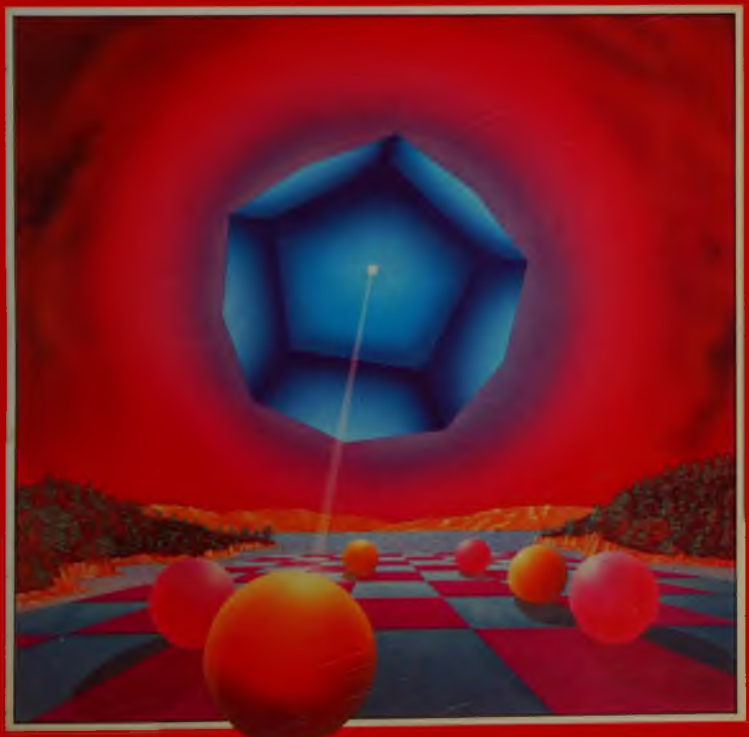
EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR SINCLAIR QL

Tim Hartnell



Interface Publications
LONDON · MELBOURNE

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR SINCLAIR QL

TIM HARTNELL

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR SINCLAIR QL

"The appearance on earth of a nonhuman entity with intelligence approaching or exceeding mankind's would rank with the most significant events in human history"
— Fortune magazine.

This book is part of the 'Interface Artificial Intelligence Library'.

First published in the UK by:
Interface Publications Ltd
9–11 Kensington High Street
LONDON W8 5NP

First printing January 1985

ISBN 0 947695 18 4

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

Cover design – Richard Kelly
Cover art – David John Rowe
Printed and Bound by Short Run Press Ltd, Exeter

CONTENTS

SECTION THREE — TALKING

SECTION FOUR — HELPING

This book is part of the 'Interface Artificial Intelligence Library'.

FOREWORD

You are about to start a fascinating journey, into the realm where science fact interacts with science fiction.

Since the first computers were made, anguished debates have been fought over topics such as:

— Can a machine really think?

— What is the nature of intelligence, and will a machine ever be built which could partake of that nature?

Once you've worked through this book, you will be ready to enter that debate, and enter it with authority. For in this book we are going to investigate the fascinating world of Artificial Intelligence, and are going to replicate some of its most famous programs.

From programs which learn and reason, to those which will talk to you, obey you and advise you, we cover a great deal of ground.

Writing this book, which is part of the 'Interface Artificial Intelligence Library', has been fascinating. Reading through the extensive literature on the subject, becoming acquainted with the aspirations of AI pioneers, and writing programs which — admittedly crudely — allowed some of their findings to be duplicated on a microcomputer has been an extremely interesting and enjoyable exercise.

I hope some of the fascination I've experienced is transmitted in this book, and some of the excitement I've felt watching the programs will be felt by you when you run them.

Tim Hartnell,
London, 1984.

SECTION ONE — THINKING

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

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

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

However, there are two areas of behavior which are both reasonable candidates for classing behavior 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 the 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 three 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 AI has had. Man has been forced to look closely at himself, and to examine areas of human behavior 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 — reason-

ing 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 his thesis that complex behavior, 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 behavior 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 present 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 kinds 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.

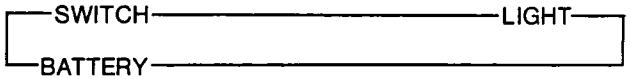
Boole wrote in the preface to his work:

> 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.

SWITCHES AND DECISIONS

We can investigate Boole's claims, and see how they relate to your computer, decision-making and AI, by mentally reconstructing some of the devices that Stibitz built on his kitchen table. We'll start with a very simple circuit, containing a power supply, a single switch, and a light:



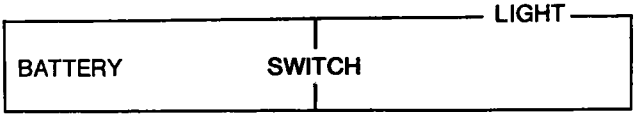
You can see that when the switch is closed, power will flow and the light will light up. We'll indicate that the switch has been turned on by saying that its state is '1'. When the switch is off, and the current does not flow, its state will be said to be '0'. On equals 1, off equals 0. Further we will adopt the convention that when the light is lit, its state is 1; when the light is off, its state is 0.

This is said to be an **ASSERTION** circuit. When the switch is on, the light is on. That is, switch state equals light state. If we draw up a little table to show the relationship between the states of the lamp and the switch in an assertion circuit we would get something like this:

SWITCH	LIGHT
0	0
1	1

A table like this, by the way, is called a 'truth table'.

Now, we'll look at another simple circuit:



If you look at this, you'll see that the light is on (light state equals 1) when the switch is open (switch state is 0) and — once the switch has been closed (switch state set to 1) — the current will flow through it rather than through the light.

This is a **NEGATION** circuit, and the truth table for it looks like this:

SWITCH	LIGHT
0	1
1	0

Now we get to the interesting bits, where circuits can 'make decisions'. Imagine we have a circuit with two switches in it, as follows:

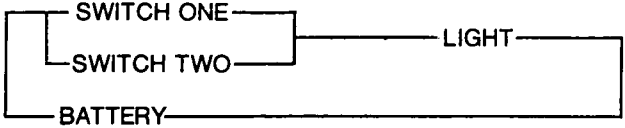
With both switches closed (that is, turned on, with their states **both** equal to 1 [1 1]) the light will glow. If **either** of the switches is off (one switch set to 1, and the other to 0

[1 0]) or both of them are off (switch one set to 0 and switch two equals 0 [0 0]) the light will be off. This is called an **AND** gate circuit.

The truth table looks like this:

SWITCH ONE	SWITCH TWO	LIGHT
0	0	0
1	0	0
0	1	0
1	1	1

From AND we move on to OR. the **OR** gate circuit looks like this:



In this circuit, with the switches in parallel (they were in series in the AND circuit), the light will be on (state 1) if **either** switch one or switch two is one (either [0 1] or [1 0]) or both switches are one [1 1]. Before you read on, try and construct a truth table for the OR gate circuit.

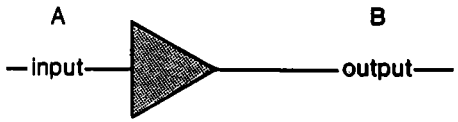
SWITCH ONE	SWITCH TWO	LIGHT
0	0	0
1	0	1
0	1	1
1	1	1

'REAL' LOGIC GATES

Your computer uses logic gates just like these, except of course they are not great big switches which need to be turned off and on. One reason Shannon and Stibitz used relays is because these are switches which can be turned on without actually touching them (when an electrical current is applied, a magnetic force is generated which closes the switch).

There are no electrical relays of Shannon's type in your computer either, although the elements of the chips in your computer act like thousands upon thousands of relays. In schematic diagrams of circuits, the gates we've examined above are shown as follows.

Firstly there is the 'inverter'. If an incoming signal is state 0, it leaves the device as state 1, and vice versa:

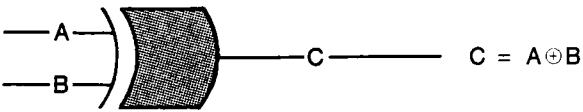


This is an AND gate:

And this the OR gate:

There's another gate which is often used in circuits, and which will be helpful to you when trying to work out how circuits reach decisions. This is **XOR** gate, the EXCLUSIVE OR. With this, if **either** input is 1 ([1 0] or [0 1]) the state of the output is 1. However, if they are **both** 1 [1 1] or **both** 0 [0 0] the output state is 0.

Here's a schematic for an XOR gate:



Here's the truth table for an XOR gate:

SWITCH ONE (A)	SWITCH TWO (B)	LIGHT (C)
0	0	0
0	1	1
1	0	1
1	1	0

With these few elements, you can now construct 'circuits' to reach decisions. You can easily select, for example, from the gates we've examined (and I suggest you try and work out which ones they are) a sequence of gates to represent statements like these:

IF A AND B are true (i.e. [1 1] AND C OR
D (but not both) is true ([1 0] or [0 1]
but not [1 1] and certainly not [0 0])
THEN D is true (i.e. output is 1).

Working out the switch arrangements to mimic the above statement — and others like is — is fascinating, and can give a real insight into the way a simple sequence of Boolean operations can process decisions to arrive at results (if the above circuit was actually built correctly, a light would come on at D). Doing this should help you get a little closer to understanding how your computer works, and from that understanding, you may well find you'll really appreciate how complicated such things can become, and how complex they have to be to simulate any kind of 'intelligent' behavior above the most basic.

Consider, for example, the circuitry required to emulate the action of the computer in the first program in this section, TICTAC.

CHAPTER TWO — A PROGRAM WHICH LEARNS

Many AI programs do not spring into the computer fully formed. Even when they are debugged, and operating, they are far from finished. The program we'll look at in this section of the book, 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 accord 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 center 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 deliberately 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 288 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, and 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 center 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 a call to the procedure **initialise** which is defined as follows:

```
1180 DEFine PROCedure initialise
1182  CLS #0
1184  BORDER 7,7
1186  PAPER 5
1188  INK 1
1190  CLS
1200  DIM a(9): REMark board
1210  DIM m(10): REMark to hold knowledg
e base
1220  DIM w(24): REMark win/block data
1230  DIM d(5): REMark to hold moves in
current game
1240  REMark win/block data
1245  RESTORE
```

```
1250  FOR j=1 TO 24
1260   READ w(j)
1270  END FOR 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  REMark initial knowledge base
1320  FOR j=1 TO 10
1330   READ m(j)
1340  END FOR j
1350  DATA 2,6,8,4,7,3,1,9,5,2
1360 END DEFine initialise
```

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:

```
300 REMark update knowledge base
310  FOR b=1 TO 5
320   FOR j=2 TO 9
330    IF m(j)=d(b): adjust_array_m
340   END FOR j
350  END FOR b
430  PRINT: PRINT
440  PRINT "   This is my updated priori
ty"
450  PRINT: PRINT "    ";
460  FOR j=1 TO 9
470   PRINT !m(j)!
480  END FOR j
```

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

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

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

I said earlier that TICTAC's playing strategy does not come solely from its knowledge base. It also has information on the rows of three which it is trying to build (and which it is trying to prevent its opponent from completing). This is the section of code which looks for a move here, before using the knowledge base:

```
540 DEFine PROCedure machine_move
550   p=CODE('o')
```

```
560  x=0
570  j=1
580  IF a(w(j))=a(w(j+1)) AND a(w(j+2))=
32 AND a(w(j))=p: x=w(j+2): GO TO 750
590  IF a(w(j))=a(w(j+2)) AND a(w(j+1))=
32 AND a(w(j))=p: x=w(j+1): GO TO 750
600  IF a(w(j+1))=a(w(j+2)) AND a(w(j))=
32 AND a(w(j+1))=p: x=w(j): GO TO 750
610  IF j<21: j=j+3: GO TO 580
620  IF p=CODE('o'): p=CODE('x'): GO TO
   570
```

It looks first for a winning move for itself (when P equals the ASCII code of the letter "0") 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:

```
630 REMark if no win/block move found
640 REMark then this next section used
650  j=1
660  IF a(m(j))=32: x=m(j): GO TO 750
670  IF j<10: j=j+1: GO TO 660
```

If this fails to give it a move, it tries numbers at random:

```
680  h=0
690  h=h+1
700  x=RND(1,9): IF a(x)=32: GO TO 750
710  IF h<100: GO TO 690
720  r$="draw": REMark it is a draw
```

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.

```
750 REMark make move
760  a(x)=CODE('o')
770  count=count+1
780  d(count)=x
790  flag=0
800  FOR j=1 TO 9
810   IF a(j)=32: flag=1
820  END FOR j
830  IF flag=0 AND r$="": r$="draw"
840  REMark if all positions full, and r
$ not assigned, it is a draw
850 END DEFine machine_move
```

After each move, human or QL, the following procedure is visited:

```
870 DEFine PROCedure win_check
880  j=1
890  IF a(w(j))=32: j=j+3
900  IF j>23: RETurn
910  IF a(w(j))=a(w(j+1)) AND a(w(j))=a(
w(j+2)): GO TO 940
920  IF j<22: j=j+3: GO TO 890
930 RETurn
940  IF a(w(j))=CODE('o'): r$="win": REM
ark machine wins
950  IF a(w(j))=CODE('x'): r$="lose": RE
Mark machine loses
955  IF j<22: j=j+3: GO TO 890
960 END DEFine win_check
```

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

```
10 REMark ======
11 REMark TICTAC
```

```
12 REMark ======
15 :
20 initialise
30 REPeat whole_game
40  FOR j=1 TO 9
50   a(j)=32
60  END FOR j
70  FOR j=1 TO 5
80   d(j)=0
90  END FOR j
100  count=0
110  r$=""
120  print_board
130  REPeat main_cycle
140   machine_move
150   print_board
160   win_check
170   IF r$<>"": EXIT main_cycle
180   human_move
190   print_board
200   win_check
210   IF r$<>"": EXIT main_cycle
220  END REPeat main_cycle
230 :
240 REMark end of game
250  print_board
260  PRINT: PRINT
270  IF r$="win": PRINT "         I win":
 flag=-1
280  IF r$="lose": PRINT "         You wi
n": flag=1
290  IF r$="draw": PRINT "       It's a d
raw": GO TO 430
300 REMark update knowledge base
310  FOR b=1 TO 5
```

```
320    FOR j=2 TO 9
330      IF m(j)=d(b): adjust_array_m
340    END FOR j
350  END FOR b
430  PRINT: PRINT
440  PRINT "    This is my updated priori
ty"
450  PRINT: PRINT "    ";
460  FOR j=1 TO 9
470    PRINT !m(j)!
480  END FOR j
490  PRINT: PRINT
500  PRINT "    Press ENTER to continue"
510  INPUT a$
520 END REPeat whole_game
523 DEFine PROCedure  adjust_array_m
525  temp = m(j+flag)
527  m(j+flag)=m(j)
529  m(j)=temp
531  j=9
533 END DEFine adjust_array_m
535 :
540 DEFine PROCedure machine_move
550  p=CODE('o')
560  x=0
570  j=1
580  IF a(w(j))=a(w(j+1)) AND a(w(j+2))=
32 AND a(w(j))=p: x=w(j+2): GO TO 750
590  IF a(w(j))=a(w(j+2)) AND a(w(j+1))=
32 AND a(w(j))=p: x=w(j+1): GO TO 750
600  IF a(w(j+1))=a(w(j+2)) AND a(w(j))=
32 AND a(w(j+1))=p: x=w(j): GO TO 750
610  IF j<21: j=j+3: GO TO 580
620  IF p=CODE('o'): p=CODE('x'): GO TO
   570
```

```
630 REMark if no win/block move found
640 REMark then this next section used
650  j=1
660  IF a(m(j))=32: x=m(j): GO TO 750
670  IF j<10: j=j+1: GO TO 660
680  h=0
690  h=h+1
700  x=RND(1,9): IF a(x)=32: GO TO 750
710  IF h<100: GO TO 690
720  r$="draw": REMark it is a draw
730 RETurn
740 :
750 REMark make move
760  a(x)=CODE('o')
770  count=count+1
780  d(count)=x
790  flag=0
800  FOR j=1 TO 9
810   IF a(j)=32: flag=1
820  END FOR j
830  IF flag=0 AND r$="": r$="draw"
840  REMark if all positions full, and r
$ not assigned, it is a draw
850 END DEFine machine_move
860 :
870 DEFine PROCedure win_check
880  j=1
890  IF a(w(j))=32: j=j+3
900  IF j>23: RETurn
910  IF a(w(j))=a(w(j+1)) AND a(w(j))=a(
w(j+2)): GO TO 940
920  IF j<22: j=j+3: GO TO 890
930 RETurn
940  IF a(w(j))=CODE('o'): r$="win": REM
ark machine wins
```

```
950  IF a(w(j))=CODE('x'): r$="lose": RE
Mark machine loses
955  IF j<22: j=j+3: GO TO 890
960 END DEFine win_check
970 :
980 DEFine PROCedure human_move
990  PRINT: PRINT
1000  INPUT "       Enter your move"!huma
n
1020  IF human<1 OR human>9: GO TO 1000
1030  IF a(human)<>32: GO TO 1000
1040  a(human)=CODE('x')
1050 END DEFine human_move
1060 :
1070 DEFine PROCedure print_board
1080  CLS
1090  PRINT: PRINT: PRINT
1100  PRINT "       1 : 2 : 3    ";CHR$(a(
1));" : ";CHR$(a(2));" : ";CHR$(a(3))
1110  PRINT "       ---------    ---------
"
1120  PRINT "       4 : 5 : 6    ";CHR$(a(
4));" : ";CHR$(a(5));" : ";CHR$(a(6))
1130  PRINT "       ---------    ---------
"
1140  PRINT "       7 : 8 : 9    ";CHR$(a(
7));" : ";CHR$(a(8));" : ";CHR$(a(9))
1150  PRINT
1160 END DEFine print_board
1170 :
1180 DEFine PROCedure initialise
1182  CLS #0
1184  BORDER 7,7
1186  PAPER 5
1188  INK 1
```

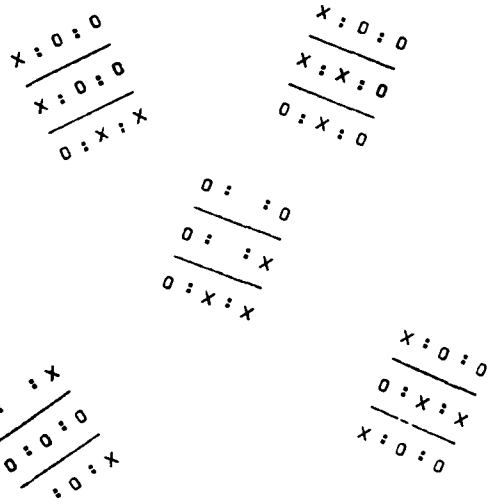
```
1190  CLS
1200  DIM a(9): REMark board
1210  DIM m(10): REMark to hold knowledg
e base
1220  DIM w(24): REMark win/block data
1230  DIM d(5): REMark to hold moves in
current game
1240  REMark win/block data
1245  RESTORE
1250  FOR j=1 TO 24
1260   READ w(j)
1270  END FOR j
1280  DATA 1,2,3,4,5,6,7,8,9
1290  DATA 1,4,7,2,5,8,3,6,9
1300  DATA 1,5,9,3,5,7
1310  REMark initial knowledge base
1320  FOR j=1 TO 10
1330   READ m(j)
1340  END FOR j
1350  DATA 2,6,8,4,7,3,1,9,5,2
1360 END DEFine initialise
1370 :
1380 REMark ==========
1390 REMark END TICTAC
1400 REMark ==========
```

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 DEFine PROCedure random_human_move
4510  h=0
4515  REPeat h_loop
4520   h=h+1
4530   square=RND(1,9)
4540   IF a(square)=32
```

```
4542     a(square)=CODE("x")
4544     RETurn
4546    END IF
4550    IF h>99: EXIT h_loop
4555   END REPeat h_loop
4560   r$="draw"
4570 END DEFine random_human_move
4580 :
4590 REMark ===========
4600 REMark END TICTAC2
4610 REMark ===========
```

To trigger this unintelligent, tireless QL opponent, simply replace line 140 with a call to the procedure.



CHAPTER 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 brackets 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 the 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 (meaning I) 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 on the QL with a multi-dimensional 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 finding 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 multi-dimensional 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.

BACK TO THE PROGRAM

Here's the start of SYLLOGY where the program processes the user input. Line 40 sends action to 910 if a question mark has been entered.

```
10 REMark =======
11 REMark SYLLOGY
12 REMark =======
15 :
20 initialise
23 syllogise
25 :
27 DEFine PROCedure syllogise
30  PRINT: INPUT " ?"!a$
40  IF a$="?": GO TO 910
50  IF a$=""
52   PRINT "End of program"
54   STOP
56  END IF
60  flag=0
70  REMark note there is a space before
the close quote in next 4 lines
```

```
80  IF a$(1 TO 3)="IS ": GO TO 480: REMa
rk conclusions
90  IF a$(1 TO 4)="THE ": a$=a$(5 TO)
100  IF a$(1 TO 3)="AN ": a$=a$(4 TO)
110  IF a$(1 TO 2)="A ": a$=a$(3 TO)
120  x=LEN(a$)
130  n=0
135  REPeat n_loop
140   n=n+1
150   IF a$(n)=" "
152    b$=a$(1 TO n-1)
154    GO TO 180
156    REMark extracts first noun
158   END IF
160   IF n>=x: EXIT n_loop
165  END REPeat n_loop
170  PRINT "I DON'T UNDERSTAND": syllogi
se
180  k=4
190  IF a$(n+1)="W": k=5
200  c$=a$(n+k TO): REMark qualifying ph
rase
210  IF c$(1 TO 2)="A ": c$=c$(3 TO): RE
Mark removes article
220  IF c$(1 TO 3)="AN ": c$=c$(4 TO)
230  IF c$(1 TO 4)="THE ": c$=c$(5 TO)
```

Line 80 detects the 'IS' at the start of the input, indicating that the user is asking SYLLOGY to try and reach a conclusion.

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 center word is 'WAS' ". This allows it to accept phrases such as:

THE DODO WAS A BIG BIRD

...as well as...

TIM IS AN IDIOT

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

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

```
240  REMark store information
250  REMark first check to see if can fi
nd subject before finding blank
260  n=0
265  REPeat n_loop
270   n=n+1
280   IF z$(1,n)=b$: GO TO 320: REMark s
ubject heading exists
```

```
290    IF z$(1,n)="": z$(1,n)=b$: GO TO 3
20
300    IF n>=25: EXIT n_loop
305  END REPeat n_loop
310  PRINT "I HAVE NO MORE SUBJECT STORA
GE ROOM": syllogise
```

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).

```
320  REMark program reaches here with su
bject stored as heading
330  REMark now put object under this
340  k=0
345  REPeat k_loop
350    k=k+1
360    IF z$(k,n)=c$: GO TO 400: REMark i
nformation already stored under that hea
ding
370    IF z$(k,n)="": z$(k,n)=c$: GO TO 4
00
380    IF k>=25: EXIT k_loop
385  END REPeat k_loop
390  PRINT "I HAVE NO MORE OBJECT STORAG
E SPACE": syllogise
400  IF flag=1: PRINT "       > OK": syll
ogise: REMark swap has been done
410  REMark now swap object and subject
and save again
420  flag=1
430  m$=b$
440  b$=c$
450  c$=m$
460  GO TO 250
```

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:

```
480  REMark conclusions
490  REMark first split input
500  a$=a$(4 TO): REMark strip "IS "
510  IF a$(1 TO 2)="A ": a$=a$(3 TO): RE
Mark strip "A " if present
520  IF a$(1 TO 3)="AN ": a$=a$(4 TO): R
EMark strip "AN " if present
525  IF a$(1 TO 4)="THE ": a$=a$(5 TO):
REMark strip "THE " if present
530  REMark get first word - f$
540  x =LEN(a$)
550  n=0
555  REPeat n_loop
560   n=n+1
570   IF a$(n)=" ": f$=a$(1 TO n-1): GO
TO 600
580   IF n>=x: EXIT n_loop
585  END REPeat n_loop
590  PRINT "       > I DO NOT UNDERSTAND"
: syllogise
```

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$. The next section extracts the second word, to set it equal to S$.

```
600  REMark now get second word - s$
610  s$=a$(n+3 TO)
620  IF s$(1)=" ": s$=s$(2 TO): REMark s
trips leading zero if article was "AN"
630  PRINT "          (LOOKING FOR ";f$;"
)"
640  x=0
645  REPeat x_loop
650   x=x+1
660   IF z$(1,x)=f$: PRINT "          (
FOUND AT 1"!x!")": GO TO 700
670   IF x>=25: EXIT x_loop
675  END REPeat x_loop
680  PRINT "      > I CANNOT FIND THE SU
BJECT": PRINT "        ";f$
690  syllogise
```

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.

