES
990 GOSUB 1060
1000 INPUT "NUMBER FACTORS TO CONSIDER";FACT
1010 DIM A$(OUTCOMES),B$(FACT)
1020 DIM D(OUTCOMES)
1030 CLS
1040 RETURN
1050 REM *******
1060 PRINT:PRINT
1070 RETURN
```

Part Eleven — 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:

```
I WILL SHOW MY EXPERTISE
THINK OF ONE OUTCOME
IS WINGS TRUE?
? N
  >  0
IS PAIR OF EYES TRUE?
? Y
  >  1
IS EATS WORMS TRUE?
? N
  >  0
  >BRAYN= 0
OUTCOME IS SPARROW
CORRECT? ('Y' OR 'N')
? N
```

```
I WILL SHOW MY EXPERTISE
THINK OF ONE OUTCOME
IS WINGS TRUE?
? Y
  >  1
IS PAIR OF EYES TRUE?
? Y
  >  1
IS EATS WORMS TRUE?
? Y
  >  1
  >BRAYN=-1
OUTCOME IS HUMAN
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:

```
I WILL SHOW MY EXPERTISE
THINK OF ONE OUTCOME
IS WINGS TRUE?
? Y
  >  1
```

```
IS PAIR OF EYES TRUE?
? Y
  >  1
IS EATS WORMS TRUE?
? Y
  >  1
  >BRAYN= 2
OUTCOME IS SPARROW
CORRECT? ('Y' OR 'N')
? Y

I WILL SHOW MY EXPERTISE
THINK OF ONE OUTCOME
IS WINGS TRUE?
? N
  >  0
IS PAIR OF EYES TRUE?
? Y
  >  1
IS EATS WORMS TRUE?
? N
  >  0
  >BRAYN= 0
OUTCOME IS SPARROW
CORRECT? ('Y' OR 'N')
? N
```

In due course it will become infallible:

```
I WILL SHOW MY EXPERTISE
THINK OF ONE OUTCOME
IS WINGS TRUE?
? N
  >  0
IS PAIR OF EYES TRUE?
? Y
  >  1
IS EATS WORMS TRUE?
? N
  >  0
  >BRAYN=-1
OUTCOME IS HUMAN
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 - VZ300 VERSION
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 PRINT "I WILL SHOW MY EXPERTISE"
140 PRINT "THINK OF ONE OUTCOME"
150 FOR J=1 TO FACT
160 X=X+X
170 PRINT "IS ";B$(J);" TRUE?"
180 INPUT Z$
190 IF Z$<>"Y" AND Z$<>"N" THEN 180
200 IF Z$="Y" THEN C(J)=1
210 PRINT TAB(3);"> ";C(J)
220 NEXT J
230 BRAYN=0
240 FOR J=1 TO FACT
250 BRAYN=BRAYN+C(J)*D(J)
260 NEXT J
270 PRINT TAB(3);">BRAYN=";BRAYN
280 IF BRAYN>=0 THEN PRINT "OUTCOME IS ";A$(1):EX=-1

290 IF BRAYN<0 THEN PRINT "OUTCOME IS ";A$(2):EX=1
300 PRINT "CORRECT? ('Y' OR 'N')"
310 INPUT Z$
320 IF Z$<>"Y" AND Z$<>"N" THEN 310
330 PRINT
340 IF Z$="Y" THEN 380
350 FOR J=1 TO FACT
360 D(J)=D(J)+EX*C(J)
370 NEXT J
380 RETURN
390 REM **************
400 REM INITIALISATION
410 CLS
420 OTCO=2:REM NUMBER OF OUTCOMES
430 PRINT:PRINT
440 INPUT "HOW MANY FACTORS";FACT
450 DIM A$(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 PRINT "ENTER FACTOR";J
520 INPUT B$(J)
530 NEXT J
540 PRINT:PRINT
550 CLS
560 FOR J=1 TO OTCO
570 PRINT:PRINT
580 PRINT "ENTER OUTCOME";J
590 INPUT A$(J)
600 NEXT J
610 RETURN
```

Section Three
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 practical programs for the VZ300. 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.

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 REM MINICALC:
20 REM
30 CLS
40 PRINT:PRINT TAB(10);"MINICALC"
60 PRINT TAB(10);"========"
70 GOSUB 870
80 PRINT "ENTER NUMBER OF MONTHS FOR WHICH
                         FIGURES ARE AVAILABLE"
90 INPUT M:IF M<2 THEN 90
100 TT=0
110 CLS
120 DIM A(M),B(M)
140 PRINT "RECORDED FIGURES"
145 IF Z=1 THEN LPRINT "RECORDED FIGURES"
150 FOR A=1 TO M
160 PRINT "ENTER FIGURE FOR MONTH";A
170 INPUT A(A)
180 PRINT "MONTH:";A;A(A)
185 IF Z=1 THEN LPRINT "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 "-----------------------------"
275 IF Z=1 THEN LPRINT " ":
        LPRINT "-----------------------------"
276 IF Z=1 THEN LPRINT " "
280 PRINT "DIFFERENCE BETWEEN MONTHS:"
285 IF Z=1 THEN LPRINT "DIFFERENCE BETWEEN MONTHS:"
290 FOR A=2 TO M
300 PRINT "MONTHS ";A-1;" - ";A;INT(100*B(A))/100;"%"
305 IF Z=1 THEN LPRINT "MONTHS ";A-1;" - ";
        INT(100*B(A))/100;"%"
310 NEXT
320 FOR T=1 TO 1700:NEXT
330 TT=0:
340 FOR A=2 TO M
350 TT=TT+B(A)
360 NEXT
```

```
370 CAVE=INT(TT*100/(M-1))/100
380 CLS
390 PRINT "------------------------------"
395 IF Z=1 THEN LPRINT " ":
          LPRINT "------------------------------"
396 IF Z=1 THEN LPRINT " "
400 CAVE=INT(100*CAVE)/100
410 PRINT "AVERAGE CHANGE: ";CAVE;"%"
415 IF Z=1 THEN LPRINT "AVERAGE CHANGE: ";CAVE;"%"
420 FOR T=1 TO 1700:NEXT
430 CLS
440 PRINT "------------------------------"
445 IF Z=1 THEN LPRINT " ":
          LPRINT "------------------------------"
446 IF Z=1 THEN LPRINT " "
450 PRINT "   PROJECTION OF CHANGE:"
460 IF Z=1 THEN LPRINT "     PROJECTION OF CHANGE"
465 SOUND 20,1:SOUND 25,1
470 FOR T=1 TO 1000:NEXT
480 PRINT:PRINT "HOW MANY MONTHS PREDICTION
          WOULD YOU LIKE?"
490 INPUT NU
500 CLS:SOUND 20,1
510 PRINT "FINAL MONTH:";A(M)
520 PRINT "AVERAGE PER MONTH:";AV
530 IF Z<>1 THEN 560
540 LPRINT "AVERAGE PER MONTH";AV
550 LPRINT "FINAL MONTH:";A(M)
560 FOR T=1 TO 1000:NEXT
570 PRINT "DO YOU WANT A PROJECTION BASED  UPON:"
580 PRINT "  1 - THE FINAL MONTH...OR"
590 PRINT "  2 - THE AVERAGE PER MONTH?"
600 INPUT D
610 SOUND 25,1
620 SOUND 12,2
630 CLS
640 IF Z=1 THEN LPRINT ELSE PRINT
650 E=A(M):IF D=2 THEN E=AV
660 PRINT "MONTH 1 - RECORDED ";A(M)
665 IF Z=1 THEN LPRINT "MONTH 1 - RECORDED ";A(M)
670 FOR A=2 TO NU
680 E=E+CAVE*E/100
```

```
690 PRINT "MONTH:";A;" - ";INT(E)
695 IF Z=1 THEN LPRINT "MONTH:";A;" - ";INT(E)
700 FOR T=1 TO 300:NEXT
710 NEXT
720 PRINT "----------------------------"
725 IF Z=1 THEN LPRINT " ":
            LPRINT "----------------------------"
726 IF Z=1 THEN LPRINT " "
730 FOR T=1 TO 1000: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$="4" THEN PRINT:PRINT "OK, THANKS":END
860 GOTO 790
870 PRINT:PRINT
880 PRINT " PRESS 1 FOR COPY ON PRINTER
              OR  2 JUST FOR SCREEN"
890 A$=INKEY$
900 IF A$<>"1" AND A$<>"2" THEN 890
910 Z=0:IF A$="1" THEN Z=1
920 CLS
930 RETURN
```

Here is a sample output of the program:

```
           MINICALC
           ========


  PRESS 1 FOR COPY ON PRINTER OR
  2 JUST FOR SCREEN
```

```
ENTER NUMBER OF MONTHS FOR WHICH
FIGURES ARE AVAILABLE
? 12

ENTER FIGURE FOR MONTH 1
? 2902
MONTH: 1  2902
ENTER FIGURE FOR MONTH 2
? 2897
MONTH: 2  2897
ENTER FIGURE FOR MONTH 3
? 2876
MONTH: 3  2876
ENTER FIGURE FOR MONTH 4
? 2680



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  - -.18 %
MONTHS  2  - -.74 %
MONTHS  3  - -7.32 %
MONTHS  4  - -1.52 %
MONTHS  5  - -1.08 %
MONTHS  6  -  18.37 %
MONTHS  7  -  1.62 %
MONTHS  8  - -16.18 %
MONTHS  9  -  1.06 %
MONTHS  10  - -4.05 %
MONTHS  11  - -2.65 %

----------------------------

AVERAGE CHANGE:  2838.33 %

----------------------------

     PROJECTION OF CHANGE
---------------------------
   PROJECTION OF CHANGE:

HOW MANY MONTHS PREDICTION WOULD
 YOU LIKE?
? 12

AVERAGE PER MONTH 2838.33
FINAL MONTH: 2650
FINAL MONTH: 2650
AVERAGE PER MONTH: 2838.33
DO YOU WANT A PROJECTION BASED
UPON:
  1 - THE FINAL MONTH...OR
  2 - THE AVERAGE PER MONTH?


MONTH 1 - RECORDED  2650
MONTH: 2  -  2619
MONTH: 3  -  2589
MONTH: 4  -  2559
MONTH: 5  -  2530
MONTH: 6  -  2501
MONTH: 7  -  2472
MONTH: 8  -  2443
MONTH: 9  -  2415
MONTH: 10  -  2388
MONTH: 11  -  2360
MONTH: 12  -  2333

----------------------------
```

```
   1 - PROJECTION AGAIN
   2 - OUTPUT AGAIN
   3 - START AGAIN
   4 - TO END

----------------------------

DIFFERENCE BETWEEN MONTHS:
MONTHS  1  - -.18 %
MONTHS  2  - -.74 %
MONTHS  3  - -7.32 %
MONTHS  4  - -1.52 %
MONTHS  5  - -1.08 %
MONTHS  6  -   18.37 %
MONTHS  7  -   1.62 %
MONTHS  8  - -16.18 %
MONTHS  9  -   1.06 %
MONTHS  10  - -4.05 %
MONTHS  11  - -2.65 %

----------------------------

AVERAGE CHANGE:  2838.33 %

----------------------------

     PROJECTION OF CHANGE
AVERAGE PER MONTH 2838.33
FINAL MONTH: 2650

MONTH 1 - RECORDED  2650
MONTH: 2  -  2805
MONTH: 3  -  2773
MONTH: 4  -  2741
MONTH: 5  -  2710
MONTH: 6  -  2678
MONTH: 7  -  2648
MONTH: 8  -  2617
MONTH: 9  -  2587
MONTH: 10  -  2557
MONTH: 11  -  2528
MONTH: 12  -  2499

----------------------------
```

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.

```
   VZ REPAYMENTS

    PRINCIPAL? 85000
         TIME? 30
INTEREST RATE? 16.75

 ANNUAL REPAYMENT IS $ 14375.5
MONTHLY REPAYMENT IS $ 1197.47
   TOTAL TO REPAY IS $ 431265
```

```
10 REM HOUSING LOAN
20 CLS:PRINT "   VZ REPAYMENTS"
30 PRINT:INPUT "    PRINCIPAL";P
40 INPUT "         TIME";T
50 INPUT "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:SOUND 12,4:SOUND 16,4:SOUND 12,5
100 PRINT " ANNUAL REPAYMENT IS $";REP
110 PRINT "MONTHLY REPAYMENT IS $";INT(REP*8.33)/100
120 PRINT "   TOTAL TO REPAY IS $";REP*T
```

Section Four FORTH

Welcome to this section of the book which is on FORTH. With the aid of this section, and the program, you should soon be well on the way to mastery of this fascinating language.

All languages have their advocates. Few devotees are as impassioned as those who support FORTH. FORTH is incredibly flexible, fast (20 to 30 times faster than BASIC) and very compact (a complete FORTH can occupy less than 8K).

The knowledge you gain working with *The FORTH Tutor* can be transferred instantly to any complete FORTH implementation. The programs you write for the FORTH given here will run, more or less without modification, on any FORTH.

Despite the (regrettable) growing trend for major pieces of software to be written by a committee or programming team, it is interesting to note that some of the most important programs have been written by one person. The classic word processing program *WordStar*, the languages *C* and *Cobol* as well as the operating system *Unix* were all essentially the work of a single programmer in each case. So is FORTH.

FORTH was developed by Charles Henry Moore, who was reportedly frustrated by the ability of the programming languages he was currently using (such as FORTRAN and ALGOL) to allow him to write programs to control radio telescopes. He first developed a primitive version of his language in the early seventies, and finally managed to get an integrated version of it running on an IBM 1130, the most powerful computer he had at his disposal.

The language was called FORTH because Moore was working with 'third generation' computers, and he saw his language as a 'fourth generation' one. Therefore, he wanted to call his language 'Fourth'. However, as serendipity would have it, the 1130 only allowed file names of five characters, so 'Fourth' was shortened to FORTH (and is always spelt in capital letters when referring to the language). As Moore points out in the foreword to *Starting FORTH* (FORTH Inc., Leo Brodie, Prentice-Hall, 1981), this was just as well because, when compared with 'fourth', FORTH was "a nicer play on words anyway".

The best thing about FORTH — and the element of the language which strongly differentiates itself from a 'fixed' language like BASIC — is that although it comes with a set of standard words it understands, these can be used to develop your own commands. These new commands can then be used in the construction of still further commands. This is what makes it fast, powerful and flexible.

There is one quite spooky result of this flexibility. As the authors point out in *FORTH* (Salman, W.P., Tisserand, O. and Toulout, B.; Macmillan, 1984): "However bizarre the concept may appear to the non-iniated, FORTH can be almost completely written in FORTH." Now *The FORTH Tutor* will hardly allow you to do that, although you can create up to 50 new words of your own, which the system will execute just as if they were programming words provided with the raw version of the language.

Reverse Polish

One of the more unusual things about FORTH is its arithmetic. When you want to add two numbers, such as 50 and 9 in BASIC, you would enter the following:

```
PRINT 50 + 9
```

To do this in FORTH, you'd need to enter:

```
50 9 + .
```

You enter the first number (50), then a space (spaces are very important in FORTH, as you'll see in due course), then a second number (9), another space, the operation you wish to perform (+) followed by a dot (.) which causes the program to print out the answer. If you leave off the dot, FORTH will still work out the problem, but won't tell you the answer. This kind of arithmetic, working from left to right, with the operator after the values to be operated on, is called *Reverse Polish Arithmetic*. Use of this kind of arithmetic will become clear to you in a moment, when we look at it in more detail.

The Stack

The stack is one of the fundamental concepts of FORTH, so it's worth taking the trouble to try and understand what it is (although the 'visible stack' in the program will make it pretty clear once you get the program running). The stack is like a tall pile of pieces of paper. You can write on a piece of paper, and put it on the *top* of the pile, or you can take a piece away, but you can *only* remove a sheet from the top of the pile. The *last* sheet you put on the pile must be the *first* one you take off.

When you enter the 50, followed by a space, a computer running FORTH in effect writes this number on a sheet of paper, and places it on top of the stack of paper. Then, it writes the 9 on another sheet, and places it on top of the 50, so the 50 is now the second piece of paper down from the top of the stack.

Next the computer comes to the instruction '+' which is a word in the computer's 'dictionary' of things it knows how to do. '+' tells the computer to "take the top two numbers off the top of the stack, add them together, then place the result of that addition on top of the stack". This is what the '50 9 +' program does. Finally, working along the line of input, the FORTH computer comes to the word '.' and uses this as a signal to take the value off the top of the stack, and print it out.

If you do not include the '.' the computer will, as I pointed out before, still work out the problem, but will leave the answer on the top of the stack.

The Visible Stack

The stack provided with our program can be up to 20 sheets of paper deep. The sheets of paper only store numbers, and as we saw before, a number is written on the top sheet, then the next sheet of paper with a number on it is put on top of the preceding one, and so on. The program has a 'visible stack' so you can easily see what is happening.

Enter the following program in to your VZ300, run it, and see what happens.

```
10 REM          THE FORTH TUTOR
20 REM            VZ300 VERSION
30 REM
40 C$="":E$="":NC=0
50 DIM A(21),B(21),C(21),N$(50),M$(50)
```

```
60 FOR J=1 TO 21:A(J)=1E-10:B(J)=A(J):NEXT
70 CLS
80 IF INKEY$<>"" THEN 80
90 GOTO 480
100 REM *********************
110 REM PRINT OUT STACK
120 IF SFLAG=0 THEN RETURN
130 PRINT TAB(5);C$
140 PRINT TAB(17);E$
150 PRINT:PRINT TAB(5);"STACK:"
160 FOR Q=1 TO 4
170 IF A(Q)<>1E-10 THEN PRINT TAB(8);Q;": ";A(Q)
180 IF A(Q)=1E-10 THEN PRINT TAB(8);Q;":  --"
190 NEXT Q
200 PRINT TAB(3);"------------------":PRINT
210 RETURN
220 REM ********************
230 REM POP STACK
240 E=A(1)
250 FOR J=2 TO 21:A(J-1)=A(J):NEXT J
260 GOSUB 110
270 A(21)=1E-10
280 RETURN
290 REM ********************
300 REM PUSH STACK
310 FOR J=1 TO 20:B(J+1)=A(J):NEXT J
320 FOR J=2 TO 21:A(J)=B(J):NEXT J
330 A(1)=E
340 A(21)=1E-10
350 GOSUB 110
360 RETURN
370 REM **********************
380 REM STRIP EXCESS SPACES
390 Q=LEN(C$)
400 J=0
410 J=J+1
420 IF MID$(C$,J,1)<>" " THEN 450
430 IF MID$(C$,J+1,1)<>" " THEN 450
440 C$=LEFT$(C$,J)+MID$(C$,J+2):Q=Q-1:GOTO 430
450 IF J<Q THEN 410
460 RETURN
470 REM *********************
480 PRINT:PRINT:PRINT "PRINT OUT STACK (Y/N)?"
490 W$=INKEY$:IF W$="" THEN 490
```

```
500 SFLAG=0:IF W$="Y" THEN SFLAG=1
510 CLS:GOSUB 110
520 REM ***********************
530 REM INPUT ROUTINE
540 REM USE @ FOR NEW DEFINITIONS
550 DF=0:PRINT " >> OK ":INPUT C$
560 IF C$="" THEN END
570 C$=C$+" "
580 GOSUB 380:REM STRIP EXCESS SPACES
590 REM ***********************
600 REM SCAN INPUT
610 J=0
620 J=J+1
630 IF LEN(C$)=0 AND DF=1 THEN RETURN
640 IF LEN(C$)=0 THEN 550
650 IF MID$(C$,J,1)<>" " THEN 620
660 E$=LEFT$(C$,J-1)
670 C$=MID$(C$,J+1)
680 REM *******************
690 REM ACTION
700 IF E$="ABORT" THEN 530
710 IF E$="CR" THEN PRINT:GOTO 610
720 IF E$<>"LLIST" THEN 740
730 FOR U=NC TO 1 STEP -1:LPRINT N$(U),M$(U):NEXT:GO
TO 610
740 IF E$="STACK" THEN SFLAG=-SFLAG+1:GOTO 610
750 REM *******************
760 REM NOW THE FORTH WORDS
770 KW=0
780 IF E$="@" THEN KW=-99:GOSUB 1290:REM DEFINE NEW
WORDS
790 IF E$="+" THEN KW=1
800 IF E$="-" AND MID$(C$,3,2)<>"IF" THEN KW=2
810 IF E$="*" THEN KW=3
820 IF E$="/" THEN KW=4
830 IF E$="MOD" THEN KW=5
840 IF E$="/MOD" THEN KW=6
850 IF E$="**" THEN KW=7
860 IF E$="SWAP" THEN KW=8
870 IF E$="*/" THEN KW=9
880 IF E$="*/MOD" THEN KW=10
890 IF E$="NEGATE" OR E$="MINUS" THEN KW=11
900 IF E$="ABS" THEN KW=12
```

```
910 IF E$="MAX" THEN KW=13
920 IF E$="MIN" THEN KW=14
930 IF E$="DUP" OR E$="?DUP" THEN KW=15
940 IF E$="OVER" THEN KW=16
950 IF E$="PICK" THEN KW=17
960 IF E$="DROP" THEN KW=18
970 IF E$="ROT" THEN KW=19
980 IF E$="ROLL" THEN KW=20
990 IF E$=".'" THEN KW=21
1000 IF E$="." THEN KW=22
1010 IF E$="EMIT" THEN KW=23
1020 IF E$="VLIST" THEN KW=24
1030 IF E$="FORGET" THEN KW=25
1040 IF LEFT$(E$,1)="'" THEN KW=26
1050 IF E$="KEY" THEN KW=27
1060 IF E$="RAND" THEN KW=28
1070 IF E$="DO" THEN KW=29
1080 IF E$="SPACES" THEN KW=30
1090 IF E$="1+" OR E$="1-" OR E$="2+" OR E$="2-" THE
N KW=32
1100 IF E$="2*" OR E$="2/" THEN KW=32
1110 IF E$="=" OR E$="<" OR E$=">" THEN KW=31
1120 IF E$="0=" OR E$="0<" OR E$="0>" THEN KW=31
1130 IF (E$="-" AND KW=0) OR E$="NOT" THEN KW=31
1140 IF KW>0 AND KW<9 THEN GOSUB 1580:GOTO 610
1150 IF KW<9 OR KW>16 THEN 1170
1160 ON KW-8 GOSUB 1890,1940,2010,2040,1580,1580,207
0,2120
1170 IF KW<17 OR KW>22 THEN 1190
1180 ON KW-16 GOSUB 2170,2240,2270,2170,2320,2420
1190 IF KW<21 OR KW>28 THEN 1210
1200 ON KW-22 GOSUB 2470,2500,2660,2830,2900,2940
1210 IF KW>28 THEN ON KW-28 GOSUB 2980,3150,3180,354
0
1220 REM NEXT LINES PUSH NUMBERS ON STACK
1230 CF=0:IF KW=0 AND LEN(E$)>0 THEN CF=1
1240 IF CF<>1 THEN 1270
1250 IF ASC(E$)>44 AND ASC(E$)<58 THEN E=VAL(E$):GOS
UB 300
1260 IF ASC(E$)>57 THEN GOSUB 1490:GOTO 570
1270 GOTO 610
1280 REM *****************
1290 REM DEFINE NEW WORDS
```

```
1300 REM USE @ TO GET TO THIS POINT
1310 REM NOTE WORD NAME MUST NOT BEGIN WITH A NUMBER

1320 IF NC<50 THEN NC=NC+1:REM ALLOWS UP TO 50 NEW
 WORDS
1330 X=0
1340 X=X+1
1350 IF MID$(C$,X,1)<>";" THEN 1370
1360 R$=LEFT$(C$,X-1):C$=MID$(C$,X+1):GOTO 1390
1370 IF X<LEN(C$) THEN 1340
1380 PRINT "MISSING ';'":C$="":RETURN
1390 REM NOW GET NAME OF NEW WORDS
1400 J=0
1410 J=J+1
1420 IF MID$(R$,J,1)=" " THEN 1450
1430 IF J<LEN(R$) THEN 1410
1440 PRINT "ERROR IN INPUT":C$="":RETURN
1450 N$(NC)=LEFT$(R$,J-1)
1460 M$(NC)=MID$(R$,J+1)
1470 RETURN
1480 REM ******************
1490 REM CHANGE DEFINED WORDS INTO DEFINITION
1500 J=NC+1
1510 J=J-1
1520 IF E$=N$(J) THEN 1550
1530 IF J>0 THEN 1510
1540 PRINT "WORD NOT DEFINED":KW=-99:RETURN
1550 C$=M$(J)+C$
1560 RETURN
1570 REM ******************
1580 REM TWO NUMBER OPERATIONS
1590 T1=0:T2=0:T3=0:T4=0:K$=""
1600 GOSUB 230:T1=E
1610 GOSUB 230:T2=E
1620 IF E$="MAX" OR E$="MIN" THEN 1840
1630 IF E$="+" THEN T3=T2+T1
1640 IF E$="-" THEN T3=T2-T1
1650 IF E$="*" THEN T3=T2*T1
1660 IF (E$<>"/" AND E$<>"/MOD" AND E$<>"MOD") THEN
1680
1670 IF T1=0 THEN PRINT "DIVISION BY 0":C$="":RETURN

1680 IF E$="/" OR E$="/MOD" THEN T3=INT(T2/T1)
1690 IF E$="MOD" THEN T3=T1*(T2/T1-INT(T2/T1))
```

```
1700 IF E$="**" THEN T3=T2^T1
1710 IF E$="/MOD" THEN 1750
1720 IF E$="SWAP" THEN 1800
1730 E=INT(T3+.5):GOSUB 300:RETURN
1740 REM ************
1750 REM /MOD
1760 T4=T1*(T2/T1-INT(T2/T1))
1770 E=INT(T4+.5):GOSUB 300
1780 E=INT(T3+.5):GOSUB 300:RETURN
1790 REM ************
1800 REM SWAP
1810 E=T1:GOSUB 300
1820 E=T2:GOSUB 300:RETURN
1830 REM ************
1840 REM MAX AND MIN
1850 A=T1:B=T2:IF T1>T2 THEN A=T2:B=T1
1860 E=A:IF E$="MAX" THEN E=B
1870 GOSUB 300:RETURN
1880 REM ************
1890 REM KW=9 */
1900 GOSUB 230:P3=E:GOSUB 230:P2=E:GOSUB 230:P1=E
1910 E=INT(P1*P2/P3):GOSUB 300
1920 RETURN
1930 REM ************
1940 REM KW=10 */MOD
1950 GOSUB 230:CQ=E:GOSUB 230:BQ=E:GOSUB 230:AQ=E
1960 DQ=AQ*BQ/CQ
1970 E=INT(CQ*((DQ)-INT(DQ))):GOSUB 300
1980 E=INT(DQ):GOSUB 300
1990 RETURN
2000 REM ************
2010 REM KW=11 NEGATE/MINUS
2020 GOSUB 230:E=-E:GOSUB 300:RETURN
2030 REM ************
2040 REM KW=12 ABS
2050 GOSUB 230:E=ABS(E):GOSUB 300:RETURN
2060 REM ************
2070 REM KW=15 DUP  ?DUP/-DUP
2080 GOSUB 230:IF E<>0 THEN 2100
2090 IF (E$="?DUP" OR E$="-DUP") THEN GOSUB 300:RETU
RN
2100 GOSUB 300:GOSUB 300:RETURN
2110 REM ************
```

```
2120 REM KW=16 OVER
2130 GOSUB 230:XW=E:GOSUB 230:YW=E
2140 E=YW:GOSUB 300:E=XW:GOSUB 300:E=YW:GOSUB 300
2150 RETURN
2160 REM ************
2170 REM KW=17 PICK/KW=20 ROLL
2180 GOSUB 230:FR=E-1
2190 FOR T=1 TO FR:GOSUB 230:C(T)=E:NEXT T
2200 GOSUB 230:XR=E:IF KW=17 THEN GOSUB 300
2210 FOR T=FR TO 1 STEP -1:E=C(T):GOSUB 300:NEXT T
2220 E=XR:GOSUB 300:RETURN
2230 REM ************
2240 REM KW=18 DROP
2250 GOSUB 230:RETURN
2260 REM ************
2270 REM KW=19 ROT
2280 GOSUB 230:G3=E:GOSUB 230:G2=E:GOSUB 230:G1=E
2290 E=G2:GOSUB 300:E=G3:GOSUB 300:E=G1:GOSUB 300
2300 RETURN
2310 REM ************
2320 REM KW=21 .'
2330 X=0
2340 X=X+1
2350 IF MID$(C$,X,1)="'" THEN 2380
2360 IF X<LEN(C$) THEN 2340
2370 PRINT "NO CLOSING ' IN INPUT":RETURN
2380 PRINT LEFT$(C$,X-1);
2390 C$=MID$(C$,X+1)
2400 RETURN
2410 REM ************
2420 REM KW=22 .
2430 GOSUB 230
2440 IF E<>1E-10 THEN PRINT E;" ";:RETURN
2450 PRINT "STACK EMPTY":PRINT:E=1E-10:GOSUB 300:RET
URN
2460 REM ************
2470 REM KW=23 EMIT
2480 GOSUB 230:PRINT CHR$(E);:RETURN
2490 REM ************
2500 REM KW=24 VLIST
2510 PRINT:IF NC=0 THEN 2540
2520 FOR J=NC TO 1 STEP -1:IF N$(J)<>"" THEN PRINT N
$(J),
2530 NEXT
2540 PRINT "+","-","*","/","MOD","/MOD","**","SWAP",