```
700  y=1
705  REPeat y_loop
710   y=y+1
```

```
720   IF z$(y,x)=s$: PRINT "        > YES"
: syllogise
730   IF y>=25: EXIT y_loop
733  END REPeat y_loop
740  y=1
745  REPeat y_loop
750   y=y+1
760   p$=z$(y,x)
770   m=0
780   m=m+1
790   IF z$(1,m)=p$: GO TO 830
800   IF m<25: GO TO 780
810   IF y>=25: EXIT y_loop
815  END REPeat y_loop
820  PRINT "       > NO": syllogise
830  q=1
835  REPeat q_loop
840   q=q+1
850   IF z$(q,m)=s$: PRINT "        > YES"
: syllogise
860   IF q>=25: EXIT q_loop
865  END REPeat q_loop
870  IF m<25: GO TO 780
880  GO TO 820
```

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.

```
910 REMark check contents of particular
file
920  INPUT "SUBJECT TO CHECK?"!h$
930  t=0
935  REPeat t_loop
940   t=t+1
950   IF z$(1,t)=h$: GO TO 990
960   IF t>=25: EXIT t_loop
965  END REPeat t_loop
970  PRINT "I HAVE NO DATA STORED ON"!h$
980  syllogise
990  k=1
995  REPeat k_loop
1000   k=k+1
1010   IF z$(k,t)<>"": PRINT k!t!z$(k,t)
1020   IF k>=25: EXIT k_loop
1025  END REPeat k_loop
1030  syllogise
```

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

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

```
10 REMark =======
11 REMark SYLLOGY
12 REMark =======
15 :
20 initialise
23 syllogise
25 :
27 DEFine PROCedure syllogise
30  PRINT: INPUT " ?"!a$
40  IF a$="?": GO TO 910
```

```
50  IF a$=""
52   PRINT "End of program"
54   STOP
56  END IF
60  flag=0
70  REMark note there is a space before
the close quote in next 4 lines
80  IF a$(1 TO 3)="IS ": GO TO 480: REMa
rk conclusions
90  IF a$(1 TO 4)="THE ": a$=a$(5 TO)
100  IF a$(1 TO 3)="AN ": a$=a$(4 TO)
110  IF a$(1 TO 2)="A ": a$=a$(3 TO)
120  x=LEN(a$)
130  n=0
135  REPeat n_loop
140   n=n+1
150   IF a$(n)=" "
152    b$=a$(1 TO n-1)
154    GO TO 180
156    REMark extracts first noun
158   END IF
160   IF n>=x: EXIT n_loop
165  END REPeat n_loop
170  PRINT "I DON'T UNDERSTAND": syllogi
se
180  k=4
190  IF a$(n+1)="W": k=5
200  c$=a$(n+k TO): REMark qualifying ph
rase
210  IF c$(1 TO 2)="A ": c$=c$(3 TO): RE
Mark removes article
220  IF c$(1 TO 3)="AN ": c$=c$(4 TO)
230  IF c$(1 TO 4)="THE ": c$=c$(5 TO)
240  REMark store information
250  REMark first check to see if can fi
```

```
nd subject before finding blank
260  n=0
265  REPeat n_loop
270   n=n+1
280   IF z$(1,n)=b$: GO TO 320: REMark s
ubject heading exists
290   IF z$(1,n)="": z$(1,n)=b$: GO TO 3
20
300   IF n>=25: EXIT n_loop
305  END REPeat n_loop
310  PRINT "I HAVE NO MORE SUBJECT STORA
GE ROOM": syllogise
320  REMark program reaches here with su
bject stored as heading
330  REMark now put object under this
340  k=0
345  REPeat k_loop
350   k=k+1
360   IF z$(k,n)=c$: GO TO 400: REMark i
nformation already stored under that hea
ding
370   IF z$(k,n)="": z$(k,n)=c$: GO TO 4
00
380   IF k>=25: EXIT k_loop
385  END REPeat k_loop
390  PRINT "I HAVE NO MORE OBJECT STORAG
E SPACE": syllogise
400  IF flag=1: PRINT "       > OK": syll
ogise: REMark swap has been done
410  REMark now swap object and subject
and save again
420  flag=1
430  m$=b$
440  b$=c$
450  c$=m$
460  GO TO 250
```

```
480  REMark conclusions
490  REMark first split input
500  a$=a$(4 TO): REMark strip "IS "
510  IF a$(1 TO 2)="A ": a$=a$(3 TO): RE
Mark strip "A " if present
520  IF a$(1 TO 3)="AN ": a$=a$(4 TO): R
EMark strip "AN " if present
525  IF a$(1 TO 4)="THE ": a$=a$(5 TO):
REMark strip "THE " if present
530  REMark get first word - f$
540  x =LEN(a$)
550  n=0
555  REPeat n_loop
560   n=n+1
570   IF a$(n)=" ": f$=a$(1 TO n-1): GO
TO 600
580   IF n>=x: EXIT n_loop
585  END REPeat n_loop
590  PRINT "      > I DO NOT UNDERSTAND"
: syllogise
600  REMark now get second word - s$
610  s$=a$(n+3 TO)
620  IF s$(1)=" ": s$=s$(2 TO): REMark s
trips leading zero if article was "AN"
630  PRINT "         (LOOKING FOR ";f$;"
)"
640  x=0
645  REPeat x_loop
650   x=x+1
660   IF z$(1,x)=f$: PRINT "          (
FOUND AT 1"!x!")": GO TO 700
670   IF x>=25: EXIT x_loop
675  END REPeat x_loop
680  PRINT "      > I CANNOT FIND THE SU
BJECT": PRINT "        ";f$
690  syllogise
```

```
700  y=1
705  REPeat y_loop
710   y=y+1
720   IF z$(y,x)=s$: PRINT "       > YES"
: syllogise
730   IF y>=25: EXIT y_loop
733  END REPeat y_loop
740  y=1
745  REPeat y_loop
750   y=y+1
760   p$=z$(y,x)
770   m=0
780   m=m+1
790   IF z$(1,m)=p$: GO TO 830
800   IF m<25: GO TO 780
810   IF y>=25: EXIT y_loop
815  END REPeat y_loop
820  PRINT "       > NO": syllogise
830  q=1
835  REPeat q_loop
840   q=q+1
850   IF z$(q,m)=s$: PRINT "       > YES"
: syllogise
860   IF q>=25: EXIT q_loop
865  END REPeat q_loop
870  IF m<25: GO TO 780
880  GO TO 820
890 :
900 :
910 REMark check contents of particular
file
920  INPUT "SUBJECT TO CHECK?"!h$
930  t=0
935  REPeat t_loop
940   t=t+1
```

```
950   IF z$(1,t)=h$: GO TO 990
960   IF t>=25: EXIT t_loop
965  END REPeat t_loop
970  PRINT "I HAVE NO DATA STORED ON"!h$
980  syllogise
990  k=1
995  REPeat k_loop
1000   k=k+1
1010   IF z$(k,t)<>"": PRINT k!t!z$(k,t)
1020   IF k>=25: EXIT k_loop
1025  END REPeat k_loop
1030  syllogise
1035 END DEFine syllogise
1040 :
1050 DEFine PROCedure initialise
1052  CLS #0
1054  BORDER 4,1
1056  PAPER 0
1058  INK 6
1060  CLS
1080  DIM z$(25,25,50)
1083  PRINT
1085  PRINT"Please engage CAPS LOCK"
1090 END DEFine
1100 :
1110 REMark ===========
1120 REMark END SYLLOGY
1130 REMark ===========
```

SECTION TWO — SEARCHING

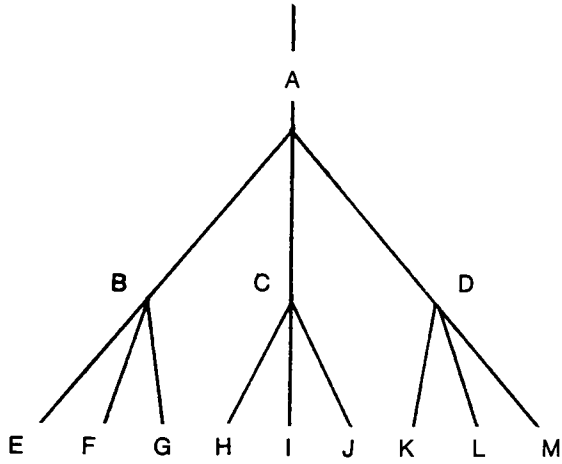
CHAPTER FOUR — SEARCH TREES AND SNICKERS

In this section of the book, we will develop a checkers-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 predetermined 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 BLOCKWORLD), or you are not too clear as to the criteria by which your program could be making choices.

We have developed the checkers-like program, called 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. We'll look at snapshots from a completed game in due course, but for now, we only need to see the first few moves.

Here's what the board looks like at the start of the game:

```
COMPUTER: 0    HUMAN: 0

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

The score (currently zero) for both the machine and the human is printed above the board. Each player starts with eight pieces (as opposed to 12 in checkers). The computer's pieces are at the top of the board (the C's) and the human's are at the bottom (the H's). The computer is playing down the screen, and the human is playing up it.

The dots represent the black squares on a checker-board. The pieces move as in checkers, that is diagonally from black square to black square. Each piece, then, is actually 'sitting on' a dot which will only be revealed when the piece is moved. Each of the dots represents a position to which a piece can be moved.

As I said, each piece moves like checkers' pieces, 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.

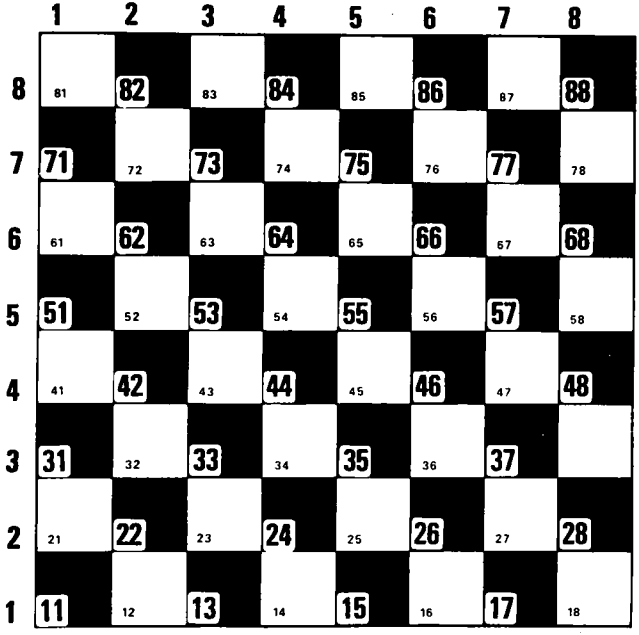
VANISHING ACTS

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 in our sample game in due course. The computer will tell you the 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, shown in the board printed a little earlier, there are seven possible opening moves. The computer finds each legal move, then prints up the moves on the screen, before making the move, as follows (with the numbers themselves being worked out by specifying the number down the edges of the board first, followed by the number across the top or bottom):

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

The numbers printed here by the computer refer to those within a master array which holds the board inside the computer. This is the numbered board 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, than is one in which only the black squares are numbered from one to thirty-two.

A computer 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 64	-9
73 to 62	-11
75 to 66	-9
75 to 64	-11
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 computer can make decisions fairly easily on this board. Assume the square the computer is on, is numbered X. If there is a human piece on X-9, and X-18 is empty, it knows it can capture by leaping into X-18. Its score can then be incremented, and X-9 turned into a blank square.

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

The position after the capture (remember, you as computer 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 earlier, or on a checkers board you have numbered in the same way.

The computer 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 share of games (including the one which is used to demonstrate the program, later in this section). You may well be tempted to think that, if a tree was set up and searched completely, the program may 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.

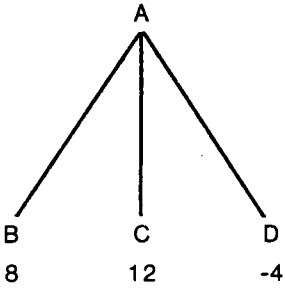
MINI-MAXING

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.

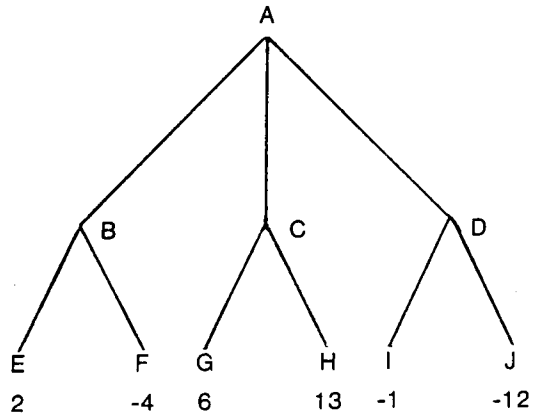
To use this, however, the computer should be able to assign numerical values to the positions it discovers.

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 center the piece will be after the move, if it threatens (immediately, or could do so after another move) an enemy piece, if the square it is considering moving to is under threat, if the move actually makes a capture, or achieves some other goal (such as reaching the opposite back row).

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



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



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

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

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

I said earlier that SNICKERS works by assigning a value to each possible move, in a hierarchy. It chooses its moves in reference to this hierarchy, which puts a value on the possible moves in the following order. It will always make a move which is higher up the tree if it can.

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

For example, the program may see two possible captures,

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

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

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

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

— Other captures.

— Moves to protect pieces under threat.

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

— Non-capture moves onto the back row.

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

— Any legal move.

If it finds any capture moves, it will not bother looking further down the tree. In effect, it automatically prunes branches with 'lower' nodes, by not even considering them. This may seem rash, and certainly means the program is unable to play with any kind of overall strategy, but it works surprisingly well (aided, of course, by the simple nature of the game) in practice, and manages to play with an appearance of skill.

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

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

If this search fails to find a move, our computer looks to see if it has any pieces on the second back row which could be moved onto the back row, adding to its score. It has a predetermined order for doing this, ensuring that if two pieces are on the back row and can be moved, the one closest to the center 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 computer chooses at random from these.

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

Incidentally, you may fee 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 center', 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
BISHOP	— 3
KNIGHT	— 3.5
ROOK	— 5
QUEEN	— 9
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 possibly 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 x 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.

Let us now return to the SNICKERS game. You may recall that we looked at the opening board position, and the computer generated a list of the moves it was considering. The opening moves given by it were:

71 to 62
73 to 64
73 to 62
75 to 66
75 to 64
77 to 68
77 to 66

All of these moves are determined by it to be 'safe, non-capture' moves, and of equal worth. Therefore, it chooses at random from among them, and makes the move 71 to 62 as you can see:

```
COMPUTER: 0    HUMAN: 0

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

In fact, the moves may not all be of equal worth, as there is value in developing the center of the field whenever possible, but the program has been given no information upon which to make this assessment, so it believes (and acts on the belief) that all the moves are equal.

The human response is then entered, using the number down the side and across the bottom of the 'from' square (entered as a single number) and then, when prompted, entering the 'to' square in the same way:

```
MOVE FROM? 24
      TO? 35
```

The board is reprinted, and the computer reveals the moves it is considering:

```
COMPUTER: 0    HUMAN: 0

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

I AM CONSIDERING 73 TO 64
I AM CONSIDERING 75 TO 66
I AM CONSIDERING 75 TO 64
```

```
I AM CONSIDERING 77 TO 68
I AM CONSIDERING 77 TO 66
I AM CONSIDERING 62 TO 53
I AM CONSIDERING 62 TO 51
```

A few moves later, the board looks like this (not all the possible computer moves are shown in this printout):

```
COMPUTER: 0    HUMAN: 0

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

I AM CONSIDERING 82 TO 73
I AM CONSIDERING 82 TO 71
I AM CONSIDERING 64 TO 55
I AM CONSIDERING 64 TO 53
```

There are no captures the computer can make, and all the possible moves can be classed as 'good, safe'. The computer is still playing by choosing at random from among the equally-good (in its own eyes) moves it has discovered and stored:

```
COMPUTER: 0    HUMAN: 0

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

MOVE FROM? 37
       TO? 48


COMPUTER: 0    HUMAN: 0

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

The position now becomes a little more complex. The opponents' pieces are getting close to each other on the board, so the possibility of moving into danger now exists. For example, if the computer were to move 66 to 57, the human would probably respond by capturing the piece on 57, using the piece currently on 48. The program's search mechanism reveals this. Remember, it is at all times trying to ensure that its score after the move is as good as possible, and the human is not given the chance to increase his or her score. A move from 66 to 57 would work against both these aims, so it would be a poor program which proceeded — given all the alternative moves which currently exist — and made the 66 to 57 move.

Fortunately, SNICKERS can see such elementary dangers, and moves from 73 to 62 which is a move classed as 'good, safe'.

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

MOVE FROM? 17
      TO? 26
```

```
COMPUTER: 0    HUMAN: 0

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

MOVE FROM? 28
        TO? 37
```

And so the game develops:

```
COMPUTER: 0    HUMAN: 0

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

```
COMPUTER: 0    HUMAN: 0

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

MOVE FROM? 24
       TO? 35
```

There are now very few easy ('good, safe') moves available to the program, and it makes a move which appears to be its first blunder, moving from 55 into danger (from 33) into 44. The human responds by making the capture, and the program offers congratulations.

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

MOVE FROM? 33
       TO? 55   WELL DONE
```

Of course, the human has now moved into danger, a fact which SNICKERS is quick to appreciate. It reports the move it has found, and is about to make:

```
COMPUTER: 0    HUMAN: 1

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


66 TO 44 CAPTURING ON 55

        >> CAPTURE FOUND
```

The score is now one all. Remember, you gain points by capturing an opponent piece, and also by getting one of your own pieces onto the opposing back row. We will see that second method of gaining points in action shortly.

Fortunately, the capture by the program has not placed it in a position to be captured in turn:

```
COMPUTER: 1    HUMAN: 1

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

MOVE FROM? 48
        TO? 57
```

The program knows that it can move right up next to an enemy piece, provided that it has backup which prevents the human from capturing it, so it moves from 51 to 42:

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

MOVE FROM? 37      TO? 48
```

A few moments later, a wealth of choices is facing the program:

```
COMPUTER: 1   HUMAN: 1

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


 44 TO 26 CAPTURING ON 35
I AM CONSIDERING 44 TO 53
 33 TO 11 CAPTURING ON 22

      >> CAPTURE FOUND
I CAPTURED AND LANDED ON 11 ON BACK ROW
```

SNICKERS has chosen the best move possible on the board, capturing an enemy piece and ending up on the back row (on 11), thus gaining two points (and vanishing from the board).

The choices facing the human player are not very palatable. The piece on 35 is under threat from 44 and there seems no way to avoid that danger. 44 cannot be

captured, and 35 cannot be moved. The human ignores the threat as nothing can be done about it, and works towards getting a piece closer to the opposing back row (and imposing a weak threat — because the human piece is unprotected — on the computer piece on 75):

```
COMPUTER: 3    HUMAN: 1

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

MOVE FROM? 57
       TO? 66
```

SNICKERS decides not to capture the new piece on 66, but instead goes for the one on 35, jumping over it into 26. The human then tells the computer that the capture of the piece on 75 will be made, so the computer offers congratulations:

```
COMPUTER: 4    HUMAN: 1

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

MOVE FROM? 66
       TO? 84
WELL DONE
```

The human, of course, has gained two points by this jump, ending up on the back row. The computer now has four points (one less than that needed for victory) and the human has three:

```
COMPUTER: 4    HUMAN: 3

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

One of the priorities within SNICKERS' hierarchy of moves is that of moving onto the back row if possible. It is now possible, so SNICKERS makes that move.

```
I'M MOVING ONTO BACK ROW FROM 26
```

This, of course, gives the game to the program:

```
COMPUTER: 5    HUMAN: 3

    12345678
    --------
8   . . . C 8
7 . . . . . 7
6   . C . C 6
5 . C . .   5
4   . . H H 4
3 H . . H   3
2   . . . . 2
1 . . . . . 1
    --------
    12345678
THE GAME IS OVER          I'M THE WINNER
```

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 procedure calls.

```
 10 REMark ========
 11 REMark SNICKERS
 12 REMark ========
 15 :
 20 initialise
 30 print_board
 35 :
 40 REPeat main_cycle
 50  computer_moves
 60  print_board
 70  IF cs>4: EXIT main_cycle
 80  human_move
 90  print_board
100  IF hs>4: EXIT main_cycle
105 END REPeat main_cycle
110 :
```

As you can see from looking at the major loop for SNICKERS this makes it very simple to understand the program's construction. As well, it makes it easy to track down errors. If, for example, the program was not printing the board correctly, it would make sense to look first in the procedure which prints out the board.

The action first goes to the INITIALISE routine, from line 2070:

```
2070 DEFine PROCedure initialise
2072  CLS #2
2074  WINDOW 310,210,70,12
2076  CLS #0
2080  BORDER 1,1
2085  PAPER 6
2090  CLS
2095  INK 2
2100  RANDOMISE
2110  DIM a(110): REMark board and blank
 spaces around and beyond it
2120  DIM g(3): REMark good, safe captur
e store
2130  DIM s(3): REMark safe capture stor
e
2140  DIM t(18): REMark other capture st
ore, also used for safe non-capture move
s
2150  e=CODE(" "): REMark empty "white"
square
2160  b=CODE("."): REMark empty "black"
square
2170  c=CODE("C"): REMark computer piece
2180  h=CODE("H"): REMark human piece
2190  hs=0: REMark human score
2200  cs=0: REMark computer score
2210 REMark set up starting board
2215  RESTORE
2220  FOR j=10 TO 80 STEP 10
2230   FOR k=1 TO 8
2240    READ x: a(j+k)=x
2250   END FOR k
2260  END FOR j
2270 END DEFine initialise
2275 :
```

```
2280 REMark ====
2282 REMark data
2284 REMark ====
2286 :
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
```

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. The REM statements explain them:

```
180 REMark ==========
184 REMark definitions
186 REMark ==========
188 :
190 DEFine PROCedure computer_moves
210  REMark search for captures
220  gsafe=0: REMark to count good, safe
 captures which threaten human pieces
230  csafe=0: REMark to count safe captu
res which do not place computer under th
reat
240  ccapture=0: REMark to count other c
aptures found
```

The 'stores' are emptied:

```
250  FOR j=1 TO 3
260   g(j)=0: REMark empty good, safe ca
pture store
270   s(j)=0: REMark empty safe capture
store
280   t(j)=0: REMark empty other capture
```

```
 store
290  END FOR j
```

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:

```
300  FOR j=80 TO 30 STEP -10
310   FOR k=1 TO 8
320    IF a(j+k)<>c: GO TO 390: REMark s
kip evaluation if no computer piece here
330    REMark 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: GO SUB 700
: REMark capture found
360    REMark 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: GO SUB 700
: REMark capture found
390   END FOR k
400  END FOR j
410  IF gsafe+csafe+ccapture=0: GO TO 98
0: REMark no captures found
420  REMark now choose capture to make
430   PRINT: PRINT "          >> Capture f
ound"
440   PAUSE 100
450   IF gsafe<>0: GO TO 500
460   IF csafe<>0: GO TO 670
470  REMark choose from general captures
480   mov=t(RND(1,ccapture))
```

```
490   GO TO 540
500  REMark choose from good safe
510  REMark select from stored moves
520   mov=g(RND(1,gsafe))
530  REMark make move
540   start=INT(mov/100)
550   ed=mov-100*start
560   a(start)=b
570   a(start-ed)=b
580   a(start-2*ed)=c
590   cs=cs+1
600  REMark check for additional score i
f landing on back row
610    IF start-2*ed>18: RETurn
620    a(start-2*ed)=b
630    cs=cs+1
640    PRINT "I captured and landed on"!s
tart-2*ed!"on back row"
650    PAUSE 200
660   RETurn
670   REMark safe capture
680    mov=s(RND(1,csafe))
690    GO TO 540
700   REMark check proposed capture for s
afety
710   REMark check square below in same d
irection as intended move
720    PRINT j+k!"to"!y!"capturing on"!x
730    PAUSE 90
740    IF a(z)=h: GO TO 920: REMark store
 as a non-safe capture
750    REMark check square in other direc
tion from intended move
760    IF a(y+m)=h AND a(y-m)=b: GO TO 92
0
```

```
770  REMark now check to see if will lea
ve a piece exposed by making move
780    IF a(j+k+m)=c AND a(j+k+2*m)=h: GO
 TO 920
790  REMark if reached this point then c
apture is "safe"
800  REMark store this move
810    csafe=csafe+1
820    s(csafe)=100*(j+k)+20+m: REMark th
is encodes enough info to recreate move
830  REMark now see if this deserves to
be called a "good, safe" capture
840    check =gsafe
850    IF y+2*m<1: RETurn
860    IF a(y+m)=h AND a(y-(20+m))<>b AND
 a(y+2*m)=b: gsafe=gsafe+1
870    IF check=gsafe: RETurn : REMark th
is move found not to be "good safe"
880  REMark 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  REMark 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
```

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, which is called **ed** in the program.

The four-digit number is decoded by the QL, and the move made, using the lines from 1510:

```
1510  start=INT(mov/100)
1520  ed=mov-100*start
1530  a(start)=b
1540  a(ed)=c
1550 END DEFine computer_moves
```

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.