2550 PRINT "*/","*/MOD",
2560 PRINT "NEGATE","MINUS","ABS","MAX","MIN","OVER"
,
2570 PRINT "PICK","DROP",
2580 PRINT "ROT","ROLL",".'","'",".",":",";","EMIT",

2590 PRINT "VLIST","FORGET",
2600 PRINT "KEY","DO","LOOP","SPACES","IF","THEN",
2610 PRINT "DUP","?DUP","-DUP","=","<",">","0=","0<"
,"0>",
2620 PRINT "NOT","1+","1-","2+","2-","2*","2/"
2630 PRINT "STACK","ABORT","LLIST","RAND","CR"
2640 PRINT:RETURN
2650 REM *********************
2660 REM KW=25 FORGET
2670 J=0
2680 J=J+1
2690 IF J>LEN(C$) THEN PRINT "FORGET":KW=-99:RETURN
2700 IF MID$(C$,J,1)=" " THEN 2720
2710 GOTO 2680
2720 F$=LEFT$(C$,J-1):C$=MID$(C$,J+1):FLAG=0
2730 FOR J=NC+1 TO 1 STEP -1
2740 IF N$(J)=F$ THEN N$(J)="":M$(J)="":FLAG=1:DROP=
J
2750 NEXT J
2760 IF FLAG=0 THEN PRINT F$;" NOT RECOGNISED"
2770 IF FLAG=0 THEN C$=MID$(C$,LEN(F$)+1):KW=-99:RET
URN
2780 ZN=NC:FOR J=NC TO DROP STEP-1
2790 N$(J)="":M$(J)="":ZN=ZN-1
2800 NEXT J:NC=ZN
2810 RETURN
2820 REM **********************
2830 REM KW=26 ' (TICK)
2840 PRINT:E$=MID$(E$,2):FL=0
2850 FOR J=1 TO NC:IF N$(J)=E$ THEN FL=J
2860 NEXT J:IF FL=0 THEN PRINT "NOT KNOWN":RETURN
2870 PRINT "> ";E$;" < OK":PRINT:PRINT TAB(2);M$(FL)

2880 RETURN
2890 REM **********************
2900 REM KW=27 KEY
2910 W$=INKEY$:IF W$="" THEN 2910
```

```
2920 E=ASC(W$):GOSUB 300:RETURN
2930 REM **********************
2940 REM KW=28 RAND
2950 GOSUB 230:Y1=E:GOSUB 230:Y2=E
2960 E=INT(RND(0)*Y2+Y1):GOSUB 300:RETURN)
2970 REM **********************
2980 REM KW=29  DO LOOPS
2990 REM FIND THE END OF THE LOOP
3000 JH=0
3010 JH=JH+1
3020 IF MID$(C$,JH,1)<>0 THEN 3040
3030 IF MID$(C$,JH+1,4)="LOOP" THEN 3060
3040 IF JH<LEN(C$) THEN 3010
3050 PRINT TAB(25);"DO LOOP ERROR":RETURN
3060 DF=1
3070 O$=LEFT$(C$,JH):I$=MID$(C$,JH+6)
3080 GOSUB 230:L2=E:GOSUB 230:L1=E
3090 ST=1:IF L2>=L1 THEN ST=-1:L1=L1+2
3100 FOR O=L2 TO L1-1 STEP ST:REM LETTER O, NOT ZERO

3110 C$=O$:GOSUB 570
3120 NEXT O:REM LETTER O, NOT ZERO
3130 C$=I$:DF=0:RETURN
3140 REM **********************
3150 REM KW=30 SPACES
3160 GOSUB 230:FOR J=1 TO E:PRINT " ";:NEXT J:RETURN

3170 REM **********************
3180 REM KW=31 IF...THEN
3190 IF LEFT$(C$,2)="IF" THEN C$=MID$(C$,4)
3200 IF LEFT$(C$,6)="NOT IF" THEN C$=MID$(C$,8):GOSU
B 3440
3210 REM NOW LOOK FOR THE THEN
3220 BJ=0
3230 BJ=BJ+1
3240 IF MID$(C$,BJ,4)="THEN" THEN 3270
3250 IF BJ<LEN(C$) THEN 3230
3260 PRINT TAB(25);"NO THEN TO MATCH IF":RETURN
3270 U$=LEFT$(C$,BJ-1):C$=MID$(C$,BJ+5)
3280 REM NOW CHECK CONDITION
3290 GOSUB 230:K2=E
3300 IF LEFT$(E$,1)<>"O" THEN GOSUB 230:K1=E
3310 TRUE=0
3320 IF E$="=" AND K2=K1 THEN TRUE=1
3330 IF (E$="<>" OR E$="-") AND K2<>K1 THEN TRUE=1
```

```
3340 IF E$="<" AND K1<K2 THEN TRUE=1
3350 IF E$=">" AND K1>K2 THEN TRUE=1
3360 IF E$="0=" AND K2=0 THEN TRUE=1
3370 IF E$="0<>" AND K2<>0 THEN TRUE=1
3380 IF E$="0<" AND K2<0 THEN TRUE=1
3390 IF E$="0>" AND K2>0 THEN TRUE=1
3400 IF TRUE=0 THEN RETURN
3410 C$=U$+C$
3420 RETURN
3430 REM *********************
3440 REM NOT - REVERSE CONDITION
3450 IF E$="=" THEN E$="<>":RETURN
3460 IF E$="-" THEN E$="=":RETURN
3470 IF E$="<" THEN E$=">":RETURN
3480 IF E$=">" THEN E$="<":RETURN
3490 IF E$="0=" THEN E$="0<>":RETURN
3500 IF E$="0<" THEN E$="0>":RETURN
3510 IF E$="0>" THEN E$="0<":RETURN
3520 PRINT "NOT NEEDS CONDITION":RETURN
3530 REM *********************
3540 REM 1+  1-  2+  2-  2*  2/
3550 GOSUB 230
3560 IF E$="1+" THEN E=E+1
3570 IF E$="1-" THEN E=E-1
3580 IF E$="2+" THEN E=E+2
3590 IF E$="2-" THEN E=E-2
3600 IF E$="2*" THEN E=E*2
3610 IF E$="2/" THEN E=INT(E/2)
3620 GOSUB 300:RETURN
```

This is what you'll see:

```
PRINT OUT STACK (Y/N)?
```

As you can see the program asks you if you want the stack printed out. Press the "Y" key the first time you run this. When you do, you'll see the top four positions on the stack printed out:

```
                                        STACK:
                                            1 :   --
                                            2 :   --
                                            3 :   --
                                            4 :   --
                                      ------------------

                                     >> OK
```

The double-hyphen (--) indicates that the stack is empty at that point (or the sheet of paper is blank). You can prove this by entering a dot (.) after the ">> OK." The ? means it is waiting for your input. Type in, making sure you leave spaces between each number:

333 861 123

Once you press your RETURN key, the process will begin. When you have the visible stack turned on (and you can turn it off or on by entering the word STACK) you can see the line being processed as follows. First of all the 333 will be taken from the input, and placed on the top of the stack:

```
 >> OK
? 333 861 123
    861 123
                    333

    STACK:
           1 :   333
           2 :   --
           3 :   --
           4 :   --
  ----------------
```

You can see the element of the input which is currently being processed printed just above the stack to the right (you can see the 333 there at the moment, whereas the material left in the input still to be processed — 861 123 — is shown slightly above this and to the left). With a usual FORTH compiler, of course, you would not see these things, but we included them in order to make it easy to understand what is going on (and, of course, you can turn them off at any stage if you like, simply by typing in the word STACK as part of your input.

Now follow through the process as the 861 is placed on top of the stack, causing the 333 to be pushed down to the second position. On top of this goes the 123.

```
  123
                    861                                         123

  STACK:                                STACK:
        1 :   861                             1 :   123
        2 :   333                             2 :   861
        3 :   --                              3 :   333
        4 :   --                              4 :   --
-----------------                     -----------------

                                      >> OK
```

Now, we use the 'dot' (.) to pop (as we say in the world of FORTH) the top number off the stack, and you can see the number which has been popped below the dotted line to the left:

```
                                               .
                       .                                         .

   STACK:                                  STACK:
         1 :   861                               1 :   333
         2 :   333                               2 :   --
         3 :   --                                3 :   --
         4 :   --                                4 :   --
 -----------------                       -----------------

  123                                    861


                       .

   STACK:
         1 :   --
         2 :   --
         3 :   --
         4 :   --
 -----------------

  333     >> OK
?
```

We use more dots to pop the next two numbers off the stack. What would happen if we tried to pop off some more numbers, numbers which are not there?

```
                                            .

                              STACK:
                                   1  :   --
                                   2  :   --
                                   3  :   --
                                   4  :   --
                           -----------------

                         STACK  EMPTY
```

As you can see, you get the message *Stack empty*.

Using the Mathematical Capabilities

We will now try out a simple addition sum, and see how it occurs. The sum, of course, must be written in Reverse Polish Notation. The sum we'll use is:

12 76 + .

With this, we are telling the computer to push the numbers 12 and 76 onto the stack, then add the two of them together (+), then pop (using .) the result off the stack.

```
  >> OK
?  12 76 + .
     76 + .                           + .
                  12                                        76

     STACK:                         STACK:
          1 :  12                        1 :  76
          2 :  --                        2 :  12
          3 :  --                        3 :  --
          4 :  --                        4 :  --
  -----------------              -----------------
```

Once the numbers are on the stack, as the computer works its way through the input stream, it comes to the "+" and it knows it needs two numbers to add together, so it pops the top two numbers from the stack, adds them, then pushes the result of that operation (88, in this case) back onto the stack. Finally, the computer comes to the "." which tells it to pop the top value off the stack and print it out.

```
    .                                       .
                      +                                         +

    STACK:                                  STACK:
          1  :   12                               1  :   --
          2  :   --                               2  :   --
          3  :   --                               3  :   --
          4  :   --                               4  :   --
  ----------------                        ----------------


    .
                      +                                         .

    STACK:                                  STACK:
          1  :   88                               1  :   --
          2  :   --                               2  :   --
          3  :   --                               3  :   --
          4  :   --                               4  :   --
  ----------------                        ----------------

                                            88    >> OK
                                           ?
```

Here is an addition which includes a positive number (34) and a negative one (-14):

```
? 34  -14  +  .
    -14  +  .                               +  .
                    34                                         -14

    STACK:                                  STACK:
          1  :   34                               1  :  -14
          2  :   --                               2  :   34
          3  :   --                               3  :   --
          4  :   --                               4  :   --
  ----------------                        ----------------
```

```
              +                                          .

   STACK:                                  STACK:
       1  :   20                               1  :   --
       2  :   --                               2  :   --
       3  :   --                               3  :   --
       4  :   --                               4  :   --
 -----------------                       -----------------

                                         20    >> OK
                                        ?
```

FORTH arithmetic is very flexible, as we'll now see. Imagine you have to add five numbers together. You could do this by entering the numbers, and the required operators, in a sequence like the following:

65 12 + 67 + 15 + 11 + .

Try it to see what happens. At the end, you'll see this:

```
                  .

      STACK:
          1  :   --
          2  :   --
          3  :   --
          4  :   --
    -----------------

   170    >> OK
  ?
```

All the plus signs can be placed at the end of the string of numbers like this:

65 12 67 15 11 + + + + .

Follow this through, on-screen, and you'll see it produces exactly the same result.

Small Constants

FORTH is provided with built-in 'words' to add either 1 or 2, to the number on the top of the stack. Here is 1+ in action:

```
  >> OK
? 99 1+ .
    1+ .                                    .
                    99                                      1+

    STACK:                                STACK:
         1 :   99                              1 :   --
         2 :   --                              2 :   --
         3 :   --                              3 :   --
         4 :   --                              4 :   --
  -----------------                     -----------------


    .
                    1+                                      .

    STACK:                                STACK:
         1 :   100                            1 :   --
         2 :   --                             2 :   --
         3 :   --                             3 :   --
         4 :   --                             4 :   --
  -----------------                     -----------------

                                          100    >> OK
                                         ?
```

As was pointed out earlier, the visible stack can be turned off simply by typing in the word STACK, either by itself, or as part of an input stream. For this next section of the tutorial, I've turned the stack off, but you can leave it on if you like, to make it easier to understand what is going on. Here is 2+ in action:

```
? STACK
 >> OK
? 98 2+ .
 100    >> OK
```

To subtract numbers from each other, you — fairly obviously — replace the + sign with a - one.

```
? 34 12 - .
 22    >> OK
? 88 -20 - .
 108    >> OK
? -9 -3 - .
-6    >> OK
```

The * sign is used for multiplication:

```
? 4 7 * .
 28    >> OK
```

While the / sign is used for division:

```
? 10 3 / .
 3    >> OK
```

FORTH works in integers, so the / word leaves only the quotient on the stack (3 in this case). To get the remainder, you use the FORTH word MOD:

```
? 10 3 MOD .
 1    >> OK
? -10 -3 MOD .
-1    >> OK
?
```

To get the quotient and the remainder left on the stack, with the quotient on top, you use the combined operation word /MOD, followed by two dots to pop off both numbers. In this case, 10 3 /MOD . . would produce an answer of 3 1 as you should discover when you try it.

More Combined Operations

There are also FORTH words involved in the process of combined multiplication and division operations. In our next example, we multiply 24 and 6 together, then divide the result of that by 7:

```
? 24 6 * 7 / .
 20    >> OK
```

We can do it in one operation with */ which, you'll note, does *not* have a space between the * and the /.

```
? 24 6 7 */ .
 20    >> OK
```

Three operations are combined in the operations triggered by the word */MOD, which leaves both the quotient and the remainder on the stack (with the quotient on top):

```
? 24 6 7 */MOD . .
 20   3    >> OK
```

As FORTH works in integers, it is necessary to approximate floating point numbers when they are provided. A good approximation to PI, for example, is provided by the fraction 355/113 (which would be expressed in FORTH as 355 113 /). We can use this knowledge to work out the circumference of a circle, give the diameter, as this example, with a diameter of 56, demonstrates:

```
? 56 355 113 */ .
 175   >> OK
```

All FORTH operations, in the end, more or less come down to pushing values on, and popping them from, the stack.

The Vocabulary Widens

FORTH is provided with many, many words to manipulate the values on the stack, in addition to those which allow you to use it as a (rather clumsy) calculator.

The word ** is used to raise numbers to a power:

```
? 7 3 ** .
 343   >> OK
```

NEGATE multiplies a number by minus one:

```
? 41 NEGATE .
-41    >> OK
```

There are two 'standard' versions of FORTH, fig-FORTH (from the Forth Interest Group) and FORTH-83. The *Tutor* adheres to the FORTH-83 standard, but allows the use of the two fig-FORTH words. The first of these is MINUS which has exactly the same effect as NEGATE:

```
? 41 MINUS .
-41    >> OK
```

ABS returns the absolute value of a number:

```
? 125 ABS .
 125   >> OK
? -125 ABS .
 125   >> OK
```

The words MAX and MIN return the maximum and minimum of the top two numbers on the stack respectively, leaving just the selected number on the stack:

```
? 12 -9 MAX .
 12    >> OK
? 12 -9 MIN .
-9    >> OK
? -12 -9 MAX .
-9    >> OK
? -12 -9 MIN .
-12    >> OK
```

Seeing Double

The FORTH word DUP duplicates the top number on the stack, and places the copy above the original number, as you can see here:

```
? 12

                  12                                  DUP

    STACK:                         STACK:
         1  :   12                      1  :   --
         2  :   --                      2  :   --
         3  :   --                      3  :   --
         4  :   --                      4  :   --
  -----------------              -----------------

 >> OK
? DUP


                  DUP                                   DUP

    STACK:                           STACK:
         1  :   12                        1  :   12
         2  :   --                        2  :   12
         3  :   --                        3  :   --
         4  :   --                        4  :   --
  -----------------                -----------------

                                 >> OK
```

DUP is very useful when you need to double a number (DUP +), square it (DUP *), cube it (DUP DUP * *) or even quadruple it (DUP * DUP *). Let's see the stack in action doubling a number:

```
+ .
          DUP

STACK:
    1 :  17
    2 :  12
    3 :  12
    4 :  --
-----------------

.
          +

STACK:
    1 :  17
    2 :  12
    3 :  12
    4 :  --
-----------------

.
          +

STACK:
    1 :  34
    2 :  12
    3 :  12
    4 :  --
-----------------
```

```
+ .
          DUP

STACK:
    1 :  17
    2 :  17
    3 :  12
    4 :  12
-----------------

.
          +

STACK:
    1 :  12
    2 :  12
    3 :  --
    4 :  --
-----------------

          .

STACK:
    1 :  12
    2 :  12
    3 :  --
    4 :  --
-----------------

34    >> OK
```

Here are some additional DUP operations, this time with the stack turned off:

```
  >> OK
? 12 DUP * .                  ? 12 DUP DUP * * .
 144    >> OK                  1728    >> OK
```

Flipped Over You

The word OVER is used when you want to duplicate the second value on the stack, rather than the first one:

```
? 12 33 OVER
    33 OVER
                12

    STACK:
          1  :   12
          2  :   --
          3  :   --
          4  :   --
  -----------------
```

```
                OVER

    STACK:
          1  :   12
          2  :   --
          3  :   --
          4  :   --
  -----------------
```

```
                OVER

    STACK:
          1  :   12
          2  :   --
          3  :   --
          4  :   --
  -----------------
```

```
                OVER

    STACK:
          1  :   33
          2  :   12
          3  :   --
          4  :   --
  -----------------
```

```
  OVER
                33

  STACK:
        1  :   33
        2  :   12
        3  :   --
        4  :   --
-----------------
```

```
                OVER

  STACK:
        1  :   --
        2  :   --
        3  :   --
        4  :   --
-----------------
```

```
                OVER

  STACK:
        1  :   33
        2  :   12
        3  :   --
        4  :   --
-----------------
```

```
  STACK:
        1  :   12
        2  :   33
        3  :   12
        4  :   --
-----------------

>> OK
```

To select any number on the stack, you use the word PICK, which is preceded by a number. This number (3, in the example which follows) tells the computer to select the 3rd number on the stack, and duplicate it on top of the stack:

```
? 3 PICK
  PICK
                  3                                   PICK

  STACK:                          STACK:
        1  :   3                        1  :   3
        2  :   3                        2  :   72
        3  :   72                       3  :   23
        4  :   23                       4  :   8
------------------               ------------------

                  PICK                                PICK

  STACK:                          STACK:
        1  :   72                       1  :   23
        2  :   23                       2  :   8
        3  :   8                        3  :   1
        4  :   1                        4  :   12
------------------               ------------------

                  PICK                                PICK

  STACK:                          STACK:
        1  :   8                        1  :   23
        2  :   1                        2  :   8
        3  :   12                       3  :   1
        4  :   33                       4  :   12
------------------               ------------------
                                                      PICK
                  PICK

  STACK:                          STACK:
        1  :   72                       1  :   3
        2  :   23                       2  :   72
        3  :   8                        3  :   23
        4  :   1                        4  :   8
------------------               ------------------
```

```
                    PICK                                    PICK

     STACK:                                  STACK:
          1  :   3                                1  :   23
          2  :   72                               2  :   3
          3  :   23                               3  :   72
          4  :   8                                4  :   23
   -----------------                       -----------------

                                         >>  OK
```

DROP simply gets the computer to drop the first number on the stack (but not, unlike the dot, printing it out):

```
          ?  DROP

                              DROP

               STACK:
                    1  :   3
                    2  :   72
                    3  :   23
                    4  :   8
             -----------------

            >>  OK
```

SWAP causes the top two numbers on the stack to change places:

```
 ?  SWAP

                    SWAP                                    SWAP

     STACK:                                  STACK:
          1  :   72                               1  :   23
          2  :   23                               2  :   8
          3  :   8                                3  :   1
          4  :   1                                4  :   12
   -----------------                       -----------------
```

```
                   SWAP                                   SWAP

     STACK:                             STACK:
           1  :  3                            1  :  72
           2  :  23                           2  :  3
           3  :  8                            3  :  23
           4  :  1                            4  :  8
   -----------------                  -----------------

                                     >> OK
```

The FORTH word ROT rotates the third number on the stack up to the top (but note that it does not leave a copy of the number in its original position):

```
 ? ROT

                   ROT                                   ROT

     STACK:                             STACK:
           1  :  3                            1  :  23
           2  :  23                           2  :  8
           3  :  8                            3  :  1
           4  :  1                            4  :  12
   -----------------                  -----------------

                   ROT                                   ROT

     STACK:                             STACK:
           1  :  8                            1  :  3
           2  :  1                            2  :  8
           3  :  12                           3  :  1
           4  :  33                           4  :  12
   -----------------                  -----------------

                   ROT                                   ROT

     STACK:                             STACK:
           1  :  72                           1  :  23
           2  :  3                            2  :  72
           3  :  8                            3  :  3
           4  :  1                            4  :  8
   -----------------                  -----------------
```

ROLL works somewhat like PICK, allowing you to move any number of your choice to the top of the stack:

```
 ? 4 ROLL
   ROLL
                  4                                   ROLL

   STACK:                              STACK:
        1 :  4                              1 :  23
        2 :  23                             2 :  72
        3 :  72                             3 :  3
        4 :  3                              4 :  8
 ----------------                    ----------------

                  ROLL                                ROLL

   STACK:                              STACK:
        1 :  72                             1 :  3
        2 :  3                              2 :  8
        3 :  8                              3 :  1
        4 :  1                              4 :  12
 ----------------                    ----------------

                  ROLL                                ROLL

   STACK:                              STACK:
        1 :  8                              1 :  1
        2 :  1                              2 :  12
        3 :  12                             3 :  33
        4 :  33                             4 :  12
 ----------------                    ----------------

                  ROLL                                ROLL

   STACK:                              STACK:
        1 :  72                             1 :  23
        2 :  3                              2 :  72
        3 :  1                              3 :  3
        4 :  12                             4 :  1
 ----------------                    ----------------
```

```
                    ROLL                                  ROLL

     STACK:                              STACK:
           1 :   23                            1 :   8
           2 :   72                            2 :   23
           3 :   3                             3 :   72
           4 :   1                             4 :   3
   -----------------                   -----------------

                                       >> OK
                                     ?
```

Defining Your Own Words

The most exciting thing about FORTH is the ease with which new words can be defined, and we'll look at that next.

You'll recall, from the arithmetic examples you've tried, that the dot (.) is used to pop the top value off the stack and print it out. There may be times when you want to print out the value, but you still want the value on the stack afterwards. Dot, as you have seen, is destructive. That is, you *lose* the top number by popping it off.

As we've seen, the FORTH word DUP (for 'duplicate') gets around this, by duplicating the top number on the stack, and pushing a copy of it on top of the first one (so positions one and two on the stack now have the same value). Then, you can pop the number off to print it out, but leave behind a copy of it, still on top of the stack.

We can create a word of our own, PRINTOUT, to duplicate the top number on the stack, then pop off the top value and print it out. You create words with an input as follows:

```
? @ PRINTOUT DUP . ;
```

You start defining a word with an @ symbol (although real FORTH uses :) and follow it with a space (you'll recall that spaces are vital in all aspects of FORTH) and then put the words (in this case DUP and .) that you want in your definition, followed by a semi-colon to signal that the word has been defined.

Then, from that point onwards, you could just use PRINTOUT in your FORTH programs, and your computer will automatically perform the DUP . for you. The *Tutor* program allows you to define up to 50 new words. This should be more than enough for your needs.

Here is our PRINTOUT word in action:

```
? 34 12 + PRINTOUT
   12 + PRINTOUT                  + PRINTOUT
               34                             12

   STACK:                         STACK:
       1 :  34                        1 :  12
       2 :  1                         2 :  34
       3 :  12                        3 :  1
       4 :  33                        4 :  12
 -----------------              -----------------

   PRINTOUT                       PRINTOUT
               +                              +

   STACK:                         STACK:
       1 :  34                        1 :  1
       2 :  1                         2 :  12
       3 :  12                        3 :  33
       4 :  33                        4 :  12
 -----------------              -----------------

   PRINTOUT                      .
               +                             DUP

   STACK:                        STACK:
       1 :  46                       1 :  1
       2 :  1                        2 :  12
       3 :  12                       3 :  33
       4 :  33                       4 :  12
 -----------------              -----------------

   .                             .
               DUP                           DUP

   STACK:                        STACK:
       1 :  46                       1 :  46
       2 :  1                        2 :  46
       3 :  12                       3 :  1
       4 :  33                       4 :  12
 -----------------              -----------------
```

```
      STACK:
             1  :   46
             2  :   1
             3  :   12
             4  :   33
   -----------------

  46     >> OK
 ?
```

If you wanted to define a word which would square the number on top of the stack, make a copy of the result, then print out the answer, you could define the word as follows:

```
                ? @ SQUARE DUP * DUP . ;
                 >> OK
```

In this word, SQUARE is the name of the word you are defining. DUP duplicates the top number on the stack, then uses * to multiply it by itself. The answer is then on top of the stack, where DUP . copies the answer, then prints it out, leaving a copy of the answer on the stack. Note you can call a word in our FORTH anything you like, so long as it begins with a letter and not with an operator (+, −, * or /) or a number.

Now this brings us to an even more exciting part of FORTH. You can see that the last part of our definition of SQUARE (DUP .) is the same as the definition of PRINTOUT. Why don't we use PRINTOUT in our SQUARE definition? You can do it simply by adding this word to the dictionary:

```
              ? @ SQUARE DUP * PRINTOUT ;
               >> OK
```

Let's see it in action:

```
 ? 12 SQUARE
     SQUARE                        * PRINTOUT
                   12                               DUP

     STACK:                        STACK:
            1  :   12                     1  :   46
            2  :   46                     2  :   1
            3  :   1                      3  :   12
            4  :   12                     4  :   33
   -----------------             -----------------
```

```
* PRINTOUT
              DUP

STACK:
    1 :   12
    2 :   46
    3 :   1
    4 :   12
------------------
```

```
* PRINTOUT
              DUP

STACK:
    1 :   12
    2 :   12
    3 :   46
    4 :   1
------------------
```

```
PRINTOUT
              *

STACK:
    1 :   12
    2 :   46
    3 :   1
    4 :   12
------------------
```

```
PRINTOUT
              *

STACK:
    1 :   46
    2 :   1
    3 :   12
    4 :   33
------------------
```

```
PRINTOUT
              *

STACK:
    1 :   144
    2 :   46
    3 :   1
    4 :   12
------------------
```

```
.
              DUP

STACK:
    1 :   144
    2 :   144
    3 :   46
    4 :   1
------------------
```

```
              .

STACK:
    1 :   144
    2 :   46
    3 :   1
    4 :   12
------------------

 144    >> OK
?
```

You can see that when the *Tutor* comes to process a defined word, the program turns the defined word back into its definition.

Here's another run of SQUARE, with the stack printing turned off.