```
980  REMark move to protect piece under
threat
990    mov=0
1000    j=80
1010    k=1
1020    q=j+k
1030    IF a(q)<>c: GO TO 1110: REMark do
 not consider this square, no computer p
iece
1040    IF a(q+9)=b AND a(q-9)=h AND a(q+
18)=c: mov=100*(q+18)+q+9
1050    REMark random rejection of this m
ove if it exposes a subsequent one
1060    IF mov<>0 AND a(q-2)=h AND a(q+20
)=b AND RND>.5: GO TO 1510
```

```
1070    IF a(q+9)=b AND a(q-9)=h AND a(q+
20)=c THEN mov=100*(q+20)+q+9: GO TO 151
0
1080    IF a(q+11)=b AND a(q-11)=h AND a(
q+22)=c: mov=100*(q+22)+q+11
1090    IF mov<>0 AND a(q+2)=h AND a(q+22
)=b AND RND>.5: GO TO 1510
1100    IF a(q+11)=b AND a(q-11)=h AND a(
q+20)=c: mov=100*(q+20)+q+11: GO TO 1510
1110    IF k<8: k=k+1: GO TO 1020
1120    IF j>10: j=j-10: GO TO 1010
```

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. I explained earlier how 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:

```
1140  REMark no capture found, so look f
or move to "disappear" on back row
1150  mov=0
1160 REMark undesirable moves checked fi
rst, so can be overwritten by better one
s
```

```
1170  IF a(22)=c AND a(11)=b: mov=22
1180  IF a(28)=c AND a(17)=b: mov=28
1190  IF a(22)=c AND a(13)=b: mov=22
1200  IF a(26)=c AND a(17)=b: mov=26
1210  IF a(26)=c AND a(15)=b: mov=26
1220  IF a(24)=c AND a(15)=b: mov=24
1230  IF a(24)=c AND a(13)=b: mov=24
1240  IF mov=0: GO TO 1310
1250  PRINT: PRINT "I'm moving onto back
 row from"!mov
1260  PAUSE 200
1270  a(mov)=b
1280  cs=cs+1
1290 RETurn
```

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:

```
1310 REMark safe, non-capture moves
1320  cmove=0: REMark count moves forwar
d
1330  FOR j=80 TO 30 STEP -10
1340   FOR k=1 TO 8
1350    IF a(j+k)<>c: GO TO 1460
1360    x=j+k-9: y=j+k-18: z=j+k-20
1370    q=j+k+2
1380    IF a(x)<>b: GO TO 1460
1390    IF a(y)=h OR a(z)=h AND a(q)=b:
GO TO 1460
1400    store_moves
1410    x=j+k-11: y=j+k-22: z=j+k-20
1420    q=j+k-2
1430    IF a(x)<>b: GO TO 1460
```

```
1440     IF a(y)=h OR a(z)=h AND a(q)=b:
GO TO 1460
1450     store_moves
1460    END FOR k
1470   END FOR j
1480   IF cmove=0: GO TO 1630
1490 REMark make move
1500   mov=t(RND(1,cmove))
```

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:

```
1630 REMark random non-capture move
1640   PRINT "Looking for random, legal m
ove"
1645   PAUSE 30
1650   l=0
1655   REPeat l_loop
1660     l=l+1
1670     j=10*(RND(1,8))
1680     k=RND(1,8)
1690     IF a(j+k)=c: GO TO 1720
1700     IF l>=200: EXIT l_loop
1705   END REPeat l_loop
1710   PRINT: PRINT "I concede the game":
 STOP
1720   IF a(j+k-9)=b: mov=100*(j+k)+j+k-9
: GO TO 1510
1730   IF a(j+k-11)=b: mov=100*(j+k)+j+k-
11: GO TO 1510
1740   GO TO 1700
```

Here's the complete listing of SNICKERS:

```
10 REMark ========
11 REMark SNICKERS
12 REMark ========
15 :
20 initialise
30 print_board
35 :
40 REPeat main_cycle
50  computer_moves
60  print_board
70  IF cs>4: EXIT main_cycle
80  human_move
90  print_board
100  IF hs>4: EXIT main_cycle
105 END REPeat main_cycle
110 :
120 REMark end of game
130  PRINT: PRINT "The game is over"
140  PRINT
150  IF hs>cs: PRINT "You have won"
160  IF cs>hs: PRINT "I'm the winner"
170 STOP
175 :
180 REMark ===========
184 REMark definitions
186 REMark ===========
188 :
190 DEFine PROCedure computer_moves
210  REMark search for captures
220  gsafe=0: REMark to count good, safe
 captures which threaten human pieces
230  csafe=0: REMark to count safe captu
res which do not place computer under th
reat
```

```
240  ccapture=0: REMark to count other c
aptures found
250  FOR j=1 TO 3
260   g(j)=0: REMark empty good, safe ca
pture store
270   s(j)=0: REMark empty safe capture
store
280   t(j)=0: REMark empty other capture
 store
290  END FOR j
300  FOR j=80 TO 30 STEP -10
310   FOR k=1 TO 8
320    IF a(j+k)<>c: GO TO 390: REMark s
kip evaluation if no computer piece here
330    REMark 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: GO SUB 700
: REMark capture found
360    REMark 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: GO SUB 700
: REMark capture found
390   END FOR k
400  END FOR j
410  IF gsafe+csafe+ccapture=0: GO TO 98
0: REMark no captures found
420  REMark now choose capture to make
430   PRINT: PRINT "         >> Capture f
ound"
440   PAUSE 100
450   IF gsafe<>0: GO TO 500
460   IF csafe<>0: GO TO 670
470  REMark choose from general captures
```

```
480   mov=t(RND(1,ccapture))
490   GO TO 540
500  REMark choose from good safe
510  REMark select from stored moves
520   mov=g(RND(1,gsafe))
530  REMark make move
540   start=INT(mov/100)
550   ed=mov-100*start
560   a(start)=b
570   a(start-ed)=b
580   a(start-2*ed)=c
590   cs=cs+1
600  REMark check for additional score i
f landing on back row
610   IF start-2*ed>18: RETurn
620   a(start-2*ed)=b
630   cs=cs+1
640   PRINT "I captured and landed on"!s
tart-2*ed!"on back row"
650   PAUSE 200
660  RETurn
670  REMark safe capture
680   mov=s(RND(1,csafe))
690   GO TO 540
700  REMark check proposed capture for s
afety
710  REMark check square below in same d
irection as intended move
720   PRINT j+k!"to"!y!"capturing on"!x
730   PAUSE 90
740   IF a(z)=h: GO TO 920: REMark store
 as a non-safe capture
750   REMark check square in other direc
tion from intended move
760   IF a(y+m)=h AND a(y-m)=b: GO TO 92
0
```

```
770  REMark now check to see if will lea
ve a piece exposed by making move
780    IF a(j+k+m)=c AND a(j+k+2*m)=h: GO
 TO 920
790  REMark if reached this point then c
apture is "safe"
800  REMark store this move
810    csafe=csafe+1
820    s(csafe)=100*(j+k)+20+m: REMark th
is encodes enough info to recreate move
830  REMark now see if this deserves to
be called a "good, safe" capture
840    check =gsafe
850    IF y+2*m<1: RETurn
860    IF a(y+m)=h AND a(y-(20+m))<>b AND
 a(y+2*m)=b: gsafe=gsafe+1
870    IF check=gsafe: RETurn : REMark th
is move found not to be "good safe"
880  REMark 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  REMark 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
980  REMark move to protect piece under
threat
990    mov=0
1000    j=80
1010    k=1
1020    q=j+k
```

```
1030   IF a(q)<>c: GO TO 1110: REMark do
 not consider this square, no computer p
iece
1040   IF a(q+9)=b AND a(q-9)=h AND a(q+
18)=c: mov=100*(q+18)+q+9
1050   REMark random rejection of this m
ove if it exposes a subsequent one
1060   IF mov<>0 AND a(q-2)=h AND a(q+20
)=b AND RND>.5: GO TO 1510
1070   IF a(q+9)=b AND a(q-9)=h AND a(q+
20)=c THEN mov=100*(q+20)+q+9: GO TO 151
0
1080   IF a(q+11)=b AND a(q-11)=h AND a(
q+22)=c: mov=100*(q+22)+q+11
1090   IF mov<>0 AND a(q+2)=h AND a(q+22
)=b AND RND>.5: GO TO 1510
1100   IF a(q+11)=b AND a(q-11)=h AND a(
q+20)=c: mov=100*(q+20)+q+11: GO TO 1510
1110   IF k<8: k=k+1: GO TO 1020
1120   IF j>10: j=j-10: GO TO 1010
1140  REMark no capture found, so look f
or move to "disappear" on back row
1150  mov=0
1160 REMark undesirable moves checked fi
rst, so can be overwritten by better one
s
1170  IF a(22)=c AND a(11)=b: mov=22
1180  IF a(28)=c AND a(17)=b: mov=28
1190  IF a(22)=c AND a(13)=b: mov=22
1200  IF a(26)=c AND a(17)=b: mov=26
1210  IF a(26)=c AND a(15)=b: mov=26
1220  IF a(24)=c AND a(15)=b: mov=24
1230  IF a(24)=c AND a(13)=b: mov=24
1240  IF mov=0: GO TO 1310
1250  PRINT: PRINT "I'm moving onto back
```

```
 row from"!mov
1260  PAUSE 200
1270  a(mov)=b
1280  cs=cs+1
1290 RETurn
1310 REMark safe, non-capture moves
1320  cmove=0: REMark count moves forwar
d
1330  FOR j=80 TO 30 STEP -10
1340   FOR k=1 TO 8
1350    IF a(j+k)<>c: GO TO 1460
1360    x=j+k-9: y=j+k-18: z=j+k-20
1370    q=j+k+2
1380    IF a(x)<>b: GO TO 1460
1390    IF a(y)=h OR a(z)=h AND a(q)=b:
GO TO 1460
1400    store_moves
1410    x=j+k-11: y=j+k-22: z=j+k-20
1420    q=j+k-2
1430    IF a(x)<>b: GO TO 1460
1440    IF a(y)=h OR a(z)=h AND a(q)=b:
GO TO 1460
1450    store_moves
1460   END FOR k
1470  END FOR j
1480  IF cmove=0: GO TO 1630
1490 REMark make move
1500  mov=t(RND(1,cmove))
1510  start=INT(mov/100)
1520  ed=mov-100*start
1530  a(start)=b
1540  a(ed)=c
1550 END DEFine computer_moves
1555 :
1560 DEFine PROCedure store_moves
```

```
1570  cmove=cmove+1
1580  PRINT "I am considering"!j+k!"to"!
x
1590  PAUSE 80
1600  t(cmove)=100*(j+k)+x
1610 END DEFine store_moves
1620 :
1630 REMark random non-capture move
1640  PRINT "Looking for random, legal m
ove"
1645  PAUSE 30
1650  l=0
1655  REPeat l_loop
1660   l=l+1
1670   j=10*(RND(1,8))
1680   k=RND(1,8)
1690   IF a(j+k)=c: GO TO 1720
1700   IF l>=200: EXIT l_loop
1705  END REPeat l_loop
1710  PRINT: PRINT "I concede the game":
 STOP
1720  IF a(j+k-9)=b: mov=100*(j+k)+j+k-9
: GO TO 1510
1730  IF a(j+k-11)=b: mov=100*(j+k)+j+k-
11: GO TO 1510
1740  GO TO 1700
1750 :
1760 DEFine PROCedure print_board
1770  CLS
1780  PRINT
1790  PRINT "COMPUTER:"!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    END FOR k
1880    PRINT j/10
1890   END FOR j
1900   PRINT "       --------"
1910   PRINT "       12345678"
1920   PRINT
1930 END DEFine print_board
1940 :
1950 DEFine PROCedure human_move
1955  REPeat from_loop
1960   INPUT "Move from"!start
1965   IF a(start)>=h: EXIT from_loop
1970  END REPeat from_loop
1975  REPeat to_loop
1980   INPUT "         to"!ed
1982   IF a(ed)<>b OR ABS(start-ed)>11 A
ND a((start+ed)/2)<>c
1984    PRINT "Illegal move!"
1986   ELSE
1988    EXIT to_loop
1990   END IF
1995  END REPeat to_loop
2000  a(start)=b
2010  a(ed)=h
2020  IF ABS(start-ed)>11: a((start+ed)/
2)=b: hs=hs+1: PRINT "Well done"
2030  IF ed>80: a(ed)=b: hs=hs+1: PRINT
"That's one more for you"
2040  PAUSE 70
2050 END DEFine human_move
2060 :
2070 DEFine PROCedure initialise
2072  CLS #2
```

```
2074  WINDOW 310,210,70,12
2076  CLS #0
2080  BORDER 1,1
2085  PAPER 6
2090  CLS
2095  INK 2
2100  RANDOMISE
2110  DIM a(110): REMark board and blank
 spaces around and beyond it
2120  DIM g(3): REMark good, safe captur
e store
2130  DIM s(3): REMark safe capture stor
e
2140  DIM t(18): REMark other capture st
ore, also used for safe non-capture move
s
2150  e=CODE(" "): REMark empty "white"
square
2160  b=CODE("."): REMark empty "black"
square
2170  c=CODE("C"): REMark computer piece
2180  h=CODE("H"): REMark human piece
2190  hs=0: REMark human score
2200  cs=0: REMark computer score
2210 REMark set up starting board
2215  RESTORE
2220  FOR j=10 TO 80 STEP 10
2230   FOR k=1 TO 8
2240    READ x: a(j+k)=x
2250   END FOR k
2260  END FOR j
2270 END DEFine initialise
2275 :
2280 REMark ====
2282 REMark data
2284 REMark ====
```

```
2286 :
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
2370 :
2380 REMark ============
2390 REMark END SNICKERS
2400 REMark ============
```

CHAPTER FIVE — THE WIDER VALUE OF GAMES

It was argued, back 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 about 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 Laboratories, 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 where 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 deduction.

Those who sneered at attention being put into making game-playing machines missed the point. Any advance in AI expertise is potentially a source of information which will assist in other areas of AI application. In the LEARNING AND REASONING section of this book we had 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, is 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.

In the section LEARNING AND REASONING 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.

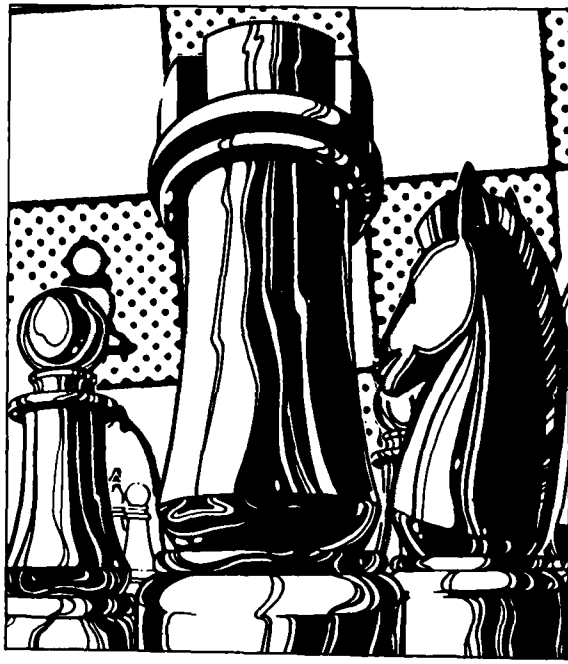
Samuel says he thought of checkers because he knew other groups were working on chess. In comparison with chess, he regarded checkers as a trivial game. But, as you can see from the LEARNING AND REASONING section of this book, even programming a computer to play Noughts and Crosses has its own difficulties.

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 least 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.

Certainly, if you want a real challenge, you could perhaps tackle at least some aspects of the game. Instead of the traditional 19 by 19 board, you could write a program for, say, a seven by seven board, or limit the possible moves. Just as Othello was created by adding an opening move limitation to Reversi, you could create a kind of 'mini-Go' with a small board, and some sort of restrictions on play.

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).



SECTION THREE — TALKING

CHAPTER 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 (and the implementation in this volume, one of the most complete ever published in BASIC, is called DOCTOR).

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

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

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

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

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

LANGUAGE PARSING

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

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

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

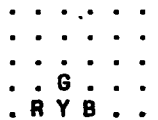
Assume you run a mining company. You have a computer program which will assist you in searching out precious minerals (at least one such program, PROSPECTOR, does exist). You would like to be able to talk to it in the words and phrases which are generally used by you when 'talking mining' with your colleagues.

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

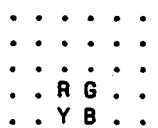
You'll discover, in this section of the book, that the only convincing demonstrations of 'natural language communication' we can give are for extremely restricted 'world views'. In BLOCKWORLD, for example, the world consists of a two-dimensional space, within which your computer manipulates four colored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

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

The program has, as I said, four colored blocks to manipulate. It can tell you where they are, by finding a specified block or by describing the whole scene, and can move them around. in the sample run which precedes the program listing, there is a green block on top of the yellow one. I asked the computer to put the red block on top of the yellow one. This meant it first had to clear the top of the green one — in order to expose the yellow one — before locating the red block and putting it on top of the yellow. Here's the program output (with the computer's speech in upper case):




```
Put the red block on the yellow one

                I UNDERSTAND
        NOW I'LL MOVE THE GREEN ONE

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



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 Joseph Weizenbaum's ELIZA (which we will be looking at in depth in due course), while they gave convincing impressions of intelligent behavior, didn't relly get us any closer to 'real' intelligence (whatever we assume that actually is).

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

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

PROBLEMS

There are a number of major problems with which AI researchers are grappling, 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, although I'm inclined to think it is 'the man' rather than Peter who is tired.

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 patriotic 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 a 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 syntactically (with each syntactic element of the structure bound within parentheses) as follows:

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

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

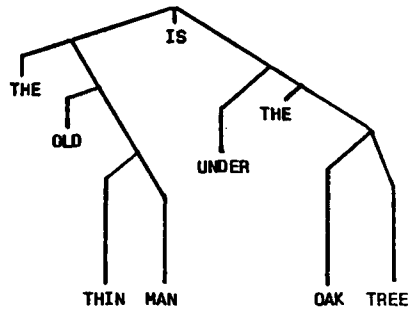
Look at this carefully, following the binding, and you may get a reasonable impression of the various elements which are thus bound together. For example, the words THIN MAN

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 and 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'.

CHAPTER 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 colored 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:

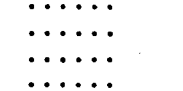
You are looking at the blocks from the 'front'. The BLOCKWORLD is essentially two-dimensional. Although you can move blocks around, and put them on top of each other, you cannot put blocks 'behind' or 'in front of' other blocks. The dots are invisible to the computer. They exist only for your benefit,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

Although you have only a limited number of sentences you can use when communicating with the BLOCKWORLD, you'll be pleased to see how convincing that conversation can be. As well, it makes an easy-to-explain (and impressive) demonstration of artificial intelligence in action to show your sceptical friends.

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

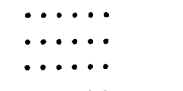
```
? TELL ME WHAT YOU CAN SEE

STARTING FROM THE RIGHT


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

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




```
? WHERE IS THE YELLOW BLOCK

        > LET ME SEE NOW <

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

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


THERE IS NOTHING ABOVE THAT
```

Of course, straight reporting is not too big a deal, even though — as you can see — the program is responding to natural language questions, and replying in a reasonable version of English. We now get to the meat of the matter, getting the computer to manipulate the elements of its restricted domain:



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

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


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


    ? WHERE IS THE YELLOW BLOCK

            > LET ME SEE NOW <

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

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


      ABOVE IT IS THE GREEN BLOCK

      THERE IS NOTHING ABOVE THAT
```

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

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

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

    I UNDERSTAND
NOW I'LL MOVE THE GREEN ONE

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

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

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

```
? TELL ME WHAT YOU SEE

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

```
I SEE THE BLUE BLOCK
...AND BELOW IT...
I SEE THE GREEN BLOCK
FINALLY ...A SPACE
```

We now make the strongest demand to date:

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

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


   >>>> PRESS 'RETURN' TO CONTINUE
?
```

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

HOW DOES IT WORK?

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

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

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

```
Pick up the big red block.
 >OK
Find a block which is taller than the one
   you are holding, and put it into the
   box.
 >OK
What does the box contain?
 >THE BLUE PYRAMID AND THE BLUE BOX
What is the pyramid supported by?
 >THE BOX
Is there a large block behind the
   pyramid?
 >YES, THREE OF THEM: A LARGE RED ONE, A
   LARGE GREEN CUBE, AND THE BLUE ONE
```

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

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

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

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

```
? PUT THE YELLOW BLOCK ON THE BLUE

    I UNDERSTAND
I MUST MOVE THE BLUE BLOCK
I'M MOVING IT TO ROW 4
I'M NOW MOVING THE YELLOW ONE
  ONTO THE BLUE BLOCK

......
......
......
...Y..
.G.BR.
```

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

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

    I UNDERSTAND

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

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

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

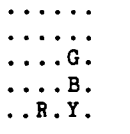
```
? PUT THE BLUE BLOCK ON THE GREEN ONE

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

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

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

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

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

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

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

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

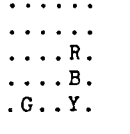
```
? PUT THE GREEN BLOCK ON THE BLUE ONE

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



? PUT THE RED BLOCK ON THE BLUE ONE

I UNDERSTAND
NOW I'LL MOVE THE GREEN ONE

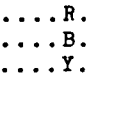
I'M MOVING IT TO ROW 2
I'M NOW MOVING THE RED ONE
ONTO THE BLUE BLOCK



? PUT THE GREEN BLOCK ON THE RED ONE

I UNDERSTAND

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

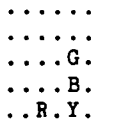


There are now two little towers, one with the blue block on top of the green one, and the other with the yellow block on top of the red one. The computer will be asked to manipulate both the blocks which are currently under other blocks. As you can see, we are putting BLOCKWORLD through a series of tests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

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

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

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

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

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

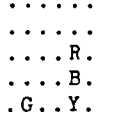
```
? PUT THE GREEN BLOCK ON THE BLUE ONE

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



? PUT THE RED BLOCK ON THE BLUE ONE

I UNDERSTAND
NOW I'LL MOVE THE GREEN ONE

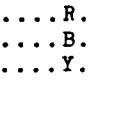
I'M MOVING IT TO ROW 2
I'M NOW MOVING THE RED ONE
ONTO THE BLUE BLOCK



? PUT THE GREEN BLOCK ON THE RED ONE

I UNDERSTAND

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



MODULES OF THE PROGRAM

As with the other programs in this book, BLOCKWORLD starts with a call to a QL procedure at the end of the program which initialises the variables used:

```
2470 DEFine PROCedure initialise
2472  BORDER 2,1
2474  PAPER 85
2476  INK 1
2480  CLS: CLS #0
2490  RANDOMISE
2500  DIM a(5,6)
2510  FOR x=1 TO 5
2520   FOR y=1 TO 6
2530    a(x,y)=46
2540   END FOR y
2550  END FOR x
2560  a(1,2)=CODE("R"): REMark red block
2570  a(1,3)=CODE("Y"): REMark yellow
2580  a(1,4)=CODE("B"): REMark blue
2590  a(1,5)=CODE("G"): REMark green
2600 END DEFine initialise
```

As you can see (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 color ('R' for red, and so on) to the block of that color. There is nothing very complex in this procedure.

Although the initialisation procedure is called just once per run, another procedure which relates to the color of the block, is called every time the QL refers to a block.

```
2400 DEFine PROCedure color_name
2405  SELect ON q
2410    =CODE("R"): PRINT "RED ";
2420    =CODE("Y"): PRINT "YELLOW ";
2430    =CODE("B"): PRINT "BLUE ";
2440    =CODE("G"): PRINT "GREEN ";
2445  END SELect
2450 END DEFine color_name
```

This procedure changes the initial letter into the full name of the relevant color. Both these procedures are at the very end of the listing.

Back at the start of the program, we find a short section of code which prints out a view of the blocks. This could well have been a separate procedure, but as it is needed every time the QL cycles through the main loop, it seemed sensible to have it here.

```
30 REPeat main_loop
40  CLS: PRINT: PRINT
50  FOR x=5 TO 1 STEP -1
60   PRINT "          ";
70   FOR y=1 TO 6
80    PRINT CHR$(a(x,y));
90   END FOR y
100  PRINT
110  END FOR x
120  PRINT: PRINT
```

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 to 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 accepts the user's input, and from it determines which procedure should be called to act upon this input.

```
130  INPUT "?"!a$
140  PRINT
150  REMark terminate run by just pressi
ng ENTER
152    IF a$=""
154     PRINT "Program ends"
156     EXIT main_loop
158    END IF
160  IF a$(1 TO 8)="WHERE IS": where_is
170  IF a$(1 TO 12)="TELL ME WHAT": tell
_me_what
180  IF a$(1 TO 7)="SHUFFLE": shuffle
190  IF a$(1 TO 7)="PUT THE": put_the
200  PRINT: PRINT: PRINT "    >>>> Press
ENTER to continue": INPUT z$
210 END REPeat main_loop
```

Constructing an AI program leads one very quickly to appreciate the complexities of intelligence in operation. BLOCKWORLD operates in a very restricted domain, and reacts only to those situations which have been specifically allowed for (although, as we saw in the sample run, it managed to grapple with a situation for which I did not realise I had prepared it). Despite the limitations of domain and performance, BLOCKWORLD demands a lot of code, with a section to carry out each investigation, and to follow each command.

Look, for example, at the routine which determines the location of a specific block. First the program must check that you are asking for a block which is within its known universe. It does this by extracting — with line 260 — the initial letter of the block you are seeking, and checks — line 270 — that this is one of the four it recognizes:

```
240 DEFine PROCedure where_is
250  p=0
260  b$=a$(14)
270  IF b$="R" OR b$="Y" OR b$="B" OR b$
="G": GO TO 330
```

If you have asked it, for example, about a 'pink block', it uses the next routine (280 to 310) to randomly choose a reply, before returning to the main program:

```
280  IF RND>.7:
285   PRINT "I HAVE NO DATA WITH WHICH T
O ANSWER YOU"
290  ELSE
300   PRINT "SORRY, I HAVE NO INFORMATIO
N ON THAT"
305  END IF
310  RETurn
```

If it bypasses that, the QL starts searching for the block. We do not use FOR/END FOR loops for this search, as we want the program to be able to exit the search at any point. Therefore, a REPEAT/END REPEAT construction is used, with EXIT to leap out of the cycle when necessary. This part of the program gives the QL the first part of its required information regarding the block's location:

```
330  m=CODE(b$)
340  PRINT "          > LET ME SEE NOW <"
350  x=5
355  REPeat x_loop
360   y=1
365   REPeat y_loop
370    IF a(x,y)=m: GO TO 410
380    IF y<6
382     y=y+1
383    ELSE
384     EXIT y_loop
386    END IF
388   END REPeat y_loop
390   IF x>1
392    x=x-1
393   ELSE
394    EXIT x_loop
396   END IF
398  END REPeat x_loop
400  GO TO 280
410  IF x>1: GO TO 910: REMark on top of
 another
420  IF y>1: GO TO 530: REMark not on le
ft
```

The REM statements in the rest of this section explain what each one does:

```
440  REMark on left
450  PRINT "IT IS ON THE LEFT"
460  IF a(1,2)=46: PRINT "THERE IS NOTHI
NG": PRINT"   TO ITS IMMEDIATE RIGHT": G
O TO 790
470  q=a(1,2)
480  PRINT
490  PRINT "BESIDE IT, I CAN SEE THE "
```

```
500  color_name
510  PRINT "BLOCK"
520  GO TO 790
530  IF y<6: GO TO 650
540 :
550  REMark on right
560  PRINT
570  PRINT "IT IS ON THE RIGHT HAND SIDE
"
580  IF a(1,5)=46: PRINT "THERE IS NOTHI
NG": PRINT "  TO THE IMMEDIATE LEFT": GO
 TO 790
590  PRINT "TO ITS LEFT I SEE THE ";
600  q=a(1,5)
610  color_name
620  PRINT "ONE"
630  GO TO 790
640 :
650  REMark middle
660  PRINT
670  PRINT "IT IS IN POSITION"!y!"FROM T
HE LEFT"
680  IF a(x,y-1)=46: PRINT "THERE IS NOT
HING": PRINT "  ON ITS IMMEDIATE LEFT":
GO TO 730
690  q=a(x,y-1)
700  PRINT "THE ";
710  color_name
720  PRINT "BLOCK IS": PRINT "  TO ITS I
MMEDIATE LEFT"
730  IF a(x,y+1)=46: PRINT "NOTHING TOUC
HES IT ON THE RIGHT": GO TO 790
740  q=a(x,y+1)
750  PRINT: PRINT "I CAN SEE THE ";
760  color_name
```

```
770  PRINT "BLOCK": PRINT "  TO ITS RIGH
T, TOUCHING IT"
780 :
790  REMark anything above?
800  PRINT
810  p=x
820  IF x=5: GO TO 910
830  IF a(x+1,y)=46: PRINT "THERE IS NOT
HING ABOVE THAT": GO TO 310
840  PRINT: PRINT "ABOVE IT IS THE ";
850  q=a(x+1,y)
860  color_name
870  PRINT "BLOCK"
880  x=x+1
890  GO TO 820
900 :
910  REMark on top of another?
920  IF p<>0: x=p
930  PRINT
940  IF x=1: GO TO 310
950  PRINT "IT IS ";
960  PRINT "ON TOP OF THE ";
970  q=a(x-1,y)
980  color_name
990  PRINT "BLOCK"
1000  x=x-1
1010  IF x<2: GO TO 310
1020  GO TO 960
1030 END DEFine where_is
```

The next procedure is called on the QL if you ask the program to TELL ME WHAT YOU SEE. This is a much simpler procedure than the one which locates a specific block:

```
1050 DEFine PROCedure tell_me_what
1060  PRINT "STARTING FROM THE RIGHT"
1070  y=6
1075  REPeat y_loop
1080   x=5
1085   REPeat x_loop
1090    IF a(x,y)<>46: GO TO 1150
1100    IF y=1 AND x=1: PRINT "FINALLY "
;
1110    IF x=1 AND a(x,y)=46: PRINT "...
A SPACE ";: IF y>1: PRINT "THEN"
1120    IF x>1
1122     x=x-1
1123    ELSE
1124     EXIT x_loop
1126    END IF
1128   END REPeat x_loop
1130   IF y>1
1132    y=y-1
1133   ELSE
1134    EXIT y_loop
1136   END IF
1138  END REPeat y_loop
1140  RETurn
1150  choice=RND(1)
1155  SELect ON choice
1160   =0: PRINT "IT'S THE ";: GO TO 119
0
1170   =1: PRINT "MY SENSORS REGISTER TH
E ";: GO TO 1190
1175  END SELect
1180  PRINT "I SEE THE ";
1190  q=a(x,y)
1200  color_name
1210  PRINT "BLOCK"
```

```
1220  IF x=1: GO TO 1130
1230  x=x-1
1240  PRINT "...AND BELOW IT..."
1250  GO TO 1180
1260 END DEFine tell_me_what
```

The **shuffle** procedure is also relatively simple:

```
1270 :
1280 DEFine PROCedure shuffle
1290  PRINT
1300  IF RND>.5
1302   PRINT "        IT'S ABOUT TIME, TO
O"
1305  ELSE
1310   PRINT "     IT'S GOOD TO BE GIVEN
A CHANCE": PRINT "     TO DO WHAT I WANT"
1315  END IF
1320  FOR x=1 TO 5
1330   FOR y=1 TO 6
1340    a(x,y)=46
1350   END FOR y
1360  END FOR x
1370  y1=RND(1,6)
1380  y2=RND(1,6)
1390  IF y2=y1: GO TO 1380
1400  y3=RND(1,6)
1410  IF y3=y2 OR y3=y1: GO TO 1400
1420  y4=RND(1,6)
1430  IF y4=y3 OR y4=y2 OR y4=y1: GO TO
1420
1440  a(1,y1)=82
1450  a(1,y2)=89
1460  a(1,y3)=66
1470  a(1,y4)=71
1480 END DEFine shuffle
```

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

```
1500 DEFine PROCedure put_the
1510  IF RND>.5
1512   PRINT "      I UNDERSTAND"
1515  ELSE
1520   PRINT "         OK"
1525  END IF
1530  b$=a$(9): REMark object block
1540  IF b$="R": l=26
1550  IF b$="B": l=27
1560  IF b$="G": l=28
1570  IF b$="Y": l=29
1580  c$=a$(l)
1590  b=CODE(b$)
1600  c=CODE(c$)
1610  flag=c
1620  REMark find b$ block
1630  x=5
1635  REPeat x_loop
1640   y=1
1645   REPeat y_loop
1650    IF a(x,y)=b: GO TO 1740
1660    IF y<6
1662     y=y+1
1663    ELSE
1664     EXIT y_loop
1666    END IF
1668   END REPeat y_loop
1670   IF x>1
1672    x=x-1
```

```
1673    ELSE
1674     EXIT x_loop
1676    END IF
1678   END REPeat x_loop
1680   PRINT "I CAN'T FIND THE ";
1690   q=b
1700   color_name
1710   PRINT "ONE..."
1720   PAUSE 200
1730   RETurn
1740   r=x: s=y
1750   REMark object block is at r,s
1760   REMark is target block clear?
1770   IF a(r+1,s)=46: GO TO 1920: REMark
 "yes"
1780   IF a(r+2,s)=46: task=1: GO TO 1800
1790   task=3: IF a(r+3,s)=46: task=2
1800   FOR w=task TO 1 STEP -1
1810    PRINT "I MUST MOVE THE ";
1820    q=a(r+w,s)
1830    color_name
1840    PRINT "BLOCK"
1850    de=RND(1,6)
1860    IF de=s OR a(1,de)=c OR a(2,de)=c
 OR a(3,de)=c: GO TO 1850
1870    PRINT "I'M MOVING IT TO ROW ";de
1880    l=1
1885    REPeat l_loop
1890     IF a(l,de)=46
1892      a(l,de)=a(r+w,s)
1894      a(r+w,s)=46
1896      EXIT l_loop
1900     ELSE
1902      l=l+1
1904     END IF
```

```
1906   END REPeat l_loop
1910  END FOR w
1920  REMark target block at r,s now cle
ar
1930  REMark is object block clear?
1940  REMark find object block
1950  x=5
1955  REPeat x_loop
1960   y=1
1965   REPeat y_loop
1970    IF a(x,y)=c: GO TO 2070
1980    IF y<6
1982     y=y+1
1983    ELSE
1984     EXIT y_loop
1986    END IF
1988   END REPeat y_loop
1990   IF x>1
1992    x=x-1
1993   ELSE
1994    EXIT x_loop
1996   END IF
1998  END REPeat x_loop
2000  PRINT "I CAN'T FIND THE ";
2010  q=c
2020  color_name
2030  PRINT "BLOCK"
2040  PAUSE 200
2050  RETurn
2060  REMark c has been found
2070  t=x: u=y: REMark location of c
2080  IF a(t+1,u)=46: GO TO 2260
2090  IF a(t+2,u)=46: task=1: GO TO 2110
2100  IF a(t+3,u)=46: task=2
2110  de=RND(1,6)
2120  IF de=u OR de=s: GO TO 2110
```

```
2130  FOR w=task TO 1 STEP -1
2140   PRINT "NOW I'LL MOVE THE ";
2150   q=a(t+w,u)
2160   color_name
2170   PRINT "ONE"
2180   PRINT
2190   PRINT "I'M MOVING IT TO ROW ";de
2200   l=1
2205   REPeat l_loop
2210    IF a(l,de)=46
2212     a(l,de)=a(t+w,u)
2214     a(t+w,u)=46
2216     EXIT l_loop
2220    ELSE
2222     l=l+1
2224    END IF
2226   END REPeat l_loop
2230  END FOR w
2240  REMark object block now clear
2250  REMark make the move
2260  PRINT "I'M NOW MOVING THE ";
2270  q=a(r,s): z=a(r,s)
2280  color_name
2290  PRINT "ONE"
2300  PRINT "  ONTO THE ";
2310  IF a(t,u)=46
2312   a(t,u)=flag
2314  ELSE
2320   q=a(t,u)
2325  END IF
2330  color_name
2340  PRINT "BLOCK"
2350  a(r,s)=46
2360  a(t+1,u)=z
2370  PAUSE 200
2380 END DEFine put_the
2390 :
```

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

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

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

TELL ME WHAT YOU SEE (or CAN SEE).

SHUFFLE THE BLOCKS.

PUT THE color BLOCK ON THE color ONE.

You can quit the program at any time (as indicated by line 150) by simply pressing the ENTER key on your QL when you are prompted for a question or command.

Here, now, is the complete listing of BLOCKWORLD:

```
10 REMark ==========
11 REMark BLOCKWORLD
12 REMark ==========
15 :
20 initialise
30 REPeat main_loop
```

```
2130  FOR w=task TO 1 STEP -1
2140   PRINT "NOW I'LL MOVE THE ";
2150   q=a(t+w,u)
2160   color_name
2170   PRINT "ONE"
2180   PRINT
2190   PRINT "I'M MOVING IT TO ROW ";de
2200   l=1
2205   REPeat l_loop
2210    IF a(l,de)=46
2212     a(l,de)=a(t+w,u)
2214     a(t+w,u)=46
2216     EXIT l_loop
2220    ELSE
2222     l=l+1
2224    END IF
2226   END REPeat l_loop
2230  END FOR w
2240  REMark object block now clear
2250  REMark make the move
2260  PRINT "I'M NOW MOVING THE ";
2270  q=a(r,s): z=a(r,s)
2280  color_name
2290  PRINT "ONE"
2300  PRINT "  ONTO THE ";
2310  IF a(t,u)=46
2312   a(t,u)=flag
2314  ELSE
2320   q=a(t,u)
2325  END IF
2330  color_name
2340  PRINT "BLOCK"
2350  a(r,s)=46
2360  a(t+1,u)=z
2370  PAUSE 200
2380 END DEFine put_the
2390 :
```

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

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

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

TELL ME WHAT YOU SEE (or CAN SEE).

SHUFFLE THE BLOCKS.

PUT THE color BLOCK ON THE color ONE.

You can quit the program at any time (as indicated by line 150) by simply pressing the ENTER key on your QL when you are prompted for a question or command.

Here, now, is the complete listing of BLOCKWORLD:

```
10 REMark ==========
11 REMark BLOCKWORLD
12 REMark ==========
15 :
20 initialise
30 REPeat main_loop
```

```
430 :
440  REMark on left
450  PRINT "IT IS ON THE LEFT"
460  IF a(1,2)=46: PRINT "THERE IS NOTHI
NG": PRINT"   TO ITS IMMEDIATE RIGHT": G
O TO 790
470  q=a(1,2)
480  PRINT
490  PRINT "BESIDE IT, I CAN SEE THE "
500  color_name
510  PRINT "BLOCK"
520  GO TO 790
530  IF y<6: GO TO 650
540 :
550  REMark on right
560  PRINT
570  PRINT "IT IS ON THE RIGHT HAND SIDE
"
580  IF a(1,5)=46: PRINT "THERE IS NOTHI
NG": PRINT "  TO THE IMMEDIATE LEFT": GO
 TO 790
590  PRINT "TO ITS LEFT I SEE THE ";
600  q=a(1,5)
610  color_name
620  PRINT "ONE"
630  GO TO 790
640 :
650  REMark middle
660  PRINT
670  PRINT "IT IS IN POSITION"!y!"FROM T
HE LEFT"
680  IF a(x,y-1)=46: PRINT "THERE IS NOT
HING": PRINT "  ON ITS IMMEDIATE LEFT":
GO TO 730
690  q=a(x,y-1)
```

```
700  PRINT "THE ";
710  color_name
720  PRINT "BLOCK IS": PRINT "  TO ITS I
MMEDIATE LEFT"
730  IF a(x,y+1)=46: PRINT "NOTHING TOUC
HES IT ON THE RIGHT": GO TO 790
740  q=a(x,y+1)
750  PRINT: PRINT "I CAN SEE THE ";
760  color_name
770  PRINT "BLOCK": PRINT "  TO ITS RIGH
T, TOUCHING IT"
780 :
790  REMark anything above?
800  PRINT
810  p=x
820  IF x=5: GO TO 910
830  IF a(x+1,y)=46: PRINT "THERE IS NOT
HING ABOVE THAT": GO TO 310
840  PRINT: PRINT "ABOVE IT IS THE ";
850  q=a(x+1,y)
860  color_name
870  PRINT "BLOCK"
880  x=x+1
890  GO TO 820
900 :
910  REMark on top of another?
920  IF p<>0: x=p
930  PRINT
940  IF x=1: GO TO 310
950  PRINT "IT IS ";
960  PRINT "ON TOP OF THE ";
970  q=a(x-1,y)
980  color_name
990  PRINT "BLOCK"
1000  x=x-1
```

```
430 :
440  REMark on left
450  PRINT "IT IS ON THE LEFT"
460  IF a(1,2)=46: PRINT "THERE IS NOTHI
NG": PRINT"   TO ITS IMMEDIATE RIGHT": G
O TO 790
470  q=a(1,2)
480  PRINT
490  PRINT "BESIDE IT, I CAN SEE THE "
500  color_name
510  PRINT "BLOCK"
520  GO TO 790
530  IF y<6: GO TO 650
540 :
550  REMark on right
560  PRINT
570  PRINT "IT IS ON THE RIGHT HAND SIDE
"
580  IF a(1,5)=46: PRINT "THERE IS NOTHI
NG": PRINT "  TO THE IMMEDIATE LEFT": GO
 TO 790
590  PRINT "TO ITS LEFT I SEE THE ";
600  q=a(1,5)
610  color_name
620  PRINT "ONE"
630  GO TO 790
640 :
650  REMark middle
660  PRINT
670  PRINT "IT IS IN POSITION"!y!"FROM T
HE LEFT"
680  IF a(x,y-1)=46: PRINT "THERE IS NOT
HING": PRINT "  ON ITS IMMEDIATE LEFT":
GO TO 730
690  q=a(x,y-1)
```

```
700  PRINT "THE ";
710  color_name
720  PRINT "BLOCK IS": PRINT "  TO ITS I
MMEDIATE LEFT"
730  IF a(x,y+1)=46: PRINT "NOTHING TOUC
HES IT ON THE RIGHT": GO TO 790
740  q=a(x,y+1)
750  PRINT: PRINT "I CAN SEE THE ";
760  color_name
770  PRINT "BLOCK": PRINT "  TO ITS RIGH
T, TOUCHING IT"
780 :
790  REMark anything above?
800  PRINT
810  p=x
820  IF x=5: GO TO 910
830  IF a(x+1,y)=46: PRINT "THERE IS NOT
HING ABOVE THAT": GO TO 310
840  PRINT: PRINT "ABOVE IT IS THE ";
850  q=a(x+1,y)
860  color_name
870  PRINT "BLOCK"
880  x=x+1
890  GO TO 820
900 :
910  REMark on top of another?
920  IF p<>0: x=p
930  PRINT
940  IF x=1: GO TO 310
950  PRINT "IT IS ";
960  PRINT "ON TOP OF THE ";
970  q=a(x-1,y)
980  color_name
990  PRINT "BLOCK"
1000  x=x-1
```

```
1470  a(1,y4)=71
1480 END DEFine shuffle
1490 :
1500 DEFine PROCedure put_the
1510  IF RND>.5
1512   PRINT "      I UNDERSTAND"
1515  ELSE
1520   PRINT "         OK"
1525  END IF
1530  b$=a$(9): REMark object block
1540  IF b$="R": l=26
1550  IF b$="B": l=27
1560  IF b$="G": l=28
1570  IF b$="Y": l=29
1580  c$=a$(l)
1590  b=CODE(b$)
1600  c=CODE(c$)
1610  flag=c
1620  REMark find b$ block
1630  x=5
1635  REPeat x_loop
1640   y=1
1645   REPeat y_loop
1650    IF a(x,y)=b: GO TO 1740
1660    IF y<6
1662     y=y+1
1663    ELSE
1664     EXIT y_loop
1666    END IF
1668   END REPeat y_loop
1670   IF x>1
1672    x=x-1
1673   ELSE
1674    EXIT x_loop
1676   END IF
1678  END REPeat x_loop
```

```
1680  PRINT "I CAN'T FIND THE ";
1690  q=b
1700  color_name
1710  PRINT "ONE..."
1720  PAUSE 200
1730  RETurn
1740  r=x: s=y
1750  REMark object block is at r,s
1760  REMark is target block clear?
1770  IF a(r+1,s)=46: GO TO 1920: REMark
 "yes"
1780  IF a(r+2,s)=46: task=1: GO TO 1800
1790  task=3: IF a(r+3,s)=46: task=2
1800  FOR w=task TO 1 STEP -1
1810   PRINT "I MUST MOVE THE ";
1820   q=a(r+w,s)
1830   color_name
1840   PRINT "BLOCK"
1850   de=RND(1,6)
1860   IF de=s OR a(1,de)=c OR a(2,de)=c
 OR a(3,de)=c: GO TO 1850
1870   PRINT "I'M MOVING IT TO ROW ";de
1880   l=1
1885   REPeat l_loop
1890    IF a(l,de)=46
1892     a(l,de)=a(r+w,s)
1894     a(r+w,s)=46
1896     EXIT l_loop
1900    ELSE
1902     l=l+1
1904    END IF
1906   END REPeat l_loop
1910  END FOR w
1920  REMark target block at r,s now cle
ar
```

```
1930  REMark is object block clear?
1940  REMark find object block
1950  x=5
1955  REPeat x_loop
1960   y=1
1965   REPeat y_loop
1970    IF a(x,y)=c: GO TO 2070
1980    IF y<6
1982     y=y+1
1983    ELSE
1984     EXIT y_loop
1986    END IF
1988   END REPeat y_loop
1990   IF x>1
1992    x=x-1
1993   ELSE
1994    EXIT x_loop
1996   END IF
1998  END REPeat x_loop
2000  PRINT "I CAN'T FIND THE ";
2010  q=c
2020  color_name
2030  PRINT "BLOCK"
2040  PAUSE 200
2050  RETurn
2060  REMark c has been found
2070  t=x: u=y: REMark location of c
2080  IF a(t+1,u)=46: GO TO 2260
2090  IF a(t+2,u)=46: task=1: GO TO 2110
2100  IF a(t+3,u)=46: task=2
2110  de=RND(1,6)
2120  IF de=u OR de=s: GO TO 2110
2130  FOR w=task TO 1 STEP -1
2140   PRINT "NOW I'LL MOVE THE ";
2150   q=a(t+w,u)
2160   color_name
```

```
2170    PRINT "ONE"
2180    PRINT
2190    PRINT "I'M MOVING IT TO ROW ";de
2200    l=1
2205    REPeat l_loop
2210     IF a(l,de)=46
2212      a(l,de)=a(t+w,u)
2214      a(t+w,u)=46
2216      EXIT l_loop
2220     ELSE
2222      l=l+1
2224     END IF
2226    END REPeat l_loop
2230   END FOR w
2240   REMark object block now clear
2250   REMark make the move
2260   PRINT "I'M NOW MOVING THE ";
2270   q=a(r,s): z=a(r,s)
2280   color_name
2290   PRINT "ONE"
2300   PRINT "  ONTO THE ";
2310   IF a(t,u)=46
2312    a(t,u)=flag
2314   ELSE
2320    q=a(t,u)
2325   END IF
2330   color_name
2340   PRINT "BLOCK"
2350   a(r,s)=46
2360   a(t+1,u)=z
2370   PAUSE 200
2380 END DEFine put_the
2390 :
2400 DEFine PROCedure color_name
2405  SELect ON q
2410   =CODE("R"): PRINT "RED ";
2420   =CODE("Y"): PRINT "YELLOW ";
```

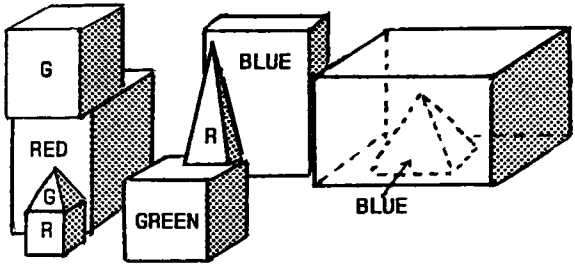
```
2430    =CODE("B"): PRINT "BLUE ";
2440    =CODE("G"): PRINT "GREEN ";
2445   END SELect
2450 END DEFine color_name
2460 :
2470 DEFine PROCedure initialise
2472   BORDER 2,1
2474   PAPER 85
2476   INK 1
2480   CLS: CLS #0
2490   RANDOMISE
2500   DIM a(5,6)
2510   FOR x=1 TO 5
2520     FOR y=1 TO 6
2530       a(x,y)=46
2540     END FOR y
2550   END FOR x
2560   a(1,2)=CODE("R"): REMark red block
2570   a(1,3)=CODE("Y"): REMark yellow
2580   a(1,4)=CODE("B"): REMark blue
2590   a(1,5)=CODE("G"): REMark green
2600 END DEFine initialise
2610 :
2620 REMark ========
2630 REMark commands
2640 REMark ========
2650 :
2660 REMark ? TELL ME WHAT YOU SEE
2670 REMark ? WHERE IS THE ... BLOCK
2680 REMark ? PUT THE ... BLOCK ON THE .
.. ONE
2690 REMark ? SHUFFLE THE BLOCKS AROUND
A BIT
2700 :
2710 REMark ==============
2720 REMark END BLOCKWORLD
2730 REMark ==============
```

IMPROVING THE PROGRAM

Although BLOCKWORLD performs quite creditably in its present form, there is much that can be done to it to develop it. As you have seen, it has a simple modular structure, with a single procedure 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 QL procedures.

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 that 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.

CHAPTER EIGHT — THE DOCTOR IS IN

No study of 'natural language processing' would be complete with a discussion of ELIZA, the most famous program of this type (and, at one stage, the most famous computer program of all).

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

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

You use Weizenbaum's program by typing in your side of a 'conversation', to which the program replies, using its inner stock of answers. Here is part of a typical exchange with the ELIZA implementation given in this book:

```
>? I AM AFRAID OF SPEAKING IN PUBLIC

DO YOU THINK IT'S ABSOLUTELY NORMAL
TO BE AFRAID OF SPEAKING IN PUBLIC
-----------------------------------

>? I THINK A LOT OF PEOPLE ARE

DO YOU REALLY THINK SO?
-----------------------------------

>? YES I DO

YOU SEEM VERY CERTAIN ABOUT IT
-----------------------------------

>? TALKING TO A COMPUTER IS RATHER ODD

ARE YOU THINKING ABOUT ME IN PARTICULAR
```

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

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

Weizenbaum reported his work on ELIZA to the computer press in due course ("ELIZA — A Computer Program for the Study of Natural Language Communication Between Man and Machine", *Communications of the Association for Computing Machinery*, vol. 9, no. 1 [January 1965], pp. 36-45) and soon a number of versions of it — based on his description — were running at other institutions in the United States.

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

He reports that even his secretary, who had worked with him on the program's development for many months, and therefore should have been one of those best situated to know it was only a program, started to relate to it emotionally. On one classic occasion, his secretary started using the program and after only a few sentences of dialogue had been exchanged became embarrassed and secretive. She asked if he would leave the room while she continued the 'conversation'.

Weizenbaum suggested, on another occasion, that he should rig up a printer to get a transcript of the talks people were having with ELIZA. This idea was greeted with horror

as it would mean he would be prying into very private conversations.

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

THE RUSSIAN CONNECTION

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

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

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

Colby, who we mentioned a short time ago, had met Weizenbaum some time earlier, at Stanford. Colby — Professor of Psychiatry at UCLA — was interested in artificial intelligence. He thought its findings might possibly lead to new views on human thinking (and Colby hoped to gain new insights into neurotic behavior through his studies). Before Weizenbaum's original paper on ELIZA appeared, a short note on it was published by Colby in the *Journal of Nervous and Mental Diseases.*

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

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

SHORT, SHARP SHOCKS

Weizenbaum was shocked that even a single practicing psychiatrist could advance the view that the healing process could be replaced purely by mechanical technique. Such a thought had never crossed his mind. Furthermore, even if it could be done, it **should not** be

done. There are some areas where machines should never be allowed to stray, claimed Weizenbaum, even if they have the ability to do so.

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

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

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

And this is where Weizenbaum's third 'shock' came in. Remember, he had been startled by the identification with and the unequivocal anthropomorphization of the program. Then he was very alarmed at the suggestions that somehow ELIZA could take the place of, or assist, human therapists. His third 'shock' came from his observation that many people came to believe that somehow the program was important in demonstrating that a real solution to the problems of a machine understanding human language was

at hand. He dismissed this idea out of hand. Indeed, in the original paper on the program, Weizenbaum had been at pains to point out that it was impossible to find a general solution to this problem.

I said earlier that I did not agree with all in Weizenbaum's book *Computer Power and Human Reason.* One of the points upon which I disagree is the 'there are some things which should never be done by machines'. John McCarthy (1976, "An Unreasonable Book", in *Three Reviews of J. Weizenabum's Computer Power and Human Reason, Memo AIM-291,* Stanford AI Laboratory, November), advances the view that if there are functions which a computer should not be taught to carry out, these should not be done at all, by a person or a machine.

Others agree. In the book *Artificial Reality* (1983: Addison-Wesley Publishing Co., Reading, MA; p. 168), Myron W. Krueger suggests that even if Weizenbaum's horror at the thought of using his program — or a development of it — for genuine therapy was real, such fear was groundless and misplaced.

However, regardless of my views (or others') of Weizenbaum's thesis, and of the value of the book (I've already said I think you should read it, if only to give your own mental mill grist regarding the debate), there is no doubt that ELIZA, as DOCTOR, has proved an extremely entertaining companion. You will soon prove this assertion for yourself.

As well as enjoying the program, you'll also be in a position to judge whether or not it actually suggests that intelligence resides in the machine which is running it. Adrian Berry (in *The Super-Intelligent Machine,* London: Jonathan

Cape Ltd., 1983; p. 63) concludes that ELIZA (and PARRY, a program which is designed to mimic a paranoid patient) is a pretty poor advocate for the possibilities of true artificial intelligence.

You'll find that your own views will sway back and forth as you use the program. When the DOCTOR produces a particularly inspired or appropriate remark, you'll feel this is sure evidence of intelligence, on at least some level. At other times, you'll discover the DOCTOR is great simply for entertainment. Nobody, you claim, could manage to ask such absurd questions (DID YOU COME TO ME BECAUSE I WAS DEAD? one implementation asked Berry, he reports in the book mentioned above) or innocently create superb nonsequiturs as the following exchange which comes from the sample run produced by the program in this book (the DOCTOR is speaking in upper case):

```
I feel you're evading my questions
  DO YOU OFTEN FEEL I AM EVADING YOUR
  QUESTIONS
I do when I talk to a computer
  THINGS HAVE A HABIT OF HAPPENING AT
  MORE OR LESS THE RIGHT TIME
```

What can one do in response to a statement like that except raise a toast to Joseph W., progenitor of ELIZA and its offspring, and hope for more? I think it's time we turned back to the computer.

THE PROGRAM

Now that we've heard about how ELIZA was written, and the reactions to it operating in the DOCTOR mode, let's see how our own version of the program works. Here's one run with the program. The human input is preceded by > ? and a line of dashes cuts off each statement by the human and the response to it from the DOCTOR:

```
THE DOCTOR IS IN...PLEASE STAND BY

WELCOME TO ANOTHER SESSION
WITH THE DOCTOR.....

HELLO...

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

>? HI THERE DOC

HI TO YOU...RELAX NOW, AND LET'S TALK
ABOUT YOUR SITUATION
---------------------------------

>? I HAVE A SERIOUS PROBLEM

HOW CAN I HELP YOU WITH A SERIOUS
PROBLEM
---------------------------------

>? IT IS TO DO WITH MY MAD TEACHER

ARE YOU CERTAIN THAT IT'S TO DO WITH
YOUR MAD TEACHER
---------------------------------
```

>? YES I AM VERY CERTAIN

YOU'RE PRETTY POSITIVE

>? IT COMES FROM WRESTLING WITH IT DAY AND NIGHT

EARLIER YOU MENTIONED YOUR MAD TEACHER TELL ME A BIT MORE ABOUT THAT

>? IT STARTED WHEN I WAS TEN YEARS OLD

THAT'S VERY INTERESTING

>? I AM AFRAID TO GO TO SCHOOL NOW

DID YOU COME TO ME BECAUSE YOU ARE AFRAID TO GO TO SCHOOL NOW

>? I THINK THAT IS TRUE

BUT YOU ARE NOT SURE THAT IS TRUE

>? PERHAPS THERE ARE OTHER REASONS

THAT'S VERY INTERESTING

>? I ALSO HAVE PROBLEMS WITH
MY CRAZY BROTHER

PLEASE ELABORATE ON THAT

>? I THINK HE HATES ME

DO YOU DOUBT HE HATES YOU

>? I FEEL YOU'RE EVADING MY QUESTIONS

DO YOU OFTEN FEEL I AM EVADING YOUR
QUESTIONS

>? I DO WHEN I TALK TO A COMPUTER

THINGS HAVE A HABIT OF HAPPENING AT
MORE OR LESS THE RIGHT TIME

>? THAT'S A DUMB ANSWER FROM A COMPUTER

WHAT ASPECT OF COMPUTERS INTERESTS
YOU THE MOST?

>? HELPING ME WITH MY PROBLEMS

PLEASE ELABORATE ON THAT

>? I WANT TO BEAT UP MY BROTHER

WHAT WOULD IT ADD TO YOUR LIFE IF
YOU GOT TO BEAT UP YOUR BROTHER

>? IT WOULD MAKE ME FEEL GREAT

LET'S EXPLORE THAT STATEMENT A BIT

>? I FEEL INFERIOR WHEN HE PUTS ME DOWN

DO YOU ENJOY FEELING INFERIOR WHEN HE
PUTS YOU DOWN

>? OF COURSE I DON'T ENJOY IT

DOES ANY CONNECTION BETWEEN THAT AND
YOUR BROTHER SUGGEST ITSELF?

>? THE LINK IS OBVIOUS TO ANYONE

PLEASE ELABORATE ON THAT

```
>? I WANT SOME ASSISTANCE WITH MY FEARS

WHAT WOULD IT MEAN TO YOU IF YOU
GOT SOME ASSISTANCE WITH YOUR FEARS
--------------------------------

>? I THINK IT WOULD MAKE ME BRAVER

DO YOU DOUBT IT WOULD MAKE YOU BRAVER
--------------------------------

>? I ALWAYS WANT TO RUN AWAY

ISN'T 'ALWAYS' A LITTLE STRONG?

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

>? I GUESS IT MIGHT BE

WHY DON'T WE GO BACK AND DISCUSS
YOUR FEARS A LITTLE MORE?

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

The sample run is fairly impressive, and once you've used the program a few times, you're certain to discover the kind of phrases and words which it responds to best. As well, as I pointed out earlier, it makes a good demonstration to run for those who are not used to computers. (But be careful in case some of that 'powerful delusional thinking' comes into play, and you find one of your friends confiding his or her darkest thoughts to your machine).

HOW IT WORKS

The program is, in essence, simple. The computer accepts an input string from the user, then scans through this, looking first for 'trigger phrases' (such as I THINK or WHY CAN'T I) for which it has a stock reply. It it fails to find any of these, it will look for 'trigger words' (such as COMPUTER), and similarly choose one of three pre-programmed replies to that trigger.

If it doesn't find a trigger, it will look to see if it has stored a phrase of yours from an earlier input (such as ATTITUDE or MAD TEACHER) and, if so, create a question around that. If all these fail, it will choose from a set of random replies (I'M NOT SURE I UNDERSTAND THAT FULLY or the like) designed to keep the conversation moving.

```
EARLIER YOU MENTIONED YOUR ATTITUDE
TELL ME A BIT MORE ABOUT THAT

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

>? IT IS A WORRY TO ME

WHAT DEGREE OF CERTAINTY WOULD YOU
PLACE ON IT BEING A WORRY TO YOU

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

>? A LOT BECAUSE THAT IS THE WAY IT IS

WHAT OTHER REASONS COME TO MIND?
```

THE REPLY DATABASE

The most important part of the program, and the one which you can have the most fun modifying, and adding to, is the huge set of DATA statements at the end of the program. (You'll see the very last DATA statement is a series of asterisks. This is to terminate the READing, and to allow you to add extra bits of DATA without then having to count right through to see how many there are. The DIM statements for the string arrays to hold this DATA are larger than they need to be, so that you can add extra words if you want to.)

You'll see that the first set of DATA statements cover 'conjugations':

```
1420 REMark ** conjugations **
1430 DATA "AM","ARE","WAS","WERE","I","Y
OU","MY","YOUR"
1440 DATA "YOU'VE","I'VE","YOUR","MY","A
RE","AM"
1450 DATA "YOU'RE","I AM","YOURS","MINE"
,"YOU","ME","ME","YOU","*","*"
```

The computer uses these in exactly the same way it handled English in TRANSLATE, swapping one for its pair. This means that if you say, for example, I AM GETTING YOUR MESSAGE the computer could simply change the words around and say back in reply YOU ARE GETTING MY MESSAGE. In fact, this is the form this DOCTOR program originally took, and even this limited kind of exchange can be significantly interesting.

After this come the major DATA statements, which look after most of the phrase swapping. They are of two types.

The first type uses either a word or a short phrase (which was used as the start of the user input) and then chooses the entire reply from the database, without taking any words directly from the user input. In these examples, the first DATA statement of each four is the 'trigger' from the user input and the next three are those from which the computer chooses its reply:

```
1830 DATA "HOW"
1840 DATA "HOW WOULD YOU SOLVE THAT?"
1850 DATA "IT WOULD BE BEST TO ANSWER TH
AT FOR   YOURSELF"
1860 DATA "WHAT IS IT YOU'RE REALLY ASKI
NG?"
```

*

```
2030 DATA "BECAUSE"
2040 DATA "IS THAT THE REAL REASON?"
2050 DATA "WHAT OTHER REASONS COME TO MI
ND?"
2060 DATA "WHAT ELSE DOES THAT EXPLAIN?"
2070 DATA "SORRY"
```

Of greater interest are the phrases which are triggered to be used as the start of the computer's reply, with the balance of the answer coming from the original user input (after any needed conjugation changes have been made):

```
1470 DATA "I NEED"
1480 DATA "WHY DO YOU NEED*"
1490 DATA "WOULD IT BE REALLY HELPFUL IF
 YOU     GOT*"
1500 DATA "ARE YOU SURE YOU NEED*"
```

*

```
1670 DATA "I AM"
1680 DATA "DID YOU COME TO ME BECAUSE YO
U        ARE*"
1690 DATA "HOW LONG HAVE YOU BEEN*"
1700 DATA "DO YOU THINK IT'S ABSOLUTELY
NORMAL    TO BE*"
```

*

```
2510 DATA "IS IT"
2520 DATA "DO YOU THINK IT IS*"
2530 DATA "IN WHAT CIRCUMSTANCES WOULD I
T*"
2540 DATA "IT COULD WELL BE THAT*"
```

You'll see that each of the phrases which form part of the reply end with '*', which the computer uses as a flag to indicate that part of the original input must be modified to complete sentence.

Let's see how it works in practice. Suppose the user input was as follows:

I WANT TO SHOW YOU THE TRUTH

The DOCTOR scans through the contents of string array C$ and finds the element which contains 'I WANT'. The equivalent elements of arrays D$, E$ and F$ contain the opening portions of suitable replies, as you can see:

```
1750 DATA "I WANT"
1760 DATA "WHAT WOULD IT MEAN TO YOU IF
YOU       GOT*"
1770 DATA "WHY DO YOU WANT*"
1780 DATA "WHAT WOULD IT ADD TO YOUR LIF
E IF      YOU GOT*"
```

The DOCTOR generates a random number between one and three and prints the D$ element if it is one, the E$ if two and the F$ if three (having previously checked to see if it ends in an asterisk, and if it does, notes this, then strips the asterisk off before printing). Assume the computer has chosen the D$ reply. Its answer so far, then, is:

WHY DO YOU WANT

Then, it goes through the balance of the user input (the material following I WANT), treating it in exactly the same way (using the same code, in fact) as the TRANSLATE program changed English words to French ones, swapping such things as YOU for I and ARE for AM (so I AM becomes YOU ARE). It prints up each word as it processes it, leaving the user-input word unchanged if there is nothing which needs swapping in the conjugation section.

The original phrase . . .

I WANT TO SHOW YOU THE TRUTH

. . . has then been transformed to . . .

WHY DO YOU WANT TO SHOW ME THE TRUTH

This is 'all' the program does, but as you'll soon be proving for yourself, it creates a remarkable effect.

If it cannot find a trigger phrase to combine with part of the user input, the DOCTOR looks for a trigger word, anywhere in the input (rather than just at the beginning, where it looks for phrases). Trigger words include COMPUTER and FRIENDS, producing results like these:

WHY DO YOU BRING UP THE SUBJECT OF FRIENDS?

PLEASE TELL ME MORE ABOUT YOUR FRIENDSHIP

. . . and . . .

WHAT ASPECTS OF COMPUTERS INTERESTS YOU THE MOST

ARE YOU THINKING ABOUT ME IN PARTICULAR?

THE PROGRAM STRUCTURE

The program begins, like the others in this book, with a call to an initialisation routine (this one starting at 1140). After that, the program prints a blank line, then the dashed line ruling off one exchange from the other, followed by another blank line.

```
 10 REMark ==========
 11 REMark = DOCTOR =
 12 REMark ==========
 15 :
 20 initialise
 22 consult
 24 :
 26 DEFine PROCedure consult
 30  PRINT: PRINT "-------------------
--------------": PRINT
 40  PRINT ">";: INPUT X$
 50  REMark quit by pressing ENTER
 52   IF X$="" 
 54    PRINT "program ended"
 56    STOP
```

```
 58   END IF
 60   PRINT
 70  REMark to control repetition
 72   IF X$=Z$
 74    PRINT "PLEASE DON'T REPEAT YOURSEL
F"
 76    consult
 78   ELSE
 80    Z$=X$
 85   END IF
 90  REMark say goodbye
```

Line 40 prints the "> " prompt, then accepts the user input. If the input is the empty string (that is, the user has just pressed RETURN rather than entering a phrase) the program terminates.

Line 60 prints a blank line, and line 70 compares this input (X$) with the one given the previous time (Z$) and if it finds they are the same says PLEASE DON'T REPEAT YOURSELF and then returns to 30 for new input. Line 80 sets the new input equal to Z$, for checking the next time around. If the first seven letters of the input spell out GOODBYE the computer replies with OK, SEE YOU AGAIN SOMETIME and terminates the program.

Having survived this series of hurdles, the work begins in earnest:

```
100 REMark look for trigger phrases at s
tart of input
110  L=0
115  REPeat loop_L
120   L=L+1
130   look=LEN(C$(L))
```

```
140    IF X$(1 TO look)=C$(L): GO TO 360:
 REMark trigger found
150    IF L>=K: EXIT loop_L
```

It scans, using the elements of the C$ array, the first portion of the input, looking for a match. If it finds one, action moves to line 360 where the job of matching the input phrase with the rest of the player input is carried out:

```
360 REMark trigger phrase found at start
 of input
370  trigger=RND(1,3)
375   SELect ON trigger
380     =1: G$=D$(L)
390     =2: G$=E$(L)
400     =3: G$=F$(L)
405   END SELect
410 REMark check to see if ends in aster
isk, if so needs part of input added
420  FLAG=0
430  IF "*" INSTR G$
432   FLAG=1
434   R=LEN(G$)
436   G$=G$(1 TO R-1)
438  END IF
440  PRINT G$;" ";
450  IF FLAG=0: consult: REMark no need
for added material
460 REMark now use balance of input
465  IF look+2>LEN(X$): consult
470  X$=" " & X$(look+2 TO) & " "
480 :
490 REMark conjugation changes
500 REMark also look for "MY" to trigger
 "MYFLAG"(K$)
```

```
510  look=LEN(X$)
520  M=0
525  REPeat M_loop
530   M=M+1
540   IF M=look: consult
550   IF X$(M)<>" ": NEXT M_loop
570   x=M+1
580   Y=0
585   REPeat Y_loop
590    Y=Y+1
600    IF X$(x+Y)=" "
602     Q$=X$(x TO x+Y-1)
604     EXIT Y_loop
606    END IF
610    IF x+Y>250: NEXT M_loop
620   END REPeat Y_loop
630   MN=0
635   REPeat mn_loop
640    MN=MN+1
650    IF Q$="MY" AND K$=""
652     K$=X$(x+3 TO LEN(X$)-1)
654    END IF
660    IF Q$=A$(MN)
662     PRINT B$(MN);" ";
664     NEXT M_loop
666    END IF
670    IF MN>=KK: EXIT mn_loop
675   END REPeat mn_loop
680   PRINT Q$;" ";
690  END REPeat M_loop
```

The section from lines 370 through to 400 chooses one of the three reply openings, from D$(n), E$(n) and F$(n). Line 420 sets a flag (called FLAG) to zero, and then uses line 430 to see if the chosen phrase ends with an asterisk (telling it, you'll recall, that this is only a partial reply, with additional material needed from the user input).

If it finds that there is an asterisk at the end, the flag is set to one and the final part of line 430 strips the flag off.

Line 440 prints the chosen phrase. If FLAG still equals zero (line 450) the program goes back to line 30 for the next input. If not, the DOCTOR must scan through the balance of the user input, making the conjugation changes (using, as I pointed out earlier, the same code as TRANSLATE employed) needed.

As well, as the REM statement in line 500 points out, the program is looking for the word MY to trigger K$, the 'myflag'. If it finds the word MY in the input (such as in the sentence IT IS TO DO WITH MY MAD TEACHER) and the 'myflag' (K$) has not been assigned, it will take the balance of the user input from the word MY and assign that to K$, so — in this case — it would be set to MAD TEACHER. Later, if the DOCTOR cannot find a trigger in a user input, it can use K$ with other phrases (such as EARLIER YOU MENTIONED YOUR MAD TEACHER. TELL ME A BIT MORE ABOUT THAT) to keep the conversation going. The effect on users of this tiny piece of trickery can be quite extraordinary.

If no trigger phrase has been found, the computer looks for a trigger word, using this section of code:

```
160 REMark program gets here if no trigg
er phrase found at start of X$
170 REMark now look for trigger words wi
thin input
180  X$=" " & X$ & " "
190  M=LEN(X$)
200  L=0
205  REPeat L_loop
210   L=L+1
```

```
220    IF L=M-1: GO TO 800: REMark no tri
gger found
230    IF X$(L)<>" ": NEXT L_loop
250    x=L+1
260    Y=0
265    REPeat Y_loop
270     Y=Y+1
280     IF X$(x+Y)=" "
282      Q$=X$(x TO x+Y-1)
284      EXIT Y_loop
286     END IF
290    END REPeat Y_loop
300    N=0
305    REPeat N_loop
310     N=N+1
320     IF Q$=C$(N): GO TO 710: REMark tr
igger word found
325     IF N>=K: EXIT N_loop
330    END REPeat N_loop
340   END REPeat L_loop
```

USING THE MYFLAG

If it fails in this search, the computer must fall back on the myflag (if it exists) or a random phrase (if myflag is an unassigned string):

```
800 REMark random replies/no trigger
810  IF K$<>"": GO TO 1010: REMark "MYFL
AG" is not empty, so go there
820  reply=RND(1,8)
830  SELect ON reply
850   =1: PRINT "WHAT DOES THAT SUGGEST
TO YOU?"
```

```
870    =2: PRINT "I'M NOT SURE I UNDERSTA
ND THAT FULLY"
890    =3: PRINT "PLEASE ELABORATE ON THA
T"
910    =4: PRINT "THAT'S VERY INTERESTING
"
930    =5: PRINT "WELL...PLEASE CONTINUE.
.."
950    =6: PRINT "WHY?"
970    =7: PRINT "AND THEN?"
990    =8: PRINT "I SEE...PLEASE TELL ME
MORE ON THAT"
995  END SELect
1000 consult
1010 REMark use "MYFLAG"
1020  response=RND(1,8)
1025  SELect ON response
1030   =1: PRINT "TELL ME MORE ABOUT YOU
R ";K$
1040   =2: PRINT "EARLIER YOU MENTIONED
YOUR ";K$: PRINT "TELL ME A BIT MORE ABO
UT THAT"
1050   =3: PRINT "DOES THAT HAVE ANYTHIN
G TO DO WITH YOUR ";K$;"?"
1060   =4: PRINT "IS THERE A LINK WITH Y
OUR ";K$;"?"
1070   =5: PRINT "WHY DON'T WE GO BACK A
ND DISCUSS YOUR ";K$;" A LITTLE MORE?"
1080   =6: PRINT "DOES ANY CONNECTION BE
TWEEN THAT AND YOUR ";K$;" SUGGEST ITSEL
F?"
1090   =7: PRINT "WOULD YOU PREFER TO TA
LK ABOUT YOUR ";K$;"?"
1100   =8: PRINT "I THINK PERHAPS WORRIE
S ABOUT YOUR ";K$;" ARE BOTHERING YOU"
1105  END SELect
```

```
1110  IF RND>.7: K$=""
1115  consult
1120 END DEFine consult
```

Line 810 checks to see what the string variable, K$, has been assigned to. If it finds that K$ is not empty, the action goes to the section from 1020 to 1110 and one of eight replies which can use the myflag is printed up (such as TELL ME MORE ABOUT YOUR . . . or WHY DON'T WE GO BACK AND DISCUSS YOUR . . . A LITTLE MORE?)