```
? STACK
 >> OK
? 9 SQUARE
 81    >> OK
?
```

FORTH always uses the *most-recently defined* version of a word, searching its dictionary from the newest word, to the oldest, so we can safely simply redefine a word, and know it will use the newest version of that word.

You can see from this inclusion of PRINTOUT within SQUARE a hint of the real magic of FORTH. You can keep adding more and more words to your dictionary, which is made up of the words the language comes with, plus the words you add.

Getting More Involved

You can use the word SQUARE in the definition of further words. CUBE, and N**6 (to raise a number to the sixth power) can be built on your earlier work:

```
 >> OK
? @ PRINTOUT DUP . ;
 >> OK
? @ SQUARE DUP * PRINTOUT ;
 >> OK
? @ CUBE DUP SQUARE * PRINTOUT ;
 >> OK
? 3 CUBE
 9   27    >> OK
? @ N**6 SQUARE CUBE PRINTOUT ;
 >> OK
? 3 N**6
 9   81    729    729    >> OK
?
```

And Now For A Vlist

FORTH comes with the command VLIST which puts the whole current vocabulary on the screen, starting with the most recently defined word. If you did a VLIST right now (entering the word as a standard input after >> OK) you'd see this:

```
 >> OK
? VLIST

N**6          CUBE
SQUARE        PRINTOUT
+             -
*             /
MOD           /MOD
**            SWAP
*/            */MOD
NEGATE        MINUS
ABS           MAX
MIN           OVER
PICK          DROP
ROT           ROLL
."            '
.             :
;             EMIT
VLIST         FORGET
KEY           DO
LOOP          SPACES
IF            THEN
DUP           ?DUP
-DUP          =
<             >
0=            0<
0>            NOT
1+            1-
2+            2-
2*            2/
STACK         ABORT
LLIST         RAND
CR

 >> OK
?
```

From the VLIST you can the final five words are STACK, ABORT, LLIST, RAND and CR. These are not standard FORTH words, but have been included with the *Tutor* as they make developing your own programs much simpler than might otherwise be the case. FORTH differentiates between a Carriage Return, which forces a line-feed, and use of the RETURN key. To emulate this, we have the word CR, to stand for Carriage Return. It can be used within a DO LOOP (which we'll be looking at shortly) to imitate the clear screen command, CLS, in VZ300 BASIC:

```
? @ CLS 25 1 DO CR LOOP ;
```

When you enter CLS the screen will clear by scrolling upwards. There may well be other BASIC programming words you wish to emulate with the *Tutor.*

The non-standard vocabulary also includes ABORT to cancel action (usually in conjunction with an IF/THEN); LLIST to get a hard copy printout of all the words you've defined, and their meanings; and STACK (as mentioned earlier) to turn the printing of the visible stack off and on.

You can include text printout in FORTH programs with the word *dot quote* (.'), followed by a space, and then the text. All text material up to the close quote (') will then be printed out, much as the PRINT command works in BASIC. We can use this knowledge to produce a more user-friendly definition of SQUARE:

```
 >> OK
? @ SQUARE DUP .' THE SQUARE OF' . DUP * .' IS' PRINTOUT ;
 >> OK
```

```
? 5 SQUARE
THE SQUARE OF 5  IS 25   >> OK
?
```

Another VLIST would show that the second SQUARE is now first on the list. FORTH will always execute the most-recently defined version of a word, so our new SQUARE will be defined so long as it is left in the dictionary (note that some FORTH systems use a double quote — as in ." — where the *Tutor* uses .').

Just in case you forget what meaning you've assigned to a word, you can use the FORTH word *tick* ('). Generally in FORTH this just replies name__of__word OK if the word has been defined, but with the *Tutor* it not only returns the word OK, but also tells you the meaning you have given to a word:

```
? 'SQUARE

          > SQUARE < OK

            DUP .' THE SQUARE OF' . DUP * .' IS' PRINTOUT
 >> OK
? 'PRINTOUT

          > PRINTOUT < OK

            DUP .
 >> OK
?
```

To get a word *out* of your dictionary, use the FORTH word FORGET. Note, however, that when you FORGET a word, all the words defined prior to that definition will also be forgotten. Look what happens when we tell our system to FORGET SQUARE:

```
 >> OK
? FORGET SQUARE
 >> OK
? VLIST

PRINTOUT      +
-             *
/             MOD
/MOD          **
SWAP          */
*/MOD         NEGATE
MINUS         ABS
MAX           MIN
OVER          PICK
DROP          ROT
ROLL          .'
'             .
:             ;
EMIT          VLIST
FORGET        KEY
DO            LOOP
SPACES        IF
```

```
-DUP          =
<             >
0=            0<
0>            NOT
1+            1-
2+            2-
2*            2/
STACK         ABORT
LLIST         RAND
CR

 >> OK
?
```

Getting The Key

KEY works like INKEY$, waiting till you touch a key, then putting the ASCII code of the character on top of the stack. Here it is in use in the definition of NUMBER_KEY which allows you to touch a number key, and this then puts this number on the stack:

```
 >> OK
? @ NUMBER-KEY KEY 48 - PRINTOUT ;
 >> OK
? NUMBER-KEY
 6    >> OK
?
```

If you wanted to enter a two-digit number, such as 67, you could use our word TWO-NUMBERS:

```
? @ TWO-NUMBERS NUMBER-KEY 10 * NUMBER-KEY + PRINTOUT ;
 >> OK
? TWO-NUMBERS
 6    7    67    >> OK
?
```

EMIT is the 'opposite' of KEY, in that it prints out the character whose ASCII code precedes it, so 42 EMIT would print out an asterisk.

```
? 42 EMIT
* >> OK
```

SPACES can be used to format output, as it takes the top number off the stack, and prints out that number of spaces. You can emulate the BASIC command TAB using SPACES:

```
? @ TAB SPACES ;
 >> OK
? 12 TAB .' TEST'
             TEST >> OK
? 20 TAB .' TEST'
                     TEST >> OK
? 5 TAB .' TEST'
      TEST >> OK
```

The usefulness of being able to define words is illustrated by the following example. Suppose you want a word called CIRCLE, which would take a number off the stack and treat it as the diameter of the circle, printing out the circumference of the circle; you'd only need to enter something like 23 CIRCLE to get your computer to print out the circumference of a circle with a diamter of 23 units.

The word CIRCLE could be defined as follows (where, as before, 355/113 is used as an integer approximation to PI, keeping in mind that most FORTHs work in integers):

```
@ CIRCLE 355 113 */ . .' IS THE CIRCUMFERENCE' ;
```

If you did this, you'd only have to enter a number (or allow the computer to take one off the stack) for the diameter, and the computer would do the rest. You'll find you should have a lot of fun defining words and using them with this program on your VZ300.

DO Loops

A DO LOOP is somewhat like a FOR/NEXT loop in VZ300 BASIC. It repeats everything which comes between the words DO and LOOP. The number of repetitions is the difference between the top two numbers on the stack. If the second number on the stack was 11 and the top number was 1, the loop would be run through ten times.

You're sure to be familiar with IF and THEN from VZ BASIC and although they perform much the same function in FORTH, the order in which they are placed in a statement may seem strange.

The *condition* being tested comes before the word IF, and if the condition is evaluated as true, all the words between IF and THEN are carried out. If it is evaluated as false, action moves along the input stream beyond the THEN. IF/THEN in FORTH can test =, < >, <, > and can check as well to see if the number on top of the stack is equal to, not equal to, less than or greater than zero. The word NOT causes IF/*THEN* to look for the opposite condition so NOT = is the same in FORTH as the BASIC < >.

```
? 7 1 DO 8 . LOOP
 8    8    8    8    8    8    >> OK
?
```

You'll find the *Tutor* program is well furnished with error messages, and these (along with the visible stack) should make it pretty easy to keep aware of what is going on in the program.

In Summary

You'll recall we started by discussing the principle of the stack, and then looked at the arithmetic operators (+ - * and /) which take the top two numbers off the stack, and then add, subtract the first one from the second, multiply them together or divide the second number by the top one. In all cases, the result is pushed onto the top of the stack.

MOD does a division like /, but where / puts the quotient on the stack, MOD puts the remainder onto it. /MOD does a division again, but puts the quotient and remainder on the stack, with the quotient on top. */ does a 'multiply then divide' leaving the quotient. It needs three numbers from the stack. */MOD performs like */, returning the quotient and remainder, with the quotient on top.

** takes two numbers from the stack, and raises the second number to the power of the first one on the stack, returning the result to the stack.

ABS performs just like ABS in BASIC, returning the absolute value of the number to the top of the stack.

MAX and MIN compare the top two numbers on the stack, leaving only the largest (MAX) or smallest (MIN) there. DUP we've met before, and it duplicates the top value on the stack. ?DUP works like DUP, but only if the value is not zero (the *fig-FORTH* equivalent is —DUP so this is also recognised). OVER takes the second number on the stack, copies it, and puts a copy of that on top of the stack. This means if the top number was A and the second one was B, an OVER would make the stack read (from top to bottom) B A B.

PICK must be preceded by a number. This command selects the numbered element on the stack (so if the number which preceded it was 6, PICK would select the sixth element, counting down), then copies it onto the top of the stack. DROP deletes the top number on the stack, and SWAP causes the two numbers on the stack to change places. ROT (for ROTate) brings the third element on the stack up to the top, moving the former items one and two down a position each.

ROLL is like PICK, in that it must be preceded by a number. It brings this numbered element to the top, deleting it from its original position, and moving all other elements down one position. VLIST lists out every word the program understands, with user-defined words first, from the newest one to the oldest.

FORGET must be followed by the name of the word you have defined. Remember, not only does this cause the program to delete the word and its definition from the dictionary, but all words defined *after* that one are also deleted. FORGET must therefore be used with caution.

KEY works like INKEY$, waiting till you touch a key, then putting the ASCII code of the character on top of the stack. The command 'dot' (.) pops the top element off the stack and prints it out, while 'dot-quote' (.') is used to preface text output. It must be followed by a space. FORTH regards as text all material which follows dot-quote up to the next quote (') which can follow the text *without* a space preceding it (many FORTHs use ." rather than .').

FORTH usually includes the word called 'tick', which is a single quote mark (') which is followed by the name of a word. If that word is in the dictionary, the computer will print out *name OK* to show the word does exist. As well, in my FORTH, tick prints out the complete definition of the word, so if you've forgotten what meaning you assigned to a word '*name* will print it out for you on the screen.

EMIT is the 'opposite' of KEY, in that it prints out the character whose ASCII code precedes it, so 42 EMIT would print out an asterisk.

SPACES can be used to format output, as it takes the top number off the stack, and prints out that number of spaces. FORTH allows for a 'carriage return' (which is *not* the same as pressing the RETURN key) and we have imitated this by allowing you to include the non-FORTH word CR which simply moves print output to the start of the next line. This is also helpful in formatting output.

A DO LOOP is somewhat like a FOR/NEXT loop in BASIC. It repeats everything which comes between the words DO and LOOP. The number of repetitions is the difference between the top two numbers on the stack. If the second number on the stack was 7 and the top number was 1, the loop would be run through six times.

That brings us to the end of this section of the book. Work through it a number of times until you can use all the words confidently, and write some programs of your own.

Further Reading

The two best books we've come across on FORTH are:

Starting FORTH — Leo Brodie, Forth Inc. (Prentice-Hall, 1981)

FORTH Programming — Leo J Scanlon (Howard Sams, 1982)

Section Five
Sorting and Searching

Many computer programs are dedicated to the laudable aim of bringing order into an increasingly chaotic universe. Industrious little routines spend their time sorting numbers and names into order, comparing them, taking actions on the basis of those comparisons, and assigning the results of their deliberations into neatly-defined pigeon holes.

However, order is not always wanted. Just suppose you wanted to survey a randomly-selected 10% of the registered voters of Wagga Wagga. It would not do simply to take the first 10% of the names on the list, nor those who live in 10% of the residential part of town. To get a fair sample, which was genuinely random, you would need to select the names on some other basis.

In this section, we look first at a couple of routines for selecting items at random from a list. Then, I'll be looking at three ways of finding specific elements of data within lists. The method chosen can radically alter the time in takes for an item in a list to be located.

The Non-Recurring Shuffle

We'll start by supposing wanted to question three Wagga Wagga residents in each street. Suppose, further, that each street in the town only has houses numbered one to ten. How would you go about deciding, for any particular street, which three numbers you would go to? One way would be to generate lists of random numbers, between one and ten, and go to the first three house numbers from that list. But what would you do if your random number generator came up with a list like 5,3,5,3,5? You need a routine which, while producing random numbers, does not produce each one more than once.

It is pretty simple to create a routine to fill an array of ten elements with random numbers in the range one to ten. It is also simple, although it requires a bit more thought, to write a routine which fills the array with numbers chosen randomly in the range one to ten, with each number appearing once, and once only, in randomly-determined positions.

If you run this program, and enter the number '10' when prompted to do so by the question "Range of numbers?", you'll see that it rapidly produces the numbers you need. The elements of the B array keep track of whether or not a number has been previously produced:

```
10 REM MOSES/OAKFORD SHUFFLE
20 REM NON-RECURRING RANDOM NUMBERS
30 CLS:INPUT "RANGE OF NUMBER";N:CLS
40 DIM A(N)
50 FOR J=1 TO N:A(J)=J:NEXT J
60 INPUT "PRESS RETURN TO START";A$
70 FOR J=N TO 1 STEP -1
80 T=RND(J)
90 TEMP=A(T):A(T)=A(J):A(J)=TEMP
100 NEXT J
110 SOUND 20,3:PRINT "FINISHED"
120 FOR J=1 TO N:PRINT A(J),:NEXT J
```

Sequential Searches

Suppose, instead of wanting a sample chosen at random from the whole population of Wagga Wagga, you wanted to question those who had reported incomes in excess of $20,000 a year in the last census. And further suppose that no list had been made, in numerical order, of income size. To locate those in the income bracket you want, you'd have to go through the whole census output, figure by figure, to isolate the ones you wanted.

And, sad to say, a computer would have to do the same thing. If the list is disordered, there is no way of cutting short the process of going through it, element by element, until the required one is found.

The next program demonstrates a sequential search. A variation of the Moses/Oakford shuffle routine is used to fill an array with randomly-generated numbers, and then the program looks for them:

```
10 REM SEQUENTIAL SEARCH
20 CLS
25 INPUT "HOW MANY ELEMENTS TO SEARCH     THROUGH";Q
30 CLS:DIM A(Q)
```

```
40 INPUT "PRESS RETURN TO BEGIN";H$
50 CLS:FOR J=1 TO Q:A(J)=J:NEXT J
60 FOR J=Q TO 1 STEP -1:T=RND(J)
70 TEMP=A(T):A(T)=A(J):A(J)=TEMP:NEXT J
80 CLS
85 INPUT "ENTER NUMBER TO BE SEARCHED     FOR ";N
90 N=INT(N):IF N<1 OR N>Q THEN 80
110 X=0
120 X=X+1:IF A(X)=N THEN 150
130 IF X<Q THEN 120
140 PRINT "I CANNOT FIND"N:GOTO 160
150 SOUND 25,3:PRINT "SEARCH COMPLETE"
155 PRINT "IT WAS AT POSITION"X
160 PRINT "PRESS RETURN FOR A NEW SEARCH"
170 INPUT S$:GOTO 80
```

What is the relationship between the number of items in the list, and the time it takes to locate any one of them? A moment's thought will show that if there are N items, on average half the time the element you're looking for will be in the first half of the list, and the rest of the time it will be in the second half. That is, the 'average position' (to use the term very loosely) of the item you're looking for will be exactly half-way through the list. The longer the list, the longer it will take to reach the half-way point, so — on average — it takes N/2 time to search sequentially through a list of N items.

However, in real life, we rarely deal with completely random lists, in which every item is needed an approximately equal number of times.

As I live in an extremely low-tech household, the ten or so telephone numbers I use most often are written on a piece of cardboard near the phone (no autodial facilities for me). If I bothered to log the calls I made in any four-week period, I am sure that one or two of them would be used far more times than the rest. Of the remainder, perhaps three or four would be the next most-often used, with the final few hardly ever being used. Your telephone usage is probably very much the same.

Now, assume I had my telephone directory on disk, and it contained some 1000 names and numbers, added from time to time over the years. Whenever I needed a number, the computer would have to search through the list. And, if my two most-often-used numbers were right at the end of the list, it would always take the computer close to the maximum possible time to find them. A list which 'knew' which elements were needed more often than others, and could re-arrange itself so that often-used items were closer to the start of the list than the end, would be very useful. Then, at the end of each day's work, I could resave my directory on disk, and eventually the numbers I called every day would be at the start of the list, where they would be found almost instantly from the 1000 numbers, and those I used rarely would be closer to the end of the list.

For a more realistic example of the usefulness of a self-organising list, imagine a car parts warehouse where every item in stock has a reference number. However, most people who ring up to find out if an item is in stock do not use the warehouse's reference number, preferring instead to say things like "a rear wheel brake shoe for the 1968 model". To save having to look up, in some vast ledger, the relevant reference number so an order can be raised, the warehouse has a computer system set up. The clerk types in "brake shoe, rear, 1968", the computer searches through its thousands of parts, and eventually prints up on the screen "IV663" as the part number which the clerk writes onto the order.

Now, there will be some parts which will be asked for far more often than others. Brakes, for example, wear out on all cars, so there will be a constant call on those parts. Other things, such as a replacement winder for the left rear window, are probably requested far less often. So if the 'rear brake shoe' is near the end of the computer's list of parts, it will take an unnecessarily long time to find out that the part number is "IV663", compared to the time it would take if this particular part was closer to the start of the list.

Our next program, the Self-Organising Search, goes some of the way towards solving this problem. Once it finds a requested item (X in this case, which is located at element number P), it swaps it with the item which follows it, moving it closer to the start of the list. You can test this program by asking for the same item over and over again, seeing how it moves closer to the start of the list each time:

```
10 REM SELF-ORGANIZING SEARCH
15 CLS
20 INPUT "HOW MANY ELEMENTS TO SEARCH    THROUGH";N
30 CLS:DIM A(N+1)
40 PRINT "PLEASE STAND BY..."
50 FOR J=1 TO N:A(J)=J:NEXT J
60 FOR J=N TO 2 STEP -1:T=RND(J)
70 TEMP=A(T):A(T)=A(J):A(J)=TEMP:NEXT J
80 CLS
```

```
85 INPUT "NUMBER TO BE SEARCHED FOR";X
90 X=INT(X):IF X<1 OR X>N THEN 80
110 A(N+1)=X
120 P=0
130 P=P+1
140 IF A(P)=X THEN 170
150 IF P<N THEN 130
160 PRINT "ELEMENT NOT FOUND":GOTO 220
170 IF P=1 THEN 210
180 TEMP=A(P-1):A(P-1)=A(P):A(P)=TEMP
200 P=P-1
210 PRINT "IT IS AT ELEMENT";P
220 PRINT "PRESS RETURN"
230 INPUT J$:GOTO 80
```

Binary Searching

If I asked you to guess a number I was think of, between one and 100, you'd probably start by saying "50". When I said "Higher", your next guess would then sensibly be "75". A reply of "lower" would prompt you to guess "62" or "63" and so on, until you'd narrowed down the field to the correct number.

Even though you may not have known it, you were conducting a binary search for the needed number. The binary search is much faster than the sequential search, and is ideal in any case when the items you are searching through are in order. If you had a list of Wagga Wagga incomes, ranked from my income level of $110.07 a year, up to $245,000 and you told the computer to find the first occurrence on the list of $20,000, a binary search would probably find the $20,000 before a sequential search did so.

The binary search program works in exactly the same way as you would when trying to guess the number I was thinking of between one and 100. It compares X (the number you're looking for) with the middle element of the list. If they are the same, the search is over, and the progbram goes on to tell you where the needed element is in the list. If the middle element is not the item you are looking for, the comparisons tell it which half of the list to examine next.

It searches this half in the same way, starting by looking at the middle element. In the program, the variables L and R stand for 'left' and 'right' of the section of the list being examined:

```
10 REM BINARY SEARCH
20 CLS
25 INPUT "HOW MANY ELEMENTS TO SEARCH     THROUGH";N
30 CLS:DIM A(N),Q(N),C(N)
40 PRINT "PLEASE STAND BY..."
50 FOR J=1 TO N:A(J)=RND(J):NEXT J
60 FOR J=1 TO N:C(A(J))=C(A(J))+1:NEXT J
70 FOR J=2 TO N:C(J)=C(J)+C(J-1):NEXT J
80 FOR K=N TO 1 STEP-1
90 TEMP=A(K):J=C(TEMP):Q(J)=TEMP
100 C(TEMP)=J-1:NEXT K
110 FOR J=1 TO N:A(J)=Q(J):NEXT J
120 CLS
130 INPUT "ENTER NUMBER TO BE SEARCHED     FOR";X
140 L=1:R=N
150 P=INT((L+R)/2)
160 IF X<A(P) THEN 190
170 IF X=A(P) THEN 230
180 L=P+1:GOTO 200
190 R=P-1
200 IF L<=R THEN 150
210 P=0
230 IF P<>0 THEN PRINT "IT IS AT POSITION";P:GOTO 250
240 PRINT "IT IS NOT IN THE LIST"
250 PRINT "PRESS RETURN"
260 INPUT E$:GOTO 120
```

Random Numbers

Your VZ300, as you know, comes with an inbuilt function to generate random numbers. Actually, the numbers are not really random, as they are the result of a decision — or decisions — made by the computer, in line with an inbuilt program. This program dictates specific actions in response to specific situations. Therefore, if you knew the computer's inner program, and what it was responding to, you'd be able to predict exactly which 'random' number it would select next.

Fortunately, although the computer chooses each number from a list, and then repeats the list when it gets to the end, the list is so long you'd have a pretty difficult time trying to work out where the list began again. One popular make of computer, for example, when you wind it up fully, can produce a random number every 1.6 milliseconds. If you let it go on generating these numbers, it would take 150 days before the sequence began to repeat itself.

How does your VZ300 create its random numbers?

There are many 'random-number' algorithms in existence. An early one was developed by one of the grandfathers of computers. John von Neumann worked out a method of generating random numbers based on taking a four-figure number (such as 8931), then squaring it (to produce, in this case, 79762761), and from that selecting the middle four digits (7627). These were used as the first random number, then they were squared (58171129) to create the next number in the sequence (1711) and so on.

Here's a program to create von Neumann numbers on your VZ300. When it starts, enter any four-digit number. It will run for a while, then stop, expecting a new input. You can stop the program at any time by entering a number which is less than 1000:

```
10 REM VON NEUMANN NUMBERS:
20 REM ENTER NUMBER BELOW 1000 TO END
30 CLS
40 PRINT:INPUT "ENTER NUMBER";A
50 IF A<1000 THEN END
60 B$=STR$(A*A)
70 A=VAL(MID$(B$,4,4))
80 PRINT A;
90 IF A>999 THEN 60
100 GOTO 40
```

As you'll soon discover, this does not produce the world's most satisfactory random numbers. In many cases, the numbers start to repeat fairly quickly.

Now most random number generators inside microcomputers use a formula along the lines of SEED=(ANUMBER*SEED*ANOTHER NUMBER) MOD YETANOTHERNUMBER. SEED is then fed back into the formula for the next run through. Modular division returns the *remainder* of a division (so 10 MOD 3 is 1) and not all computers include MOD in their vocabulary. However, it is pretty simple to simulate it. Here's a simple program to generate random numbers using an approach similar to the one which occurs deep in your computer's electronic innards:

```
10 REM MODULAR SEEDS
20 CLS
30 INPUT "FIRST BIG NUMBER";A
```

```
40 INPUT "SECOND BIG NUMBER";B
50 INPUT "NOW A LITTLE NUMBER";C
60 INPUT "AND NOW THE SEED";SEED
70 SEED=((A*SEED+B)/C)-INT((A*SEED+B)/C)
80 PRINT SEED;
90 GOTO 70
```

The first two numbers (A and B) should be pretty big, and the next two (C and SEED) should be relatively small. For a run which continues for a fair while without repeating, try 1478392 for A, 5228791 for B, 778 for C and 459 for SEED.

How random are the numbers produced by the VZ's generator?

It is pretty easy to find out how random the numbers are by writing a program which not only generates the numbers, but also works out their distribution:

```
10 REM DISTRIBUTION OF NUMBERS
20 REM FOR VZ 300
30 CLS
40 DIM A(10)
50 FOR J=1 TO 1000
60 B=RND(10)
70 A(B)=A(B)+1
80 NEXT J
90 FOR J=1 TO 10
100 PRINT J;" > ";A(J)/10;"%"
110 NEXT J
```

As you can see, this program stores the frequency with which the numbers are generated in an array, then prints the frequency out as a percentage of the whole run.

I ran the program twenty times, and took an average of the results. If the random number in my VZ was perfect, and I ran the program for an infinite time, each number from one to ten in my sample would occur exactly 10% of the time. As you can see, even with the relatively small sample, the output is pretty close to this ideal distribution:

```
1  >  10.08 %
2  >  10.035 %
3  >  10.245 %
```

```
4   >   10.12 %
5   >   9.845 %
6   >   9.805 %
7   >   9.97 %
8   >   9.859999 %
9   >   9.984999 %
10   >   10.055 %
```

Try it on your VZ300, and see how the results compare with mine.

Now there may be times, say when creating computer simulations, when you want *skewed* random numbers; numbers which are biased in some way, rather than being evenly distributed across the range. This is fairly easy to do. If you want, for example, the lower numbers to appear more often than the higher ones, all you have to do is change line 60 of the above program to:

```
60 B=INT(RND(0)*RND(0)*10+1)
```

I did this, and ran the program five times, and again averaged the results. This is what I got:

```
1   >   33.1 %
2   >   18.68 %
3   >   14.68 %
4   >   10.06 %
5   >   7.26 %
6   >   6.04 %
7   >   4.7 %
8   >   3.1 %
9   >   1.6 %
10   >   .78 %
```

How does this work? Simply by the fact that RND(0) produces a number between zero and one, and multiplying any such number with another similar one produces numbers which tend to be lower (i.e. towards zero) than higher.

John von Neumann, who invented the 'pick a four-figure number, then square it' method of generating random numbers, also developed a rather neat way of working out areas enclosed by an irregular border, based on random numbers. His method is called the Monte Carlo Method. It works on the basis that, if you had a map of an area containing a single continent, and you dropped darts on the map randomly, and then counted how many darts fell within the continent on the map, and how many fell outside it, the area of the continent would be *proportional* to the number of darts which fell within it, compared to those which fell outside it. By knowing the area covered by the whole map, it would be simple to work out an approximation to the area of the continent.

We can use such a method to work out an approximation to PI. Imagine a square, with a circle drawn in the square which just touches the sides. Now, mentally divide the square, and the circle, into four. Throw away three-quarters of the square, and keep the remaining quarter, which contains a quarter circle.

Now, imagine that you were dropping darts on the square in such a way that they had an equal chance of falling anywhere within it. Some would land within the quarter circle, and some would land outside it. If the darts were dropped in a perfectly random manner, the ratio between those which fell within the quarter circle, to those which fell outside it, would be PI divided by four. This program 'drops the darts' for you:

```
10 REM MONTE CARLO PI
20 REM FOR VZ300
30 CLS
40 A=0:B=0
45 LPRINT A;ABS(3.141593-P);TAB(23);P:GOTO 50
50 GOSUB 120
60 B=B+D
70 A=A+1
80 P=4*B/A
90 PRINT A;ABS(3.141593-P),P
100 IF 500*(INT(A/500))<>A THEN 50
110 LPRINT A;ABS(3.141593-P);TAB(23);P:GOTO 50
120 D=0
130 M=RND(0)
140 Z=RND(0)
150 IF M*M+Z*Z<1 THEN D=1
160 RETURN
```

You can see, in line 90, that I've used 3.141593 as an approximation to PI, to check the accuracy of the value of PI produced by the program. The program prints out, in line 90, the number of darts you've dropped (A), the difference between 3.141593 and the number you're calculating as an approximation to PI (3.141593—P) and, finally, 'your' version of PI (P).

After 'dropping' 500 darts, the first time I ran the program, I got a value of 3.088 for PI, an error of around .0536. This is not too bad. However, I didn't think it was good enough, so ran the program under it had dropped 74,000 darts (patience required here), and got this output towards the end of that run:

```
67000  5.5337E-03       3.13606
67500  5.53417E-03      3.13606
68000  5.82886E-03      3.13577
68500  5.24306E-03      3.13635
69000  3.97038E-03      3.13762
69500  4.09698E-03      3.1375
70000  3.30782E-03      3.13829
70500  3.21054E-03      3.13838
71000  2.94566E-03      3.13865
71500  3.07608E-03      3.13852
72000  2.14911E-03      3.13944
72500  1.89686E-03      3.1397
73000  2.25091E-03      3.13934
73500  1.89281E-03      3.1397
74000  2.07996E-03      3.13951
74500  1.83511E-03      3.13976
```

It is very interesting to watch as the program 'homes in' on the value of PI.

Now that we've looked at the mysteries of PI, it is time to examine computer sorting techniques. The majority of business programs use sorts. In fact, according to Jonathan Amsterdam, writing in *Byte* magazine (September, 1985, p. 105), 90 per cent of all computer programs do some kind of sorting.

The most basic sorting needs are for a series of strings to be placed in alphabetical order, or for numbers to be placed in an ascending or descending series. Whether it is names of products in a storeroom, a mail list to be ordered by postcodes, or a ranking of examination results from highest to lowest within a class, similar sorting techniques can be used.

However, there is a bewildering number of sorting algorithms, and they differ wildly in their efficiency. We'll be looking at five different sorting techniques in this section, in which the most efficient one works thirty times faster than the least efficient one does. You'll probably find it quite interesting to run the different sorts on your own computer, and time them as they sort out lists of your own. The difference in speed — which will be, of course, most noticeable with a long list — is quite amazing.

Although it makes little practical difference which sort you use when the list to be sorted is short, it becomes increasingly important as the length of the list grows. And if you're writing a business applications program which either sorts a long list from time to time, or sorts short lists frequently, it is very important to choose the most efficient sort.

Speed and Storage

Tony Guttmann, a lecturer at the University of Newcastle, NSW, in his book *Programming and Algorithms* (Heinemann, 1977; p.146), points out that choosing the correct sort for a job often involves a compromise between various incompatible requirements. "The two most commonly conflicting requirements," he writes, "are *storage space* and *execution time.*" Some sorts, as we will see, demand no additional memory than that which holds the original, unsorted data. In the worst case, a second array equal in size to that which holds the original data is needed to hold elements during a sort. The other sorts lie in between these two extremes.

In each of the programs here, the list to be sorted is an array filled with random numbers, which are then sorted into ascending order. The number of elements in the list can easily be altered, to demonstrate clearly that the efficiency of some sorts declines quite dramatically as the length of the list to be sorted increases.

Bubble Sort

We'll start with the *Bubble Sort*:

```
10 REM BUBBLE SORT - A
20 CLS
25 INPUT "HOW MANY ITEMS TO BE SORTED";N
30 CLS:DIM A(N)
40 FOR Q=1 TO N:A(Q)=RND(N):NEXT Q
45 SOUND 15,3
50 PRINT "SORT STARTING NOW..."
60 K=1
70 X=A(K):Y=A(K+1)
80 IF X<Y THEN 140
90 A(K)=Y:A(K+1)=X:TEMP=K-1
100 IF TEMP=0 THEN 140
110 X=A(TEMP)
115 Y=A(TEMP+1):IF X<Y THEN 130
120 A(TEMP)=Y:A(TEMP+1)=X
```

```
130 TEMP=TEMP-1:GOTO 100
140 K=K+1:IF K<N THEN 70
150 PRINT "SORT FINISHED:":SOUND 20,3
160 FOR J=1 TO N:PRINT A(J),:NEXT J
```

In this, the computer looks at the first two elements in the list — A(K) and A(K+1) — and swaps them over if necessary. Next, the program looks at elements two and three in the list, and interchanges them if necessary. Once it has got right to the end of the list on the first pass, the bubble sort program goes back and does it over and over again, until the list is in order. The time a bubble sort takes to order a list is proportional to the square of the number of elements to be sorted. My VZ took 46 seconds to sort a list of 50 elements.

Swap Sort

The Bubble Sort, even though it was slow, did not demand additional memory to hold the elements of the list as they were sorted. Similarly, the Swap Sort does not need extra memory:

```
10 REM SWAP SORT - B
20 CLS
25 INPUT "HOW MANY ITEMS TO BE SORTED";N
30 CLS:DIM A(N)
40 FOR M=1 TO N:A(M)=RND(N):NEXT M
50 SOUND 20,3:PRINT "SORT STARTING NOW"
60 FOR B=1 TO N-1
70 FOR C=B+1 TO N
80 IF A(B)<=A(C) THEN 100
90 TEMP=A(B):A(B)=A(C)
95 A(C)=TEMP
100 NEXT C:NEXT B
110 CLS:SOUND 23,2
115 PRINT "SORT FINISHED:"
120 FOR J=1 TO N:PRINT A(J),:NEXT J
```

Starting with the first two elements on the list, this sort interchanges them if necessary. If they do not need to be swapped over, the program looks at the next two. If the first two need to be swapped, the swap is made, and then the program goes back to the beginning. This occurs until it gets to the end of the list.

Whereas it took the Bubble Sort 46 seconds to put a list of 50 items in order, the Swap took just 23 seconds. When the length of the list to be ordered was increased by a factor of three (to 150), the Bubble Sort time increased by 93%, while the Swap Sort time increased by around 96%. This suggests that while the time it takes the sorts to work naturally increases as the length of the list increases, the Swap Sort may degrade to a greater extent. Try both programs with lists of 1000, and then more, numbers, and see if you can work out at which point — if any — a Bubble Sort would become more efficient than a Swap Sort.

Insertion Sort

Like the first two sorts we've looked at, the Insertion Sort does not demand additional memory. Whereas the time taken to sort a list with the Swap Sort is related to the number of elements in the list cubed, the time the Insertion Sort takes to order a list is related to the number of items squared.

Here is the listing:

```
10 REM INSERTION SORT - C
20 CLS
25 INPUT "HOW MANY ITEMS TO BE SORTED";N
30 CLS:DIM A(N)
40 FOR Q=1 TO N:A(Q)=RND(N):NEXT Q
50 SOUND 15,3:PRINT "SORT STARTING NOW"
60 FOR K=2 TO N
70 J=K-1:L=A(K)
80 IF L>=A(J) THEN 110
90 A(K J+1)=A(J)
100 J=J-1:IF J>0 THEN 80
110 A(J+1)=L:NEXT K
120 SOUND 16,3:PRINT "SORT FINISHED"
130 FOR J=1 TO N:PRINT A(J),:NEXT J
```

It took 4½ seconds to sort 50 elements, and 12 seconds to sort 100.

Shell Sort

Now we're moving into the Brands Hatch area of sorts, where things really start zipping along. The Shell Sort, although it needs a little extra storage (in this case, an array containing 10 elements), is very fast.

According to D E Knuth, in his book *The Art of Computer Programming* (Addison-Wesley, 1973) it works by filling the elements of the S array with a set of increasing integers starting with S(1)=1. "The best set is not known," he writes, "but the sequence 'S(J+1)=S(J)*3+1' is good." Once this is done, the program finds the smallest value P such that S(P+2)V=N (where N is an element in the list to be sorted). Then, for each S=S(K), where K is a loop control variable in a FOR/NEXT loop going down from P to 1 take each value of J from S+1 to N, and insert A(J) in its proper position.

```
10 REM SHELL SORT - D
15 CLS
20 INPUT "HOW MANY ITEMS TO BE SORTED";N
30 CLS:DIM A(N),S(10)
40 FOR M=1 TO N:A(M)=RND(N):NEXT M
50 PRINT "SORT BEGINNING":SOUND 19,3
60 S(1)=1
65 FOR J=1 TO 9:S(J+1)=S(J)*3+1:NEXT J
70 P=0
80 P=P+1
90 IF S(P+2)<N THEN 80
100 FOR K=P TO 1 STEP-1:S=S(K)
110 FOR J=S+1 TO N
115 L=J-S:A=A(J)
120 IF A>=A(L) THEN 140
130 A(L+S)=A(L):L=L-S:IF L>0 THEN 120
140 A(L+S)=A:NEXT J
150 IF K>1 THEN FOR Q=1 TO N:PRINT A(Q),:NEXT Q
160 PRINT:PRINT:NEXT K
180 SOUND 17,3
185 PRINT "FINAL SORTED LIST"
190 FOR J=1 TO N:PRINT A(J),:NEXT J
```

Complex as that 'explanation' may seem, you don't need to be able to make sense of it in order to use the Shell Sort.

Sort by Count

The final sort we will examine, and the one which puts all the others to shame in terms of speed of execution is the Sort by Count, which needs an array in addition to the one which holds the original data. The second array (C in our program) contains the same number of elements as the *value* of the largest element in the data (so if the numbers in the original data were 6, 84 and 17, C would need 84 elements):

```
10 REM SORT BY COUNT - E
20 CLS
25 INPUT "HOW MANY ITEMS TO BE SORTED";N
30 CLS:DIM A(N),Q(N)
40 INPUT "HIGHEST VALUE IN DATA";M
50 CLS:DIM C(M)
60 FOR Q=1 TO N:A(Q)=RND(M):NEXT Q
70 PRINT "SORT STARTING NOW":SOUND 12,3
80 FOR J=1 TO M:C(J)=0:NEXT J
90 FOR J=1 TO N:C(A(J))=C(A(J))+1:NEXT J
100 FOR J=2 TO M:C(J)=C(J)+C(J-1):NEXT J
110 FOR K=N TO 1 STEP-1
120 TEMP=A(K)
130 J=C(TEMP):Q(J)=TEMP:C(TEMP)=J-1
140 NEXT K
150 CLS:SOUND 14,3
160 PRINT "SORT FINISHED"
170 FOR J=1 TO N:PRINT Q(J),:NEXT J
```

The cost of this storage overhead is well worth paying, as the time to sort a list of N elements is directly related to N. Instead of finding the time taken increases as the square, or cube of the number of elements in the list, the Sort by Count time increases only arithmetically with the number of elements (so the time taken to sort a list of 100 elements should be exactly double the time it takes to sort 50 elements).

The program works by setting every element of the C array to zero. Then, for each element of array A, the program increments its corresponding counter C(A(J)). This means that C(J) is now set to the number of elements in the original list of data equal to J. Next, the program counts from 2 up to M (where M is, you'll recall, the value of the largest number in the original list), adding each C(J) to C(J-1), as you'll see in line 100. This makes each C(J) the number of elements less than or equal to J.

Finally, using a loop counting backwards from N (the number of items in our original list) to 1 (see line 110), each element A(K) is copied (holding the value temporarily in the variable TEMP) to Q(C(A(K))) and C(A(K)) is decremented.

At the end of all this shenanigans, we have a sorted list.

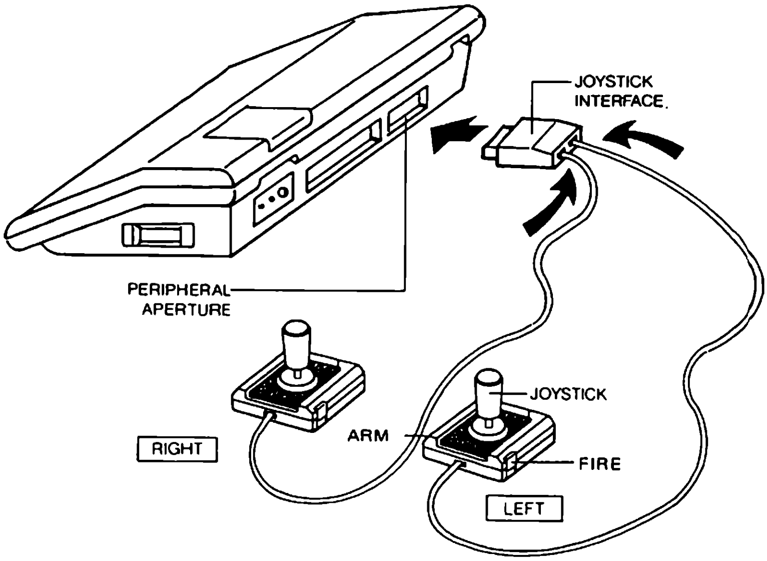
Section Six Disk Drives and Other Peripherals

The word 'peripheral' is used to describe anything which you connect up to your VZ300, whether it is a printer, a joystick, or a disk drive. In this section of the book, we'll look at the peripherals you can get for your VZ300, and also examine the commands used for controlling disks.

Joysticks

The joysticks (DSE catalogue product number X-7315) come as a pair. They are connected up as shown in the diagram.

The fast response joysticks offer you eight-direction flexibility and both an



ARM and FIRE button. The joystick interface allows your VZ300 to support the joysticks.

Make sure, when connecting any peripherals to your computer, that you have disconnected the power.

To install the joysticks, you turn off the power, and then remove the cover marked 'peripheral' at the back of the VZ300. You then plug the joystick interface into the peripheral socket slowly and smoothly.

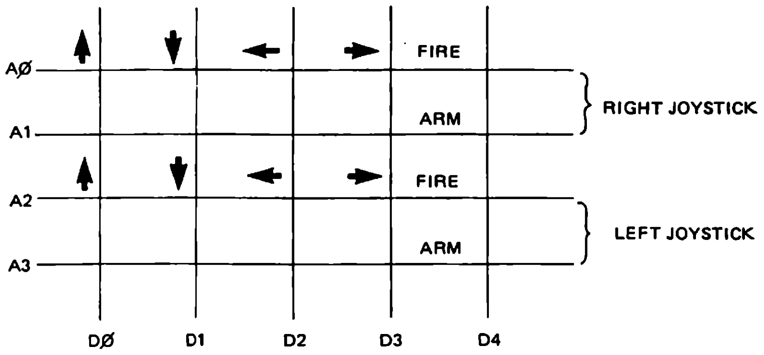
Next, you turn on the VZ300. If you have connected the joysticks correctly, the READY message will show as normal. If it doesn't come up on the screen, you simply go through the procedure again.

You can control your BASIC programs using the joysticks. Here's a simple demonstration program to show this:

```
10  CLS
20  A =(INT(43) AND 31)
30  IF A=30 THEN PRINT "UP"
40  IF A=29 THEN PRINT "DOWN"
50  IF A=27 THEN PRINT "LEFT"
60  IF A=23 THEN PRINT "RIGHT"
70  GOTO 20
```

This program controls the *left* joystick. You press control/break to stop the program.

You can access the right or left joysticks in assembly language, by using the matrix shown in the illustration.

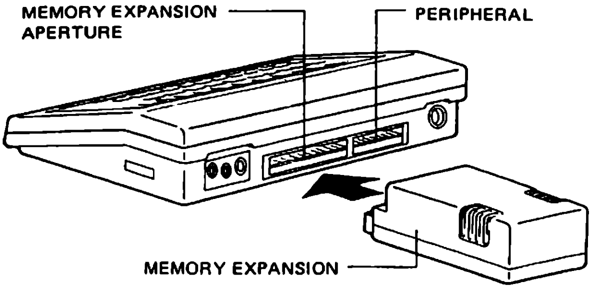


You use the IN instruction to read the joystick whose address ranges from hexadecimal 20 to 2F. All you have to do is write a program to scan the address lines, and check which data bit has become zero.

Adding Extra Memory

From time to time you'll come across programs which demand more memory than is provided with the standard VZ300. You may also find, as you become more experienced with programming your computer, that you want to write longer and more complex programs, and additional memory will be vital at this stage.

The 16K memory expansion model (catalogue number X-7306) simply plugs into the memory expansion aperture at the back of the computer. Make sure, of course, that you have the power off before plugging the memory pack in place as illustrated.

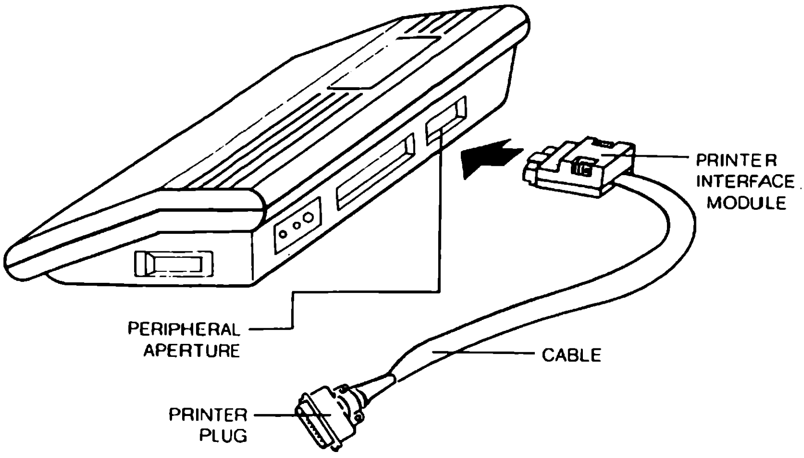


To check that the extra memory is correctly inserted, you type in PRINT PEEK (30897) and press the RETURN key, which should give you an answer of 255. Then, if you tyupe in PRINT PEEK (30898) and press RETURN, you'll get 247.

Connecting a Printer

The Printer Interface (catalogue number X-7320) allows your VZ300 to support the standard Centronics type printers. The printer interface module lugs into the peripheral aperture at the back of the computer.

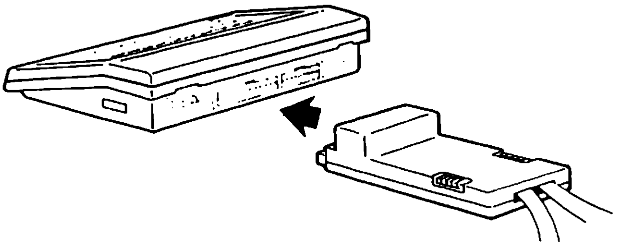
You use the command LLIST to get a program listing printed out on the screen, and LPRINT to print a specific line. These can be used within programs, as can the third command, COPY, which is used to get a copy of everything on the screen at any time.



Although COPY will cause the printer to print out the *text* on the screen, the graphics and inverse characters which you can get on the VZ300 can only be LLISTed, LPRINTed or COPYed if your printer is the DSE GP–100 (catalogue X-3250) or an SHA GP-100A.

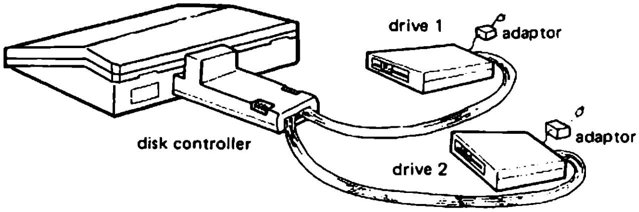
Adding Disks to your VZ300

The VZ300 Floppy Disk Controller (catalogue number X-7304) is needed to allow you to connect one or two disk drives to your VZ300. The controller plugs into the back of your VZ300.



The controller communicates with the disk drive or drives via a 20-pin connector which provides all input/output signals.

The disk drive itself (catalogue number X-7302) comes with a separate power supply. You need to plug that into the power, and plug the lead from the end of the disk controller into the drive, using the socket on the controller marked DRIVE 1 for the first drive, and DRIVE 2 for the second one. You also need a 16K or 64K memory expansion unit, which plugs into the RAM expansion slot on the disk controller.



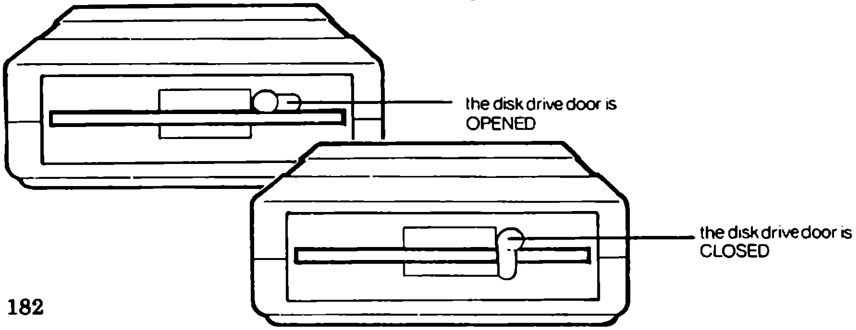
The drives use standard 5.25 single-sided floppy disks, and each disk can hold 80K of programs.

When you turn the disk drive on, if all is well, you'll get a slightly different READY message from the one you're used to:

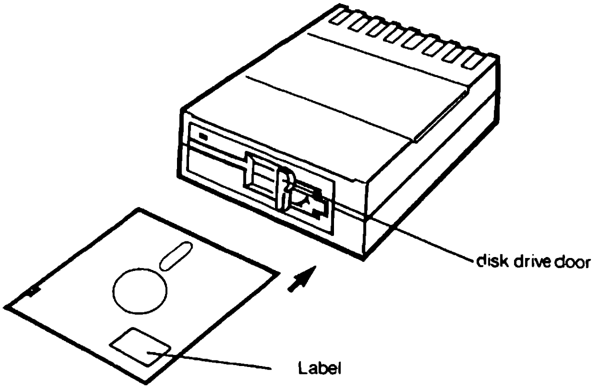
```
VIDEO TECHNOLOGY
DOS BASIC V1.2

READY
```

DOS stands for Disk Operating System.

To use the drive, you first must put a disk in place. You open the disk drive door, and insert the disk so that the label goes in last. Then, you 'lock' the door by turning the little handle to that it points downwards.



Before you can use a disk, it must be *initialised*. You do this with the command INIT. When you type this in, and press RETURN, you'll see the red light on the disk drive come on. Never try to remove a disk while the red light is on. After some whirring noises, which will continue for quite a while, the light will go out, and the disk will be initialised. Note that the initialisation process wipes *everything* on a disk, so make sure you use this command with care.



To find out what is on a disk, you use the *directory* command, which is DIR. You simply type it in, and press RETURN, and a list of all the files on your disk will come up on the screen.

To save a program to disk, you simply type in SAVE "NAME". "NAME" can be up to eight characters long. To load a program, you use the same approach, with the command LOAD "NAME". Note that the "NAME" must be used for both loading and saving, and the quote marks must be present on each side of the name.

To run a program, you can either load it in, and then type in RUN, and press RETURN, or you can enter RUN "NAME" and press RETURN.

The STATUS command is used to find out how much space you've got left on a particular disk. You simply type in the word STAT and press RETURN. The drive will whirr for a few seconds, and a message like the following will appear:

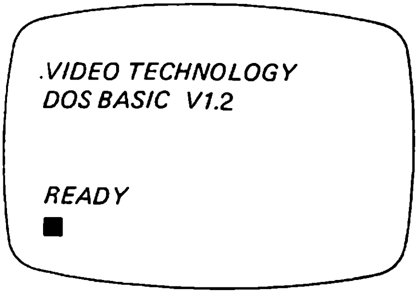
```
624 RECORDS FREE
78.0K BYTES FREE
```

This report indicates that there are 624 'records' (each of 128 bytes), or 78K, of space still on the disk.

If you have two disk drives connected, you swap between them by using the commands DRIVE 1 and DRIVE 2. Once you've typed one of these commands in, all disk commands will go to the indicated disk drive.

The command REN is used to *rename* a file on a disk. If you've saved a program under the name "GOODGAME" and you want to change the name to "AMAZING", you just type in REN "GOODGAME", "AMAZING" and then press RETURN. The drive will make its customary noises, and then when you next try a DIR command, you'll see the file is listed under the second name.

You can't save a program using a name which has already been used on that disk. Therefore, if you have an updated version of a program which you wish to save, or you simply want to get rid of a program from your disk, you need the ERA command, which stands for *erase*. You just type in ERA "PROGNAME" and then program called "PROGNAME" will be wiped.

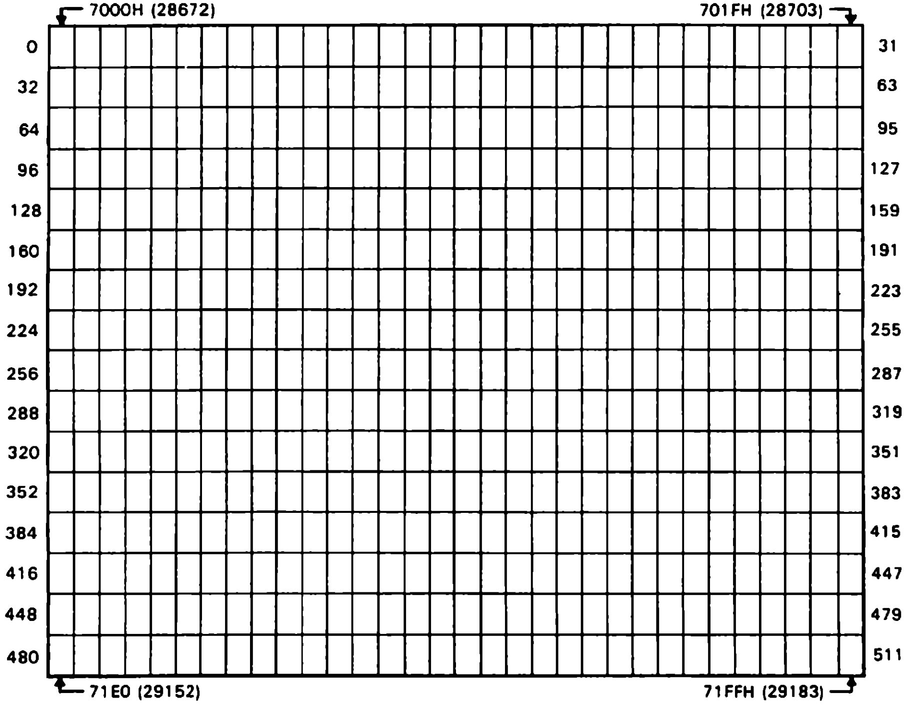


```
.VIDEO TECHNOLOGY
DOS BASIC  V1.2


READY
■
```

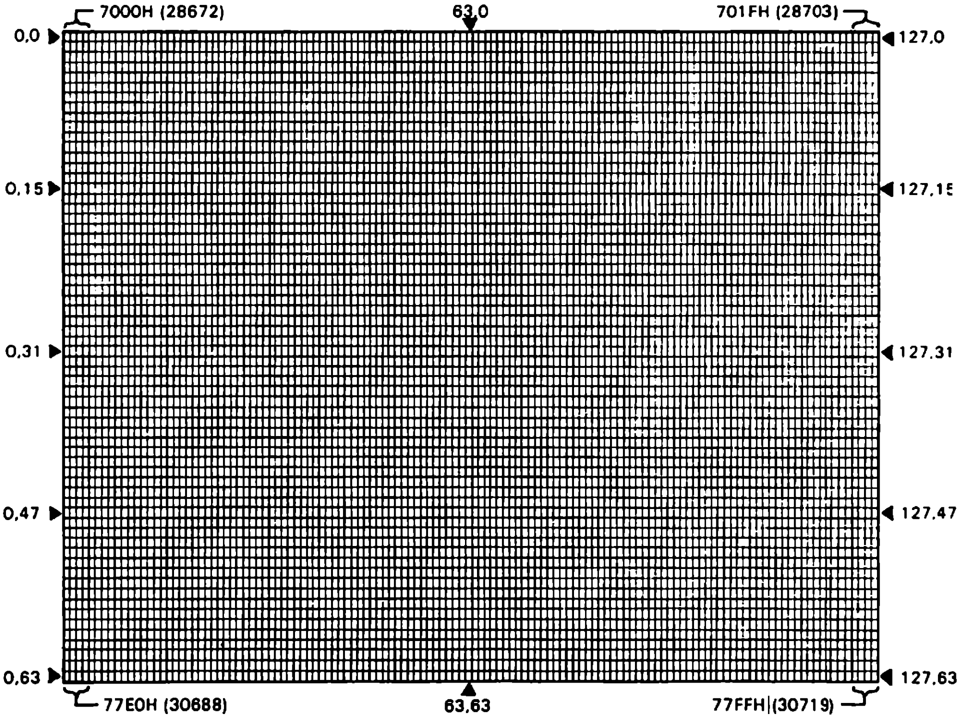
Appendix
Reference Section

Finally, in this appendix, we have video display worksheets for Mode O and Mode 1, the VZ's ASCII code table, and the computer's character codes.

VIDEO DISPLAY WORKSHEET (MODE 0)



VIDEO DISPLAY WORKSHEET (MODE 1)



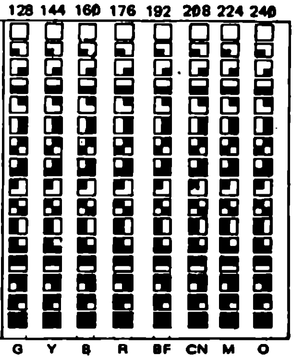
ASCII Code Table

ASCII CODE	CHARACTER	ASCII	CHARACTER
32	(Space)	64	@ (at sign)
33	! (exclamation point)	65	A
34	" (quote)	66	B
35	# (number or pound sign)	67	C
36	$	68	D
37	% (percent)	69	E
38	& (ampersand)	7Ø	F
39	' (apostrophe)	71	G
4Ø	((open parenthesis)	72	H
41	) (close parenthesis)	73	I
42	* (asterisk)	74	J
43	+ (plus)	75	K
44	, (comma)	76	L
45	– (minus)	77	M
46	. (period)	78	N
47	/ (slant)	79	O
48	Ø	8Ø	P
49	1	81	Q
5Ø	2	82	R
51	3	83	S
52	4	84	T
53	5	85	U
54	6	86	V
55	7	87	W
56	8	88	X
57	9	89	Y
58	: (colon)	9Ø	Z
59	; (semicolon)	91	[(open square bracket)
6Ø	< (less than)	92	\ (back slash)
61	= (equals)	93	] (close square bracket)
62	> (greater than)	94	↑ (up arrow)
63	? (question mark)	95	← (back arrow)

Character Codes

NO.

Relative Offset	0	16	32	48	64	80	96	112
+0	●	P		0	●	P		0
+1	A	Q	!	1	A	Q	!	1
+2	B	R	"	2	B	R	"	2
+3	C	S	#	3	C	S	#	3
+4	D	T	$	4	D	T	$	4
+5	E	U	%	5	E	U	%	5
+6	F	V	&	6	F	V	&	6
+7	G	W	'	7	G	W	'	7
+8	H	X	(	8	H	X	(	8
+9	I	Y	)	9	I	Y	)	9
+10	J	Z	*	:	J	Z	*	:
+11	K	[	+	;	K	[	+	;
+12	L	\	,	<	L	\	,	<
+13	M	]	–	=	M	]	–	=
+14	N	^	.	>	N	^	.	>
+15	O	←	/	?	O	←	/	?
	NORMAL				INVERSE			


128 144 160 176 192 208 224 240
G Y B R BF CN M O

G – GREEN		BF –	BUFF
Y – YELLOW		CN –	CYAN
B – BLUE		M –	MAGENTA
R – RED		O –	ORANGE

NOTES

Here it is. The ultimate programming resource for your VZ3000. A bumper collection of ideas, tricks, techniques and programs for you and your machine.

The major sections include:

EXPLORING ARTIFICIAL INTELLIGENCE — detailed history, seven major programs to demonstrate AI including BLOCKWORLD and X-SPURT.

GRAPHICS AND SOUND COMPANION — a generous selection of dramatic sound and graphics demonstration programs, including 3-D PRINTER PLOTTER and THE VZ SYNTHESIZER.

PRACTICAL PROGRAMS — Get your VZ300 to earn its keep with our useful MINICALC program.

FORTH — Now you can learn and run the computer language FORTH on your VZ300, without having to buy an extra language; just type in the program and follow the instructions, and your VZ300 will be running in FORTH.

SORTS AND SEARCHES — A number of routines to allow you to test and time the searching and sorting technique for yourself.

PERIPHERALS — We explore the add-ons you can buy for your VZ300, from joysticks to disk drives, and show how they work.

THE AMAZING VZ300 OMNIBUS is designed to give you and your VZ300 months and months of ideas and entertainment.