At the end of this section (line 1110), K$ is reset to the empty string around 30% of the time, thus allowing it to be reset if another MY is found in later input.

If K$ is unassigned, the DOCTOR chooses from the eight random replies (in lines 850 through to 1000). These are designed to keep the conversation flowing.

Now that you know how it works, it is time to hang up your computer's shingle, and go into practice.

```
10 REMark ==========
11 REMark = DOCTOR =
12 REMark ==========
15 :
20 initialise
22 consult
24 :
26 DEFine PROCedure consult
30  PRINT: PRINT "----------------------
--------------": PRINT
40  PRINT ">";: INPUT X$
50  REMark quit by pressing ENTER
52   IF X$=""
54    PRINT "program ended"
```

```
56     STOP
58    END IF
60    PRINT
70   REMark to control repetition
72    IF X$=Z$
74     PRINT "PLEASE DON'T REPEAT YOURSEL
F"
76     consult
78    ELSE
80     Z$=X$
85    END IF
90   REMark say goodbye
92    IF X$(1 TO 7)="GOODBYE"
94     PRINT "OK, SEE YOU AGAIN SOMETIME"
96     STOP
98    END IF
100 REMark look for trigger phrases at s
tart of input
110   L=0
115   REPeat loop_L
120    L=L+1
130    look=LEN(C$(L))
140    IF X$(1 TO look)=C$(L): GO TO 360:
 REMark trigger found
150    IF L>=K: EXIT loop_L
155   END REPeat loop_L
160 REMark program gets here if no trigg
er phrase found at start of X$
170 REMark now look for trigger words wi
thin input
180   X$=" " & X$ & " "
190   M=LEN(X$)
200   L=0
205   REPeat L_loop
210    L=L+1
```

```
220   IF L=M-1: GO TO 800: REMark no tri
gger found
230   IF X$(L)<>" ": NEXT L_loop
250   x=L+1
260   Y=0
265   REPeat Y_loop
270    Y=Y+1
280    IF X$(x+Y)=" "
282     Q$=X$(x TO x+Y-1)
284     EXIT Y_loop
286    END IF
290   END REPeat Y_loop
300   N=0
305   REPeat N_loop
310    N=N+1
320    IF Q$=C$(N): GO TO 710: REMark tr
igger word found
325    IF N>=K: EXIT N_loop
330   END REPeat N_loop
340  END REPeat L_loop
350 :
360 REMark trigger phrase found at start
 of input
370  trigger=RND(1,3)
375   SELect ON trigger
380    =1: G$=D$(L)
390    =2: G$=E$(L)
400    =3: G$=F$(L)
405   END SELect
410 REMark check to see if ends in aster
isk, if so needs part of input added
420  FLAG=0
430  IF "*" INSTR G$
432   FLAG=1
434   R=LEN(G$)
```

```
436   G$=G$(1 TO R-1)
438  END IF
440  PRINT G$;" ";
450  IF FLAG=0: consult: REMark no need
for added material
460 REMark now use balance of input
465  IF look+2>LEN(X$): consult
470  X$=" " & X$(look+2 TO) & " "
480 :
490 REMark conjugation changes
500 REMark also look for "MY" to trigger
 "MYFLAG"(K$)
510  look=LEN(X$)
520  M=0
525  REPeat M_loop
530   M=M+1
540   IF M=look: consult
550   IF X$(M)<>" ": NEXT M_loop
570   x=M+1
580   Y=0
585   REPeat Y_loop
590    Y=Y+1
600    IF X$(x+Y)=" "
602     Q$=X$(x TO x+Y-1)
604     EXIT Y_loop
606    END IF
610    IF x+Y>250: NEXT M_loop
620   END REPeat Y_loop
630   MN=0
635   REPeat mn_loop
640    MN=MN+1
650    IF Q$="MY" AND K$=""
652     K$=X$(x+3 TO LEN(X$)-1)
654    END IF
660    IF Q$=A$(MN)
```

```
662      PRINT B$(MN);" ";
664      NEXT M_loop
666     END IF
670     IF MN>=KK: EXIT mn_loop
675    END REPeat mn_loop
680    PRINT Q$;" ";
690   END REPeat M_loop
700 :
710 REMark trigger words found
720   trigger=RND(1,3)
730   Q$=""
735   SELect ON trigger
740     =1: Q$=D$(N)
750     =2: Q$=E$(N)
760     =3: Q$=F$(N)
765   END SELect
770   qq=LEN(Q$)
772    IF Q$(qq)<>"*"
774     PRINT Q$
776     consult
778    END IF
780 REMark falls through to next section
 if trigger word judged unsuitable
790 :
800 REMark random replies/no trigger
810   IF K$<>"": GO TO 1010: REMark "MYFL
AG" is not empty, so go there
820   reply=RND(1,8)
830   SELect ON reply
850     =1: PRINT "WHAT DOES THAT SUGGEST
TO YOU?"
870     =2: PRINT "I'M NOT SURE I UNDERSTA
ND THAT FULLY"
890     =3: PRINT "PLEASE ELABORATE ON THA
T"
```

```
910    =4: PRINT "THAT'S VERY INTERESTING
"
930    =5: PRINT "WELL...PLEASE CONTINUE.
.."
950    =6: PRINT "WHY?"
970    =7: PRINT "AND THEN?"
990    =8: PRINT "I SEE...PLEASE TELL ME
MORE ON THAT"
995  END SELect
1000 consult
1010 REMark use "MYFLAG"
1020  response=RND(1,8)
1025  SELect ON response
1030    =1: PRINT "TELL ME MORE ABOUT YOU
R ";K$
1040    =2: PRINT "EARLIER YOU MENTIONED
YOUR ";K$: PRINT "TELL ME A BIT MORE ABO
UT THAT"
1050    =3: PRINT "DOES THAT HAVE ANYTHIN
G TO DO WITH YOUR ";K$;"?"
1060    =4: PRINT "IS THERE A LINK WITH Y
OUR ";K$;"?"
1070    =5: PRINT "WHY DON'T WE GO BACK A
ND DISCUSS YOUR ";K$;" A LITTLE MORE?"
1080    =6: PRINT "DOES ANY CONNECTION BE
TWEEN THAT AND YOUR ";K$;" SUGGEST ITSEL
F?"
1090    =7: PRINT "WOULD YOU PREFER TO TA
LK ABOUT YOUR ";K$;"?"
1100    =8: PRINT "I THINK PERHAPS WORRIE
S ABOUT YOUR ";K$;" ARE BOTHERING YOU"
1105  END SELect
1110  IF RND>.7: K$=""
1115  consult
1120 END DEFine consult
```

```
1130 :
1140 DEFine PROCedure initialise
1150  RANDOMISE
1155  CLS #0
1160  BORDER 1,6
1165  PAPER 5
1170  CLS
1175  INK 1
1180  DIM A$(14,6),B$(14,6):REMark conju
gations
1190  DIM C$(45,15), D$(45,70), E$(45,70
), F$(45,70): REMark triggers words and
replies
1200  Z$="": REMark to stop repetitions
1210  K$="": REMark "MYFLAG"
1220  PRINT: PRINT
1230  PRINT " THE DOCTOR IS IN...PLEASE
STAND BY"
1235  RESTORE
1240  KK=0
1245  REPeat loop_kk
1250   KK=KK+1
1260   READ A$(KK),B$(KK)
1270   IF B$(KK)="*": EXIT loop_kk
1280  END REPeat loop_kk
1290  K=0
1295  REPeat loop_k
1300   K=K+1
1310   READ C$(K),D$(K),E$(K),F$(K)
1320   IF F$(K)="*": EXIT loop_k
1330  END REPeat loop_k
1340  CLS
1345  PRINT: PRINT
1350  PRINT " WELCOME TO ANOTHER SESSION
"
```

```
1360  PRINT " WITH THE DOCTOR....."
1370  PRINT
1380  PRINT " HELLO"
1390 END DEFine initialise
1400 :
1405 REMark ====
1410 REMark data
1415 REMark ====
1418 :
1420 REMark ** conjugations **
1430 DATA "AM","ARE","WAS","WERE","I","Y
OU","MY","YOUR"
1440 DATA "YOU'VE","I'VE","YOUR","MY","A
RE","AM"
1450 DATA "YOU'RE","I AM","YOURS","MINE"
,"YOU","ME","ME","YOU","*","*"
1460 REMark trigger words/reply phrases
1470 DATA "I NEED"
1480 DATA "WHY DO YOU NEED*"
1490 DATA "WOULD IT BE REALLY HELPFUL IF
 YOU      GOT*"
1500 DATA "ARE YOU SURE YOU NEED*"
1510 DATA "WHY DON'T YOU"
1520 DATA "DO YOU REALLY THINK I DON'T*"
1530 DATA "PERHAPS EVENTUALLY I WILL*"
1540 DATA "DO YOU REALLY WANT ME TO*"
1550 DATA "WHY CAN'T I"
1560 DATA "DO YOU THINK YOU SHOULD BE AB
LE        TO*"
1570 DATA "WHY CAN'T YOU*"
1580 DATA "PERHAPS YOU HAVEN'T TRIED HAR
D         ENOUGH"
1590 DATA "ARE YOU"
1600 DATA "WHY ARE YOU INTERESTED IN WHE
THER I    AM OR NOT*"
```

```
1610 DATA "WOULD YOU PREFER IT IF I WERE
      NOT*"
1620 DATA "PERHAPS YOU SOMETIMES DREAM I
 AM*"
1630 DATA "I CAN'T"
1640 DATA "HOW DO YOU KNOW YOU CAN'T*"
1650 DATA "HAVE YOU TRIED?"
1660 DATA "PERHAPS, NOW, YOU CAN*"
1670 DATA "I AM"
1680 DATA "DID YOU COME TO ME BECAUSE YO
U      ARE*"
1690 DATA "HOW LONG HAVE YOU BEEN*"
1700 DATA "DO YOU THINK IT'S ABSOLUTELY
NORMAL   TO BE*"
1710 DATA "I'M"
1720 DATA "DO YOU ENJOY BEING*"
1730 DATA "WHY TELL ME YOU'RE*"
1740 DATA "WHY ARE YOU*"
1750 DATA "I WANT"
1760 DATA "WHAT WOULD IT MEAN TO YOU IF
YOU      GOT*"
1770 DATA "WHY DO YOU WANT*"
1780 DATA "WHAT WOULD IT ADD TO YOUR LIF
E IF     YOU GOT*"
1790 DATA "WHAT"
1800 DATA "WHY DO YOU ASK?"
1810 DATA "HOW WOULD AN ANSWER TO THAT H
ELP      YOU?"
1820 DATA "WHAT DO YOU THINK?"
1830 DATA "HOW"
1840 DATA "HOW WOULD YOU SOLVE THAT?"
1850 DATA "IT WOULD BE BEST TO ANSWER TH
AT FOR   YOURSELF"
1860 DATA "WHAT IS IT YOU'RE REALLY ASKI
NG?"
```

```
1870 DATA "WHO"
1880 DATA "DO YOU OFTEN THINK ABOUT SUCH
      QUESTIONS"
1890 DATA "WHAT ANSWER WOULD PUT YOUR MI
ND AT    REST"
1900 DATA "WHO DO YOU THINK*"
1910 DATA "WHERE"
1920 DATA "THAT'S A PRETTY SILLY QUESTIO
N"
1930 DATA "DO YOU REALLY NEED TO KNOW WH
ERE*"
1940 DATA "WHAT WOULD IT MEAN TO YOU IF
I TOLD   YOU WHERE*"
1950 DATA "WHEN"
1960 DATA "HOW SHOULD I KNOW WHEN*"
1970 DATA "THE TIME SHOULD NOT BE DISCUS
SED      HERE"
1980 DATA "THINGS HAVE A HABIT OF HAPPEN
ING AT   MORE OR LESS THE RIGHT TIME"
1990 DATA "WHY"
2000 DATA "WHY DON'T YOU TELL ME THE REA
SON      WHY*"
2010 DATA "WHAT HAVE YOU TOLD ME WHICH W
OULD     ALLOW ME TO TELL YOU WHY*"
2020 DATA "DO YOU REALLY NEED TO KNOW WH
Y*"
2030 DATA "BECAUSE"
2040 DATA "IS THAT THE REAL REASON?"
2050 DATA "WHAT OTHER REASONS COME TO MI
ND?"
2060 DATA "WHAT ELSE DOES THAT EXPLAIN?"
2070 DATA "SORRY"
2080 DATA "IN WHAT OTHER CIRCUMSTANCES D
O YOU    APOLOGIZE?"
2090 DATA "THERE ARE MANY TIMES WHEN NO
```

```
APOLOGY  IS NEEDED"
2100 DATA "WHAT FEELINGS DO YOU HAVE WHE
N YOU     APOLOGIZE?"
2110 DATA "HELLO"
2120 DATA "HELLO...IT'S GOOD TO SEE YOU"
2130 DATA "HELLO TO YOU...I'M GLAD YOU C
OULD      DROP BY TODAY"
2140 DATA "HOW ARE YOU...I'M LOOKING FOR
WARD TO  ANOTHER CHAT WITH YOU"
2150 DATA "HI"
2160 DATA "HI THERE...I'M GLAD TO SEE YO
U HERE    TODAY"
2170 DATA "HI. I'M GLAD YOU DROPPED BY..
.         WE'VE GOT LOTS OF TIME TO CHAT"
2180 DATA "HI TO YOU...RELAX NOW, AND LE
T'S       TALK ABOUT YOUR SITUATION"
2190 DATA "MAYBE"
2200 DATA "YOU SEEM A LITTLE HESITANT"
2210 DATA "THAT'S PRETTY INDECISIVE"
2220 DATA "IN WHAT OTHER SITUATIONS DO Y
OU        SHOW SUCH A TENTATIVE APPROACH?
"
2230 DATA "NO"
2240 DATA "WHY ARE YOU BEING SO NEGATIVE
          ABOUT IT?"
2250 DATA "ARE YOU SAYING THAT JUST TO B
E         NEGATIVE"
2260 DATA "THAT'S PRETTY FORCEFUL. WHAT
DOES      IT SUGGEST TO YOU?"
2270 DATA "ALWAYS"
2280 DATA "PLEASE GIVE ME A SPECIFIC EXA
MPLE"
2290 DATA "ISN'T 'ALWAYS' A LITTLE STRON
G?"
2300 DATA "WHEN?"
```

```
2310 DATA "I THINK"
2320 DATA "DO YOU DOUBT*"
2330 DATA "DO YOU REALLY THINK SO?"
2340 DATA "BUT YOU ARE NOT SURE*"
2350 DATA "FRIEND"
2360 DATA "WHY DO YOU BRING UP THE SUBJE
CT       OF FRIENDS?"
2370 DATA "WHAT IS YOUR BEST MEMORY OF
         A FRIEND?"
2380 DATA "PLEASE TELL ME MORE ABOUT YOU
R        FRIENDSHIP..."
2390 DATA "FRIENDS"
2400 DATA "IN WHAT WAY DO YOUR FRIENDS'
         REACTIONS BOTHER YOU?"
2410 DATA "WHAT MADE YOU START TO TALK A
BOUT     FRIENDS JUST NOW?"
2420 DATA "IN WHAT WAY DO YOUR FRIENDS
         IMPOSE ON YOU?"
2430 DATA "YES"
2440 DATA "WHAT LIES BEHIND YOUR CERTAIN
TY?"
2450 DATA "YOU'RE PRETTY POSITIVE"
2460 DATA "YOU SEEM VERY CERTAIN ABOUT I
T"
2470 DATA "COMPUTER"
2480 DATA "WHAT FEELINGS DO YOU GET, SIT
TING     THERE TALKING TO ME LIKE THIS?"
2490 DATA "ARE YOU THINKING ABOUT ME IN
         PARTICULAR"
2500 DATA "WHAT ASPECT OF COMPUTERS INTE
RESTS    YOU MOST?"
2510 DATA "IS IT"
2520 DATA "DO YOU THINK IT IS*"
2530 DATA "IN WHAT CIRCUMSTANCES WOULD I
T*"
```

```
2540 DATA "IT COULD WELL BE THAT*"
2550 DATA "IT IS"
2560 DATA "WHAT DEGREE OF CERTAINTY WOUL
D          YOU PLACE ON IT BEING*"
2570 DATA "ARE YOU CERTAIN THAT IT'S*"
2580 DATA "WHAT EMOTIONS WOULD YOU FEEL
IF         I TOLD YOU IT PROBABLY ISN'T*"
2590 DATA "CAN YOU"
2600 DATA "WHAT MAKES YOU THINK I CAN'T*
"
2610 DATA "DON'T YOU THINK THAT I CAN*"
2620 DATA "PERHAPS YOU WOULD LIKE TO BE
          ABLE TO*"
2630 DATA "CAN I"
2640 DATA "PERHAPS YOU DON'T WANT TO*"
2650 DATA "DO YOU WANT TO BE ABLE TO*"
2660 DATA "I DOUBT IT"
2670 DATA "YOU ARE"
2680 DATA "WHY DO YOU THINK I AM*"
2690 DATA "DOES IT PLEASE YOU TO BELIEVE
          I AM*"
2700 DATA "PERHAPS YOU WOULD LIKE TO BE*
"
2710 DATA "YOU'RE"
2720 DATA "WHY DO YOU THINK I AM*"
2730 DATA "DOES IT PLEASE YOU TO BELIEVE
          I AM*"
2740 DATA "WHY DO YOU SAY I'M*"
2750 DATA "I DON'T"
2760 DATA "DON'T YOU REALLY*"
2770 DATA "WHY DON'T YOU*"
2780 DATA "DO YOU WANT TO BE ABLE TO*"
2790 DATA "I FEEL"
2800 DATA "TELL ME MORE ABOUT SUCH FEELI
NGS"
```

```
2810 DATA "DO YOU OFTEN FEEL*"
2820 DATA "DO YOU ENJOY FEELING*"
2830 DATA "FEEL"
2840 DATA "LET'S EXPLORE THAT STATEMENT
A BIT"
2850 DATA "DO YOU OFTEN FEEL LIKE THAT?"
2860 DATA "WHAT EMOTIONS DO SUCH FEELING
S        STIR UP IN YOU?"
2870 DATA "I HAVE"
2880 DATA "WHY TELL ME THAT YOU'VE*"
2890 DATA "IT'S OBVIOUS TO ME THAT YOU H
AVE*"
2900 DATA "HOW CAN I HELP YOU WITH*"
2910 DATA "I WOULD"
2920 DATA "COULD YOU EXPLAIN WHY YOU WOU
LD*"
2930 DATA "WHO ELSE HAVE YOU TOLD YOU WO
ULD*"
2940 DATA "HOW SURE ARE YOU THAT YOU WOU
LD*"
2950 DATA "IS THERE"
2960 DATA "OF COURSE THERE IS*"
2970 DATA "IT'S LIKELY THAT THERE IS*"
2980 DATA "WOULD YOU LIKE THERE TO BE*"
2990 DATA "MY"
3000 DATA "YOUR*"
3010 DATA "I SEE, YOUR*"
3020 DATA "WHAT DOES IT MEAN TO YOU, THA
T        YOUR*"
3030 DATA "YOU"
3040 DATA "THIS SESSION IS TO HELP YOU..
         NOT TO DISCUSS ME!"
3050 DATA "WHAT PROMPTED YOU TO SAY THAT
         ABOUT ME?"
3060 DATA "REMEMBER I'M TAKING NOTES ON
```

```
ALL       THIS TO SOLVE YOUR SITUATION"
3070 DATA "*","*","*","*"
3080 :
3090 REMark ==========
3100 REMark END DOCTOR
3110 REMark ==========
```

CHAPTER NINE — MACHINE TRANSLATION

It would seem — when thinking about some of the possibilities that arise from machines being able to understand and process natural language — that computers could be of great value in helping us translate from one human language to another. Such a hope has been with us since the early fifties, and a great deal of progress has been made in the field.

There are now more than 20 machine translation (MT) systems in use around the world. But, contrary to popular opinion, these systems do not 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 kind 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 two paragraphs 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 trans-

lation of literary works is another field entirely, and in so far as MT is concerned, is barely in its infancy.

FRANGLAIS

This 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.

Here's some of the output of TRANSLATE:

```
? 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


? If you turn left at Plains in Georgia
you will end up in Paris
 —>IF VOUS TURN GAUCHE AT PARIS DANS
GEORGIA VOUS WILL END UP DANS PLAINS
```

As you can see, I've deliberately fed in English that triggers the program's extremely limited vocabulary:

```
? I want some cigarettes to put behind
the door in my house
 —>JE WANT DES GAULOISES A PUT DERRIERE
 LE PORTE DANS MON MAISON

? Pass the medicine for my mother which
she must have when she is cold
```

```
 —>PASS LE VIN POUR MON MERE WHICH SHE
MUST AI QUAND SHE EST FROID

? When you play music I want to sing a
song and wave my arm in the sun
 —>QUAND VOUS PLAY MUSIQUE JE WANT A
SING UNE CHANSON ET WAVE MON BRA DANS LE
 SOLIEL

? Fish and chips are for those who stand
 in the eye of the public
 —>POISSON ET CHIPS EST POUR THOSE WHO
STAND DANS LE ORIEL DE LE PUBLIC

? I am feeling right inside my head when
 i make music behind the little cat
 —>JE SUIS FEELING DROITE INSIDE MON
TETE QUAND JE MAKE MUSIQUE DERRIERE LE
PETITE CHAT
```

PROGRAM STRUCTURE

The program is simple to follow. It starts with (as usual) a call to a subroutine at the end of the program which initialises the variables.

```
10 REMark =========
11 REMark TRANSLATE
12 REMark =========
15 :
20 initialise
```

```
400 DEFine PROCedure initialise
410  CLS #0
412  BORDER 12,2
414  PAPER 7
416  INK 1
420  CLS
430  DIM E$(100,15): REMark to hold Engl
ish
440  DIM F$(100,15): REMark to hold Fren
ch
445  RESTORE
450  count=0
455  REPeat count_loop
460   count=count+1
470   READ E$(count), F$(count)
480   IF F$(count)="*": EXIT count_loop
485  END REPeat count_loop
490 END DEFine initialise
495 :
500 REMark ====
502 REMark data
504 REMark ====
506 :
510 DATA "THE","LE","ME","MOI","I","JE",
"HERE","ICI","AM","SUIS","ARE","EST","NO
T","NE","IN","DANS"
520 DATA "WHEN","QUAND","YOU","VOUS","IS
","EST","IT","IL","DAY","JOUR","AND","ET
","SOME","DES","OF","DE"
530 DATA "HAVE","AI","A","UNE","MY","MON
","YOUR","VOTRE","OF","DE","TO","A","SEE
","VU","VERY","TRES"
540 DATA "ROOM","CHAMBRE","STEAK","ENTRE
COTE","FRIES","POMMES FRITES","BIG","GRA
ND","FOR","POUR"
```

```
550 DATA "MATCH","ALLUMETTE","SUPER","FA
NTASTIQUE","DEAD","MORT","WITH","AVEC"
560 DATA "GIN","VIN","WHISKEY","VIN","WH
ISKY","VIN","BEER","VIN","MARTINI","VIN"
,"WINE","VIN"
570 DATA "PARIS","PLAINS","PLAINS","PARI
S","HAIR","CHEVAUX","CIGARETTES","GAULOI
SES"
580 DATA "ARM","BRA","LEG","JAMBE","RIGH
T","DROITE","LEFT","GAUCHE"
590 DATA "TRENDY","AVANT-GARDE","MEDICIN
E","VIN","POLICEMAN","GENDARME"
600 DATA "DETECTIVE","CLUESO","DOOR","PO
RTE","HEAD","TETE","LOVE","AMOUR"
610 DATA "HOUSE","MAISON","CHAIR","CHAIS
E","EYE","ORIEL","SUN","SOLIEL"
620 DATA "SONG","CHANSON","FRIENDS","AMI
S","BEHIND","DERRIERE","SEA","MER","MOTH
ER","MERE"
630 DATA "CAT","CHAT","DOG","CHIEN","BLU
E","BLEU","LITTLE","PETITE"
640 DATA "MUSIC","MUSIQUE","PLEASE","S'I
L VOUS PLAIT","BOY","GARCON","GIRL","FIL
LE"
650 DATA "FISH","POISSON","CHICKEN","POU
LET","DUCK","CANARD","MUSTARD","MOUTARDE
"
660 DATA "HOT","CHAUD","COLD","FROID","E
VERYBODY","TOUT LE MONDE"
670 DATA "HELLO","BONJOUR","GOOD","BON"
680 DATA "*","*"
```

In this subroutine, E$ is used to hold the English text, the F$ contains the equivalent French. The French equivalent of E$(4) (here) is F$(4) (ici) and so on, which makes it extremely easy to use.

The variable COUNT counts the number of words fed into the system. The arrays have been dimensioned to hold more words than are currently in the vocabulary, so you can add your own (perhaps translating your name into SUPERSTAR or the like).

On returning from the initialisation subroutine, the program accepts the user input (line 30) and then checks to see if this is the empty string (that is, the user has simply pressed RETURN without entering any text). If it finds that the input, A$, is empty, the program ends.

```
30  INPUT '>'!A$: REMark accept user inp
ut
40  IF A$="": EXIT main_loop
50  upper_case
60  translate
70 END REPeat main_loop
```

Line 50 sends the text to the subroutine from 290 which converts user input to upper case, and then the subroutine from line 100 makes the actual translation. Line 70 sends the program back to 20 to accept more input.

This is the routine which converts the user input into upper case:

```
290 DEFine PROCedure upper_case
300  A$=" "&A$&" "
310  B$=""
320  l=LEN(A$)
330  FOR J=1 TO l
340   k=CODE(A$(J))
350   IF k>96 AND k<124: k=k-32
360   B$=B$&CHR$(k)
370  END FOR J
380 END DEFine upper_case
```

It simply goes through the text, element by element, converting any character it finds whose ASCII code is between 97 and 123 (that is, the lower case letters) to their upper case equivalent (by subtracting 32). Once all the user input has been converted into upper case, the actual translation can begin:

```
100 DEFine PROCedure translate
110  PRINT "  -->";
120  k=0
125  REPeat k_loop
130   k=k+1
140   IF k=1: PRINT: RETurn
150   IF B$(k)<>" "
160    NEXT k_loop
165   ELSE
170    x=k+1
175   END IF
180   y=0
185   REPeat y_loop
190    y=y+1
200    IF B$(x+y)=" "
202     Q$=B$(x TO x+y-1)
204     EXIT y_loop
206    END IF
210   END REPeat y_loop
220   m=0
225   REPeat m_loop
230    m=m+1
240    IF Q$=E$(m)
242     PRINT !F$(m)!
244     NEXT k_loop
246    END IF
250    IF m>=count: EXIT m_loop
255   END REPeat m_loop
260   PRINT !Q$!
270  END REPeat k_loop
```

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 300 — 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 line 170 which continues to search for the next space, so it can isolate the whole word. Then it simply runs through the vocabulary (lines 220 to 270) until it finds a match.

```
? My mother went to see the sea with a
very blue dog and a hot chicken supper
  —>MON MERE WENT A VU LE MER AVEC UNE
 TRES BLEU CHIEN ET UNE CHAUD POULET
 SUPPER
```

If it does find such a match, the French word is printed in place of the English one, and the program returns to 130 to continue the search. Note that once a match has been found, the program immediately reverts to 130. 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.

Time now for you to experience a little MT of your own with TRANSLATE:

```
10 REMark =========
11 REMark TRANSLATE
12 REMark =========
15 :
20 initialise
25 REPeat main_loop
30  INPUT '>'!A$: REMark accept user inp
ut
```

```
40  IF A$="": EXIT main_loop
50  upper_case
60  translate
70 END REPeat main_loop
80 PRINT: PRINT "Finis": STOP
85 :
90 REMark ==========
92 REMark definitions
94 REMark ==========
96 :
100 DEFine PROCedure translate
110  PRINT "  -->";
120  k=0
125  REPeat k_loop
130   k=k+1
140   IF k=1: PRINT: RETurn
150   IF B$(k)<>" "
160    NEXT k_loop
165   ELSE
170    x=k+1
175   END IF
180   y=0
185   REPeat y_loop
190    y=y+1
200    IF B$(x+y)=" "
202     Q$=B$(x TO x+y-1)
204     EXIT y_loop
206    END IF
210   END REPeat y_loop
220   m=0
225   REPeat m_loop
230    m=m+1
240    IF Q$=E$(m)
242     PRINT !F$(m)!
244     NEXT k_loop
246    END IF
```

```
250     IF m>=count: EXIT m_loop
255    END REPeat m_loop
260    PRINT !Q$!
270   END REPeat k_loop
280 END DEFine translate
285 :
290 DEFine PROCedure upper_case
300   A$=" "&A$&" "
310   B$=""
320   l=LEN(A$)
330   FOR J=1 TO l
340     k=CODE(A$(J))
350     IF k>96 AND k<124: k=k-32
360     B$=B$&CHR$(k)
370   END FOR J
380 END DEFine upper_case
390 :
400 DEFine PROCedure initialise
410   CLS #0
412   BORDER 12,2
414   PAPER 7
416   INK 1
420   CLS
430   DIM E$(100,15): REMark to hold Engl
ish
440   DIM F$(100,15): REMark to hold Fren
ch
445   RESTORE
450   count=0
455   REPeat count_loop
460     count=count+1
470     READ E$(count), F$(count)
480     IF F$(count)="*": EXIT count_loop
485   END REPeat count_loop
490 END DEFine initialise
495 :
```

```
500 REMark ====
502 REMark data
504 REMark ====
506 :
510 DATA "THE","LE","ME","MOI","I","JE",
"HERE","ICI","AM","SUIS","ARE","EST","NO
T","NE","IN","DANS"
520 DATA "WHEN","QUAND","YOU","VOUS","IS
","EST","IT","IL","DAY","JOUR","AND","ET
","SOME","DES","OF","DE"
530 DATA "HAVE","AI","A","UNE","MY","MON
","YOUR","VOTRE","OF","DE","TO","A","SEE
","VU","VERY","TRES"
540 DATA "ROOM","CHAMBRE","STEAK","ENTRE
COTE","FRIES","POMMES FRITES","BIG","GRA
ND","FOR","POUR"
550 DATA "MATCH","ALLUMETTE","SUPER","FA
NTASTIQUE","DEAD","MORT","WITH","AVEC"
560 DATA "GIN","VIN","WHISKEY","VIN","WH
ISKY","VIN","BEER","VIN","MARTINI","VIN"
,"WINE","VIN"
570 DATA "PARIS","PLAINS","PLAINS","PARI
S","HAIR","CHEVAUX","CIGARETTES","GAULOI
SES"
580 DATA "ARM","BRA","LEG","JAMBE","RIGH
T","DROITE","LEFT","GAUCHE"
590 DATA "TRENDY","AVANT-GARDE","MEDICIN
E","VIN","POLICEMAN","GENDARME"
600 DATA "DETECTIVE","CLUESO","DOOR","PO
RTE","HEAD","TETE","LOVE","AMOUR"
610 DATA "HOUSE","MAISON","CHAIR","CHAIS
E","EYE","ORIEL","SUN","SOLIEL"
620 DATA "SONG","CHANSON","FRIENDS","AMI
S","BEHIND","DERRIERE","SEA","MER","MOTH
ER","MERE"
```

```
630 DATA "CAT","CHAT","DOG","CHIEN","BLU
E","BLEU","LITTLE","PETITE"
640 DATA "MUSIC","MUSIQUE","PLEASE","S'I
L VOUS PLAIT","BOY","GARCON","GIRL","FIL
LE"
650 DATA "FISH","POISSON","CHICKEN","POU
LET","DUCK","CANARD","MUSTARD","MOUTARDE
"
660 DATA "HOT","CHAUD","COLD","FROID","E
VERYBODY","TOUT LE MONDE"
670 DATA "HELLO","BONJOUR","GOOD","BON"
680 DATA "*","*"
690 :
700 REMark =============
710 REMark END TRANSLATE
720 REMark =============
```

CHAPTER TEN — HANSHAN

Our final program in this section on language handling creates random poems. This is a pretty trivial program, and one which — you may argue — hardly gives evidence of the presence of artificial intelligence.

However, imagine you were reading a book like this 30 years ago. The author makes a casual remark about a lowcost 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):

```
370 REMark single words
380 DATA "scurrying","treading","gazing"
,"withered","chiselled"
390 DATA "muffled","flanked","writhed","
bending","twisting"
400 DATA "hammered","hanging","winding",
```

```
"clearest","weary"
410 DATA "earthward","cataract","sacrifi
cial","slippery","asunder"
415 :
420 REMark 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"
```

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):

```
30 REPeat choose_pattern
40   pattern=RND(1,3)
50   SELect ON pattern
```

```
 52    =1: pattern_1
 54    =2: pattern_2
 56    =3: pattern_3
 58   END SELect
 60   PAUSE 100
 70   PRINT:PRINT:PRINT
 80 END REPeat choose_pattern
 82 :
 84 REMark ==========
 85 REMark definitions
 86 REMark ==========
 88 :
 90 DEFine PROCedure pattern_1
100   PRINT word$(RND(1,20));"...";word$(
RND(1,20))
110   PRINT"    ";"....";word$(RND(1,20))
120   PRINT"       ";phrase$(RND(1,20))
130 END DEFine pattern_1
135 :
140 DEFine PROCedure pattern_2
150   PRINT phrase$(RND(1,20))
160   PRINT"    ";phrase$(RND(1,20));"..."
170   PRINT"       ";phrase$(RND(1,20))
180 END DEFine pattern_2
185 :
190 DEFine PROCedure pattern_3
200   PRINT" ";word$(RND(1,20))
210   PRINT phrase$(RND(1,20))
220   PRINT" ";word$(RND(1 TO 20));", ";p
hrase$(RND(1,20))
230 END DEFine pattern_3
```

Some of the poems produced by HANSHAN have a surprising degree of merit:

SULLEN, SULLEN
I TAKE YOUR POEMS...
MEN OF LEARNING

WHEN SHALL WE MEET
FOR THESE FEW STEPS...
IN THE BLACK DARKNESS

HANGING...TWISTING
....WRITHED
THOSE THAT ARE LEFT

WHY SHOULD YOU WASTE
WAVES OF COOLNESS...
NOW AT DUSK

MEN OF LEARNING
MEN OF ACTION...
AND MUCH GRIEVING

WEARY
IN THE BLACK DARKNESS
SCURRYING, SULLEN, SULLEN

Here is the HANSHAN listing to enable you to create a nearly infinite sequence of poems. By all means modify the DATA statements to make the program (and its output) your own:

```
 10 REMark =======
 12 REMark HANSHAN
 14 REMark =======
 16 :
 20 initialise
 30 REPeat choose_pattern
 40  pattern=RND(1,3)
 50  SELect ON pattern
 52   =1: pattern_1
 54   =2: pattern_2
 56   =3: pattern_3
 58  END SELect
 60  PAUSE 100
 70  PRINT:PRINT:PRINT
 80 END REPeat choose_pattern
 82 :
 84 REMark ===========
 85 REMark definitions
 86 REMark ===========
 88 :
 90 DEFine PROCedure pattern_1
100  PRINT word$(RND(1,20));"...";word$(
RND(1,20))
110  PRINT"    ";"....";word$(RND(1,20))
120  PRINT"       ";phrase$(RND(1,20))
130 END DEFine pattern_1
135 :
140 DEFine PROCedure pattern_2
150  PRINT phrase$(RND(1,20))
160  PRINT"    ";phrase$(RND(1,20));"..."
170  PRINT"       ";phrase$(RND(1,20))
```

```
180 END DEFine pattern_2
185 :
190 DEFine PROCedure pattern_3
200  PRINT" ";word$(RND(1,20))
210  PRINT phrase$(RND(1,20))
220  PRINT" ";word$(RND(1 TO 20));", ";p
hrase$(RND(1,20))
230 END DEFine pattern_3
240 :
250 DEFine PROCedure initialise
252  CLS #0
254  BORDER 2,4
256  PAPER 6
258  INK 1
260  CLS
270  RANDOMISE
280  DIM word$(20,12)
282  DIM phrase$(20,25)
284  RESTORE
290  FOR w=1 TO 20
300   READ word$(w)
310  END FOR w
320  FOR p=1 TO 20
330   READ phrase$(p)
340  END FOR p
350 END DEFine initialise
355 :
360 REMark ====
362 REMark data
364 REMark ====
366 :
370 REMark single words
380 DATA "scurrying","treading","gazing"
,"withered","chiselled"
390 DATA "muffled","flanked","writhed","
```

```
bending","twisting"
400 DATA "hammered","hanging","winding",
"clearest","weary"
410 DATA "earthward","cataract","sacrifi
cial","slippery","asunder"
415 :
420 REMark short phrases
430 DATA "in the cool stream"
440 DATA "nodded in clustered grace"
450 DATA "waves of coolness"
460 DATA "out from the deepest"
470 DATA "sullen, sullen"
480 DATA "in the black darkness"
490 DATA "I take your poems"
500 DATA "I put out the lamp"
510 DATA "my short span runs out"
520 DATA "those that are left"
530 DATA "men of learning"
540 DATA "men of action"
550 DATA "I hurry forward"
560 DATA "why should you waste?"
570 DATA "when shall we meet?"
580 DATA "little sleeping"
590 DATA "and much grieving"
600 DATA "for these few steps"
610 DATA "now at dusk"
620 DATA "I have done with profit"
630 :
640 REMark ===========
650 REMark END HANSHAN
660 REMark ===========
```

SECTION FOUR — HELPING

CHAPTER ELEVEN — EXPERT SYSTEMS

There is a limited number of experts in the world on any one subject. It doesn't matter what field you're talking about — mending cars, mining for uranium, diagnosing human illness, sorting edible mushrooms from poisonous ones — there is a limit to the number of experts we have available.

Now while the world is not exactly crying out for more mushroom-sorting experts, there are areas of the world (most of it in fact) where there are not enough doctors. One idea of an expert system is to 'capture' the expertise of one of our experts on a computer, in such a way that a non-expert can tap the information.

Expert systems is the one area of 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 one would hardly need an expert system to make a diagnosis like this (and note that I am not suggesting the

diagnosis of my IF/THEN chain is necessarily correct). An expert system comes into its own when either of the following conditions exist:

— the expert is not present but his or her expertise is;

— even the 'expert' doesn't know with 100% certainty the casual links between the observations and the results. This could happen if a medical researcher was aware that patients contracting disease X have tended to have had contact with foods A and B and have blood group C . . . although no way of linking A, B and C — apart from the fact that they appear together — had been discovered. In this case, a properly programmed expert system could make predictions about the likelihood of individual D contracting the disease, even when the percentage contribution that factors A, B and C made were unknown. By studying enough cases, the expert system could not only devise its own rules for predicting whether a particular individual would, or would not, contract the disease, but could then explain its reasoning to a human physician.

In the section of the book on learning and reasoning we talked about 'logic circuits' and discussed the way these made decisions, according to the rules of Boolean Algebra.

The 'mathematics of reasoning' is 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 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 had 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 Sir,' 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 necktie. An idiot savant status is a 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 despite 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 own 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 are 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.

—TAXMAN: Under development at Rutgers University, this system is intended to examine changing tax rules, and from them give advice to companies on how to best operate within those rules.

—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.

CHAPTER 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 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 pauses for a nanosecond or two, then prints the following message on the screen:

```
THANK YOU. YOUR LIFE EXPECTANCY IS 79 YEARS, THUS
EXCEEDING 11% OF THE POPULATION. TO INCREASE YOUR
CHANCES OF REACHING, OR EXCEEDING THIS, I SUGGEST
        YOU - TRY TO STOP SMOKING
            - GET REGULAR MEDICAL CHECKUPS
            - INCREASE YOUR EXERCISE EACH WEEK

          THANK YOU FOR CONSULTING MEDICI
```

What did MEDICI do? How did it turn your YES/NO answers into a life expectancy prediction? Actually, as I'm sure you've already decided, this is not a very sophisticated program, and would not demand a very high level of expertise. However, it shows how a real medical diagnosis program might begin, if the expert system was interacting directly with a patient, rather than with a physician as is generally the case at present.

Pleased that you're going to live longer than 11% of the population, you settle down to make the acquaintance of another expert, young SPURT. Here's what you see on your screen:

```
I WANT YOU TO THINK OF A MAN, A HORSE
     OR A SPARROW


DOES IT HAVE TWO LEGS
     Y OR N? Y
```

```
CAN IT WALK
      Y OR N? Y

CAN IT FLY
      Y OR N? N

YOU WERE THINKING OF A MAN

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

PRESS 'RETURN' FOR ANOTHER ONE, OR
ANY KEY AND THEN 'RETURN' TO QUIT
?
```

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:

```
I WANT YOU TO THINK OF A MAN, A HORSE
     OR A SPARROW


DOES IT HAVE TWO LEGS
      Y OR N? Y

CAN IT WALK
      Y OR N? Y

CAN IT FLY
      Y OR N? Y

YOU WERE THINKING OF A SPARROW
```

Surely it couldn't do it again, you think, and try for the third alternative:

```
I WANT YOU TO THINK OF A MAN, A HORSE
     OR A SPARROW


DOES IT HAVE TWO LEGS
      Y OR N? N

CAN IT WALK
      Y OR N? Y

CAN IT FLY
      Y OR N? N

YOU WERE THINKING OF A HORSE
```

This time you decide to quit. How does SPURT record the answers to your questions so it can determine that if you said the creature you were thinking of had two legs and could walk, but could not fly, was a man? How, for that matter, 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.

MEDICI not only counted your answers, but noted which question they referred to. A 'Y' to DO YOU SMOKE? clocked three years off your expectancy, while a 'Y' answer to DO YOU TAKE VIGOROUS EXERCISE REGULARLY? added five years to your expected span.

The SPURT listing starts as follows:

```
20 initialise
25 REPeat main_loop
30  PRINT: PRINT "I want you to think of
"\"  a man"\"     a horse"\"       or a sp
arrow"
50  PAUSE 200
60  PRINT: PRINT
70  ask_questions
```

After setting the scene, the real business of determining which creature you're thinking about begins:

```
170 DEFine PROCedure ask_questions
180  count=0
190  PRINT "Does it have two legs?"
200  process_answer
210  PRINT "Can it walk?"
220  process_answer
230  PRINT "Can it fly?"
240  process_answer
250  PRINT "You were thinking of a ";
255  SELect ON count
260   =1: PRINT "horse"
270   =2: PRINT "man"
280   =3: PRINT "sparrow"
285  END SELect
290 END DEFine ask_questions
300 :
310 DEFine PROCedure process_answer
```

```
315  REPeat control
320   INPUT "       y or n   ";z$
330   IF z$="y" OR z$="n"
332    EXIT control
334   END IF
335  END REPeat control
340  IF z$="y": count=count+1
350  PRINT
360 END DEFine
```

You'll see that the variable COUNT is set to zero at the start of the run, and incremented by one each time a 'Y' answer is given. Using this information, SPURT has no trouble deciding which creature you're thinking about:

```
250 DEFine PROCedure initialise
252  CLS #0
254  BORDER 2,4
256  PAPER 6
258  INK 1
260  CLS
270  RANDOMISE
280  DIM word$(20,12)
282  DIM phrase$(20,25)
284  RESTORE
290  FOR w=1 TO 20
```

As you can see, it is a pretty simple program, but one which lays a foundation upon which expert systems could be built. Here's the complete listing:

```
10 REMark =====
12 REMark SPURT
14 REMark =====
16 :
20 initialise
25 REPeat main_loop
```

```
30  PRINT: PRINT "I want you to think of
"\"  a man"\"     a horse"\"       or a sp
arrow"
50  PAUSE 200
60  PRINT: PRINT
70  ask_questions
80  PRINT
90  PRINT "-----------------------------
------"
100  PRINT: PRINT "Press ENTER for anoth
er one , or"
110  PRINT "any key and then ENTER to qu
it"
120  INPUT q$
130  IF q$<>"": EXIT main_loop
140  CLS
150 END REPeat main_loop
152 STOP
154 :
160 REMark ===========
162 REMark definitions
164 REMark ===========
166 :
170 DEFine PROCedure ask_questions
180  count=0
190  PRINT "Does it have two legs?"
200  process_answer
210  PRINT "Can it walk?"
220  process_answer
230  PRINT "Can it fly?"
240  process_answer
250  PRINT "You were thinking of a ";
255  SELect ON count
260   =1: PRINT "horse"
270   =2: PRINT "man"
280   =3: PRINT "sparrow"
```

```
285  END SELect
290 END DEFine ask_questions
300 :
310 DEFine PROCedure process_answer
315  REPeat control
320   INPUT "       y or n   ";z$
330   IF z$="y" OR z$="n"
332    EXIT control
334   END IF
335  END REPeat control
340  IF z$="y": count=count+1
350  PRINT
360 END DEFine
370 :
380 DEFine PROCedure initialise
390  CLS #0
400  BORDER 7,4
410  PAPER 1
420  INK 7
430  CLS
440 END DEFine initialise
450 :
460 REMark =========
470 REMark END SPURT
480 REMark =========
```

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 looked at, it is considerably more sophisticated.

You can see this increased sophistication by looking at a sample run from it. Firstly, we will get it to perform much as SPURT did. However, you can tell from the opening frame that this is a rather different program. It is largely 'soft', that is the expertise is not hardwired as in the case of SPURT but can be entered differently for each run.

```
NAME OF SYSTEM? CREATURES

NUMBER OF OUTCOMES? 3

NUMBER OF FACTORS TO BE CONSIDERED? 3
```

You tell the program its subject matter (CREATURES in this case), and then the number of OUTCOMES (that is, results) and the number of FACTORS TO BE CONSIDERED. These are the variables (such as CAN IT FLY) which must be considered. Having given it the framework, X-SPURT now asks you to fill in the outlines:

```
CREATURES

WHAT IS OUTCOME 1 ? MAN

WHAT IS OUTCOME 2 ? HORSE

WHAT IS OUTCOME 3 ? SPARROW
```

Having told it the outcomes, it asks you to enter the questions which relate to the factors which determine which outcome you are seeking:

```
PLEASE ENTER QUESTION 1
? DOES IT FLY UNAIDED

PLEASE ENTER QUESTION 2
? DOES IT HAVE TWO LEGS

PLEASE ENTER QUESTION 3
? DOES IT WALK
```

This may seem like a lot of trouble we're going to, just to emulate SPURT, but — as you'll see shortly — it will be worthwhile. This simple exercise is showing you how X-SPURT can be trained to become an expert on just about anything.

X-SPURT now goes through each of the outcomes you have entered, and says "If I asked the following question, in respect of this outcome, would you answer 'yes' or 'no'. From this information, X-SPURT can assemble an equivalent knowledge base to the one which was hardwired into SPURT.

```
PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  MAN  <
```

```
ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY

    > DOES IT FLY UNAIDED? N

    > DOES IT HAVE TWO LEGS? Y

    > DOES IT WALK? Y

PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  HORSE  <

ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY

    > DOES IT FLY UNAIDED? N

    > DOES IT HAVE TWO LEGS? N

    > DOES IT WALK? Y
```

```
PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  SPARROW  <

ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY

   > DOES IT FLY UNAIDED? Y

   > DOES IT HAVE TWO LEGS? Y

   > DOES IT WALK? Y
```

Once you've been through each of the possible outcomes, and told it what your answers would be for the questions, X-SPURT creates a 'knowledge base', which in this case is little more than adding up the total 'Y' replies. X-SPURT reports its findings to you:

```
THIS IS MY EXPERT BASE:

MAN ---- 6

HORSE ---- 4

SPARROW ---- 7
```

But where did it get those numbers? You could not have given four 'Y' answers for horse, or 7 for sparrow, because there are only three questions. X-SPURT does not add a single one for each 'Y' answer, but instead gives a number which changes for each answer. If there was just one awarded for each 'Y', and you answered 'Y' to, say, questions one and three for one thing, and to questions two and three for another thing, it would have the same total for both objects.

To get round this, to ensure that the actual **order** in which the 'Y' answers are given is important, we proceed as follows:

```
A 'Y' answer to question 1 is worth 1
A 'Y' answer to question 2 is worth 2
A 'Y' answer to question 3 is worth 4
A 'Y' answer to question 4 is worth 8
                    ..... 5     ... 16
                    ..... 6     ... 32
                    ..... 7     ... 64

              ...and so on...
```

This makes sure that, even if the same number of 'Y' answers are given for two different things, a different identifying number will be given to our expert by which to make judgements.

Does it work? Of course it does, and here is X-SPURT showing itself in action:

```
PLEASE ENTER 'Y' OR 'N'...

DOES IT FLY UNAIDED
? N

DOES IT HAVE TWO LEGS
? Y

DOES IT WALK
? Y
        > MY RESULT WAS 6

        > YOU WERE THINKING
          OF MAN
```

I said before that X-SPURT was capable of doing a great deal more than SPURT, and now I will show the truth of that claim. We are going to train our expert system in another field, one in which I have no expertise whatsoever. The knowledge base fed into X-SPURT came from a book, written by an expert called Oliver Chambers *(The Observer's Book of Rocks and Minerals,* New York: Frederick Warne, 1979). With the help of Mr. Chambers' expertise, X-SPURT is about to acquire the skills to identify five different types of minerals, using four factors.

This is something I could not do without X-SPURT's help. This is true for most people using expert systems today. An expert system encodes, as it were, an absent expert's expertise, so non-experts can make use of that knowledge base at will.

Let's pass some of Mr. Chambers' knowledge onto our system (starting with a new run, so that minerals do not become confused with horses and sparrows):

We tell it the subject matter:

```
NAME OF SYSTEM? MINERAL IDENTIFICATION

NUMBER OF OUTCOMES? 5

NUMBER OF FACTORS TO BE CONSIDERED? 4
```

Then we give it the five minerals it will be trying to identify:

```
WHAT IS OUTCOME 1 ? PLEONASTE

WHAT IS OUTCOME 2 ? LIMONITE

WHAT IS OUTCOME 3 ? IODYRITE

WHAT IS OUTCOME 4 ? IRIDOSMINE

WHAT IS OUTCOME 5 ? SYLVANITE
```

Next, X-SPURT learns some questions which will assist it in discriminating between the minerals:

```
PLEASE ENTER QUESTION 1
? IS IT HARD


PLEASE ENTER QUESTION 2
? DOES IT CONTAIN STREAKS OF A DIFFERENT
  COLOR FROM THE MAIN COLOR
```

```
PLEASE ENTER QUESTION 3
? IS ITS SPECIFIC GRAVITY ABOVE 5


PLEASE ENTER QUESTION 4
? IS IT FUSIBLE
```

Now it takes the user through the long process of encoding the expertise:

```
PLEASE ANSWER THE FOLLOWING QUESTION
FOR AN OUTCOME OF >  PLEONASTE  <


ENTER 'Y' FOR 'YES' ANSWER
   OR 'N' FOR A 'NO' REPLY


   > IS IT HARD? Y


   >
DOES IT CONTAIN STREAKS OF A DIFFERENT
COLOR FROM THE MAIN COLOR? N


   > IS ITS SPECIFIC GRAVITY ABOVE 5? N


   > IS IT FUSIBLE? N
```

It does this for the rest of the minerals, limonite, sylvanite and all. Finally, it reports its findings:

```
THIS IS MY EXPERT BASE:

PLEONASTE —— 1

LIMONITE —— 8

IODYRITE —— 12

IRIDOSMINE —— 7

SYLVANITE —— 13
```

Let's put it into action and see how well it does:

```
PLEASE THINK OF ONE OF THE FOLLOWING
  PLEONASTE
   LIMONITE
    IODYRITE
     IRIDOSMINE
      OR SYLVANITE


PLEASE ENTER 'Y' OR 'N'...


IS IT HARD
? N
```

```
DOES IT CONTAIN STREAKS OF A DIFFERENT
COLOR FROM THE MAIN COLOR
? N


IS ITS SPECIFIC GRAVITY ABOVE 5
? N


IS IT FUSIBLE
? Y
        > MY RESULT WAS 8


        > YOU WERE THINKING
          OF LIMONITE
```

In a matter of minutes, X-SPURT has acquired knowledge which allows me, as a total non-expert in that field, to make use of expert judgement in a practical situation.

We will now look at the construction of the program, to see how a 'soft-wired' SPURT has been created.

The program is controlled by a loop:

```
10 REMark ======
12 REMark XSPURT
14 REMark ======
16 :
20 initialise
30 gain_expertise
35 REPeat main_loop
40  demonstrate_expertise
50  space
```

```
60  PRINT "Press ENTER for another run,
or"
70  PRINT "any key and then ENTER to qui
t"
80  INPUT q$
90  IF q$<>"": EXIT main_loop
100 END REPeat main_loop
```

In the initialisation routine, X-SPURT acquires a name for the system (useful if you wish to save the entire expertise as a file) as well as the number of outcomes and factors. Arrays are dimensioned to hold the names and totals of the outcomes, as well as the questions relating to the factors.

```
940 DEFine PROCedure initialise
942  CLS #0
944  BORDER 1,4
946  PAPER 0
948  INK 4
950  CLS
955  PRINT
960  INPUT "Name of system?"!n$
970  space
980  INPUT "Number of outcomes?"!outcome
s
990  space
1000  INPUT "Number of factors to be con
sidered?"!factors
1010  DIM a$(outcomes,24),b$(factors,60)
1020  DIM d(outcomes)
1030  CLS
1040 END DEFine initialise
```

The next section of code the program visits gets the names of the outcomes:

```
450 REMark fill arrays
460  PRINT "     ";n$
470  space
480 REMark get outcome names
490  FOR j=1 TO outcomes
500   space
510   PRINT "What is outcome"!j!"? ";
520   INPUT a$(j)
530  NEXT j
```

And then X-SPURT asks for the factors, the questions to be asked:

```
550 REMark get questions to be asked
560  FOR j=1 TO factors
570   space
580   PRINT "Please enter question"!j
590   INPUT "?"!b$(j)
600  NEXT j
```

All this is just preparation. Now, X-SPURT wants to get some hard information, so it runs through the outcomes (using the J loop, from 630 through to 810) and within that the factors (with the K loop, 720 through to 800):

```
620 REMark acquire expertise
630  FOR j=1 TO outcomes
640   CLS
650   space
660   PRINT "Please answer the following
 question"
670   PRINT "for an outcome of >"!a$(j)!
"<"
680   space
690   PRINT "Enter 'y' for 'yes' answer"
700   PRINT "   or 'n' for a 'no' reply"
710   x=.5
```

```
720    FOR k=1 TO factors
730     x=x+x
740     space
750     PRINT "     >"!b$(k)!"?"!;
760     multi=0
770     INPUT y$
780     IF y$<>"n": multi=1
790     d(j)=d(j)+x*multi: REMark compile
 expert base
800    NEXT k
810   NEXT j
```

Having done this, X-SPURT shows you what it has managed to learn:

```
830   PRINT "This is my expert base:"
840   FOR j=1 TO outcomes
850    space
860    PRINT a$(j)!"---";d(j)
870   NEXT j
880   space
890   PRINT "          Press ENTER"
900   INPUT q$
910   CLS
920 END DEFine gain_expertise
```

With this information safely under its belt, X-SPURT is ready and able to perform in much the same way SPURT did, asking questions, adding up numbers, and from the total, making a decision:

```
120 DEFine PROCedure demonstrate_experti
se
130   CLS
140   space
150   PRINT "Please think of one of the f
ollowing"
```

```
160  FOR j=1 TO outcomes
170   PRINT"     "
180   IF j=outcomes: PRINT "or ";
190   PRINT a$(j)
200  NEXT j
210  space
220  result=0
230  x=.5
240  PRINT "Please enter 'y' or 'n'..."
250  FOR j=1 TO factors
260   x=x+x
270   space
280   PRINT b$(j)
290   INPUT e$
300   IF e$<>"n": result=result+x
310  NEXT j
320  PRINT "          > My result was"!res
ult
330  space
340  m=0
345  REPeat m_loop
350   m=m+1
360   IF d(m)=result: EXIT m_loop
370   IF m>=outcomes
380    PRINT "          > I cannot identif
y it"
390    RETurn
392   END IF
394  END REPeat m_loop
400  PRINT "          > You were thinking"
410  PRINT "            of"!a$(m)
430 END DEFine demonstrate_expertise
```

As you can see, X-SPURT allows itself a little bit of fallibility, with I CANNOT IDENTIFY IT if the total it has reached does not tally with any of your input (line 380).

You can see that it tells you its tally after each run, so you can keep track of what it is doing. If you wish to impress people with your expert system, you'll probably enhance the impression it creates if the 'works' are not so publicly displayed.

Here's the complete listing of X-SPURT:

```
 10 REMark ======
 12 REMark XSPURT
 14 REMark ======
 16 :
 20 initialise
 30 gain_expertise
 35 REPeat main_loop
 40  demonstrate_expertise
 50  space
 60  PRINT "Press ENTER for another run,
or"
 70  PRINT "any key and then ENTER to qui
t"
 80  INPUT q$
 90  IF q$<>"": EXIT main_loop
100 END REPeat main_loop
105 STOP
110 :
120 DEFine PROCedure demonstrate_experti
se
130  CLS
140  space
150  PRINT "Please think of one of the f
ollowing"
```

```
160  FOR j=1 TO outcomes
170   PRINT"     "
180   IF j=outcomes: PRINT "or ";
190   PRINT a$(j)
200  NEXT j
210  space
220  result=0
230  x=.5
240  PRINT "Please enter 'y' or 'n'..."
250  FOR j=1 TO factors
260   x=x+x
270   space
280   PRINT b$(j)
290   INPUT e$
300   IF e$<>"n": result=result+x
310  NEXT j
320  PRINT "         > My result was"!res
ult
330  space
340  m=0
345  REPeat m_loop
350   m=m+1
360   IF d(m)=result: EXIT m_loop
370   IF m>=outcomes
380    PRINT "         > I cannot identif
y it"
390    RETurn
392   END IF
394  END REPeat m_loop
400  PRINT "         > You were thinking"
410  PRINT "           of"!a$(m)
430 END DEFine demonstrate_expertise
435 :
440 DEFine PROCedure gain_expertise
```

```
450 REMark fill arrays
460  PRINT "     ";n$
470  space
480 REMark get outcome names
490  FOR j=1 TO outcomes
500   space
510   PRINT "What is outcome"!j!"? ";
520   INPUT a$(j)
530  NEXT j
540  CLS
550 REMark get questions to be asked
560  FOR j=1 TO factors
570   space
580   PRINT "Please enter question"!j
590   INPUT "?"!b$(j)
600  NEXT j
610  CLS
620 REMark acquire expertise
630  FOR j=1 TO outcomes
640   CLS
650   space
660   PRINT "Please answer the following
 question"
670   PRINT "for an outcome of >"!a$(j)!
"<"
680   space
690   PRINT "Enter 'y' for 'yes' answer"
700   PRINT "   or 'n' for a 'no' reply"
710   x=.5
720   FOR k=1 TO factors
730    x=x+x
740    space
750    PRINT "     >"!b$(k)!"?"!;
760    multi=0
770    INPUT y$
```

```
780     IF y$<>"n": multi=1
790     d(j)=d(j)+x*multi: REMark compile
 expert base
800    NEXT k
810   NEXT j
820   CLS
830   PRINT "This is my expert base:"
840   FOR j=1 TO outcomes
850    space
860    PRINT a$(j)!"---";d(j)
870   NEXT j
880   space
890   PRINT "         Press ENTER"
900   INPUT q$
910   CLS
920 END DEFine gain_expertise
930 :
940 DEFine PROCedure initialise
942   CLS #0
944   BORDER 1,4
946   PAPER 0
948   INK 4
950   CLS
955   PRINT
960   INPUT "Name of system?"!n$
970   space
980   INPUT "Number of outcomes?"!outcome
s
990   space
1000   INPUT "Number of factors to be con
sidered?"!factors
1010   DIM a$(outcomes,24),b$(factors,60)
1020   DIM d(outcomes)
1030   CLS
1040 END DEFine initialise
```

```
1045 :
1050 DEFine PROCedure space
1060  PRINT: PRINT
1070 END DEFine space
1080 :
1090 REMark ==========
1100 REMark END XSPURT
1110 REMark ==========
```

CHOOSING A CHIP

It would be very inconvenient if we had to educate our expert, as we did X-SPURT, every time we wanted to make use of the expertise. It is unlikely, in a real situation, a totally 'soft' expert would be needed. This next program shows an expert body of knowledge — the ability to distinguish between several chips (of the silicon variety) — encoded in DATA statements.

Here's the program in action:

```
THIS IS MY EXPERT BASE:

TMS 9940 (NMOS) —— 44

68000 (NMOS) —— 12

9940 (I3L) —— 56

MN1610 (NMOS) —— 46

8086 —— 60

Z8001 —— 28
```

```
I CAN IDENTIFY FROM THE FOLLOWING:
  TMS 9940 (NMOS)
   68000 (NMOS)
    9940 (I3L)
     MN1610 (NMOS)
      8086
       OR Z8001
```

After telling you what it can do, the program asks you to answer a number of questions in relation to the chip you are trying to identify. It will then tell you the name of the chip:

```
PLEASE ENTER 'Y' OR 'N'...


IS THE WORD SIZE 32 BITS
? N


DOES IT ADDRESS 64K
? N


IS THE CLOCK RATE 5MHz OR LESS
? Y


IS SHORTEST OBEY 3 MICROSECONDS OR LESS
? Y


DOES INSTRUCTION SET CONTAIN MORE THAN
71 INSTRUCTIONS
? N
```

```
IS THE PACKAGE 40 PIN DIP
? N
        > MY RESULT WAS 12


        > THE ONE YOU HAVE
          IS 68000 (NMOS)
```

The expert base came from another expert, Ken Ozanne, whose expertise was once again encoded in a book *(The Interface Computer Encyclopedia,* London: Interface Publications, 1983). Once the information is locked into the DATA statements in the program, it is ready for use at any time.

Here's the crucial part of the program, where the knowledge is stored:

```
580 DEFine PROCedure initialise
590  CLS
600  RESTORE
610  outcomes=6
620  factors=6
630  DIM a$(outcomes,15), b$(factors,60)
, d(outcomes)
640  FOR j=1 TO outcomes
650   READ a$(j), d(j)
660  END FOR j
670  FOR j=1 TO factors
680   READ b$(j)
690  END FOR j
700 END DEFine initialise
705 :
710 DEFine PROCedure double_space
720  PRINT: PRINT
```

```
730 END DEFine double_space
735 :
740 REMark ====
742 REMark data
744 REMark ====
746 :
750 REMark outcomes (chip titles)
760 DATA "TMS 9940 (NMOS)",44,"68000 (NM
OS)", 12
770 DATA "9940 (13L)",56,"MN1610 (NMOS)"
,46
780 DATA "8086",60,"Z8001",28
785 :
790 REMark questions
800 DATA "Is the word size 32 bits"
810 DATA "Does it address 64K"
820 DATA "Is the clock rate 5MHz or less
"
830 DATA "Is the shortest obey 3 microse
conds  or less"
840 DATA "Does instruction set contain m
ore    than 71 instructions"
850 DATA "Is the package 41 pin DIP"
```

Even if you have no desire whatsoever to identify chips, you can still make use of the expert system encoded in this program. As you can see, the variables OUTCOMES and FACTORS are assigned in lines 610 and 620, and these are used to dimension the arrays in line 630. Change the variables to the outcomes and factors you have, modify the DATA statements, and you have your very own expert system, ready to help you. The crucial number which the system uses to identify the chip (or to isolate whichever outcome you want) is held in the DATA statements immediately following the name of the chip.

To work out the numbers, I set up a chart like the following. It is simple to do the same for your subject:

OUTCOME	*FACTOR*						*TOTAL*
	1	*2*	*3*	*4*	*5*	*6*	
TMS 9940	*0*	*0*	*1*	*1*	*0*	*1*	*44*
68000	*0*	*0*	*1*	*1*	*0*	*0*	*12*
994013L	*0*	*0*	*0*	*1*	*1*	*1*	*56*
MN1610	*0*	*1*	*1*	*1*	*0*	*1*	*46*
8086	*0*	*0*	*1*	*1*	*1*	*1*	*60*
Z8001	*0*	*0*	*1*	*1*	*1*	*0*	*28*

Here's the complete program now, so you can create the expert of your choice.

```
10 REMark ==========
12 REMark CHIP-CHOICE
14 REMark ==========
16 :
20 initialise
30 show_base
35 REPeat main_loop
40  identify_chip
50  double_space
60  PRINT "Press ENTER for another chip,
 or"
70  PRINT "any key and then ENTER to qui
t"
```

```
80  INPUT q$
90  IF q$<>"": EXIT main_loop
95 END REPeat main_loop
100 STOP
105 :
110 REMark ==========
112 REMark definitions
114 REMark ==========
116 :
120 DEFine PROCedure identify_chip
130  CLS
140  double_space
150  PRINT "I can identify from the foll
owing:"
160  FOR j=1 TO outcomes
170   PRINT "     ";
180   IF j=outcomes: PRINT "OR ";
190   PRINT a$(j)
200  END FOR j
210  double_space
220  result=0
230  x=.5
240  PRINT 'Please enter "y" or "n"...'
250  FOR j=1 TO factors
260   x=x+x
270   double_space
280   PRINT b$(j)
290   INPUT e$
300   IF e$<>"n": result=result+x
310  END FOR j
320  PRINT "          > My result was"!res
ult
330  double space
340  m=0
345  REPeat m_loop
```

```
350    m=m+1
360    IF d(m)=result: EXIT m_loop
370    IF m>=outcomes
380     PRINT "          > I cannot identif
y it"
390     RETurn
392    END IF
394   END REPeat m_loop
400   PRINT "          > The one you have"
410   PRINT "            is"!a$(m)
430 END DEFine identify_chip
440 :
450 DEFine PROCedure show_base
460   CLS
470   PRINT: PRINT "This is my expert bas
e:"
480   FOR j=1 TO outcomes
490    PRINT
500    PRINT "    ";a$(j);" ---";d(j)
510   END FOR j
520   double_space
530   PRINT "          Press ENTER"
540   INPUT q$
550   CLS
560 END DEFine show_base
570 :
580 DEFine PROCedure initialise
590   CLS
600   RESTORE
610   outcomes=6
620   factors=6
630   DIM a$(outcomes,15), b$(factors,60)
, d(outcomes)
640   FOR j=1 TO outcomes
650    READ a$(j), d(j)
```

```
660  END FOR j
670  FOR j=1 TO factors
680   READ b$(j)
690  END FOR j
700 END DEFine initialise
705 :
710 DEFine PROCedure double_space
720  PRINT: PRINT
730 END DEFine double_space
735 :
740 REMark ====
742 REMark data
744 REMark ====
746 :
750 REMark outcomes (chip titles)
760 DATA "TMS 9940 (NMOS)",44,"68000 (NM
OS)", 12
770 DATA "9940 (13L)",56,"MN1610 (NMOS)"
,46
780 DATA "8086",60,"Z8001",28
785 :
790 REMark questions
800 DATA "Is the word size 32 bits"
810 DATA "Does it address 64K"
820 DATA "Is the clock rate 5MHz or less
"
830 DATA "Is the shortest obey 3 microse
conds  or less"
840 DATA "Does instruction set contain m
ore    than 71 instructions"
850 DATA "Is the package 41 pin DIP"
860 :
870 REMark ===============
880 REMark END CHIP-CHOICE
890 REMark ===============
```

CHAPTER THIRTEEN — SELF-LEARNING SYSTEMS

You'll recall, in the second system we looked at in this section, that the program X-SPURT allowed you to enter expertise on any subject. Once you'd fed it in, the program was ready to be your expert on the subject you had chosen.

However, it had one disadvantage. It demanded that you run through each of the factors, for each of the outcomes, in order to acquire a knowledge base from which it could work.

Our next program, SELFLEARN, does not require the same kind of spoonfeeding which was needed with X-SPURT. here it is in action:

```
HOW MANY FACTORS? 3


ENTER FACTOR 1
? WINGS


ENTER FACTOR 2
? PAIR OF EYES


ENTER FACTOR 3
? EATS WORMS
```

```
ENTER OUTCOME 1
? SPARROW


ENTER OUTCOME 2
? HUMAN
```

Once you have this information in place, you can run the program, and it will proceed to teach itself how to tell the difference between various outcomes:

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


NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? Y
```

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

For a while it will get things wrong, as you see above, but then will start getting some correct guesses:

```
NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? Y
                                  > 1
                               >BRAYN= 2
OUTCOME IS SPARROW
IS THIS CORRECT? ('Y' OR 'N')
? Y
```

```
NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? N
                                  > 0
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? N
                                  > 0
                               >BRAYN=-1
OUTCOME IS HUMAN
IS THIS CORRECT? ('Y' OR 'N')
? Y
```

In due course it will become infallible:

```
NOW I WILL DEMONSTRATE MY EXPERTISE...
THINK OF ONE OF THE OUTCOMES
IS WINGS TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS PAIR OF EYES TRUE? ('Y' OR 'N')
? Y
                                  > 1
IS EATS WORMS TRUE? ('Y' OR 'N')
? Y
                                  > 1
                               >BRAYN= 1
OUTCOME IS SPARROW
IS THIS CORRECT? ('Y' OR 'N')
? Y
```

HOW IT WORKS

The important thing (and the major limitation) of this program is that it can only distinguish between two outcomes (such as SPARROW and MAN in our example). The program starts with the assumption that its total (the variable BRAYN) will be either greater than or equal to zero, or less than zero. The actual value BRAYN achieves does not matter.

When you first run it, the program asks for the raw information it will need:

```
400 DEFine PROCedure initialise
410  CLS
420  outcomes=2
430  PRINT: PRINT
440  INPUT "How many factors?"!factors
450  DIM a$(outcomes,12)
460  DIM b$(factors,24)
470  DIM c(factors), d(factors)
480  CLS
490  FOR j=1 TO factors
500   PRINT: PRINT
510   PRINT "Enter factor"!j
520   INPUT "?"!b$(j)
530  END FOR j
540  PRINT: PRINT
550  CLS
560  FOR j=1 TO outcomes
570   PRINT: PRINT
580   PRINT "Enter outcome"!j
590   INPUT "?"!a$(j)
600  END FOR j
610 END DEFine initialise
```

Each time through the loop, SELFLEARN begins by filling each element of the C array (there is one element for each FACT) with zero:

```
30 REPeat learning_loop
40  CLS
50  FOR j=1 TO factors
60   c(j)=0
70  END FOR j
80  PRINT
90  demonstration_time
100 END REPeat learning_loop
```

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:

```
120 DEFine PROCedure demonstration_time
130  PRINT "Now I will demonstrate my ex
pertise.."
140  PRINT "Think of one of the outcomes
"
150  FOR j=1 TO factors
170   PRINT "Is"!b$(j)!"true? ('y' or 'n
')"
175   REPeat control
180    INPUT "?"!z$
190    IF z$="y" OR z$="n": EXIT control
195   END REPeat control
200   IF z$='y': c(j)=1
210   PRINT "
  >"!c(j)
220  END FOR j
```

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:

```
230  brayn=0
240  FOR j=1 TO factors
250   brayn=brayn+c(j)*d(j)
260  END FOR j
270  PRINT "                                    >bray
n=";brayn
```

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:

```
280  IF brayn>=0: PRINT "Outcome is";a$(
1): ex=-1
290  IF brayn<0: PRINT "Outcome is";a$(2
): ex=1
```

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:

```
300  PRINT "Is this correct? ('y' or 'n'
)"
305  REPeat control
310   INPUT "?"!z$
320   IF z$="y" OR z$="n": EXIT control
325  END REPeat control
330  PRINT
340  IF z$='y': RETurn
350  FOR j=1 TO factors
360   d(j)=d(j)+ex*c(j)
370  END FOR j
380 END DEFine demonstration_time
```

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 complete listing:

```
 10 REMark =========
 12 REMark SELFLEARN
 14 REMark =========
 16 :
 20 initialise
 30 REPeat learning_loop
 40  CLS
 50  FOR j=1 TO factors
 60   c(j)=0
 70  END FOR j
 80  PRINT
 90  demonstration_time
100 END REPeat learning_loop
105 :
110 REMark ==========
```

```
112 REMark definitions
114 REMark ===========
116 :
120 DEFine PROCedure demonstration_time
130  PRINT "Now I will demonstrate my ex
pertise.."
140  PRINT "Think of one of the outcomes
"
150  FOR j=1 TO factors
170   PRINT "Is"!b$(j)!"true? ('y' or 'n
')"
175   REPeat control
180    INPUT "?"!z$
190    IF z$="y" OR z$="n": EXIT control
195   END REPeat control
200   IF z$='y': c(j)=1
210   PRINT "
  >"!c(j)
220  END FOR j
230  brayn=0
240  FOR j=1 TO factors
250   brayn=brayn+c(j)*d(j)
260  END FOR j
270  PRINT "                              >bray
n="!brayn
280  IF brayn>=0: PRINT "Outcome is"!a$(
1): ex=-1
290  IF brayn<0: PRINT "Outcome is"!a$(2
): ex=1
300  PRINT "Is this correct? ('y' or 'n'
)"
305  REPeat control
310   INPUT "?"!z$
320   IF z$="y" OR z$="n": EXIT control
325  END REPeat control
```

```
330  PRINT
340  IF z$='y': RETurn
350  FOR j=1 TO factors
360   d(j)=d(j)+ex*c(j)
370  END FOR j
380 END DEFine demonstration_time
390 :
400 DEFine PROCedure initialise
410  CLS
420  outcomes=2
430  PRINT: PRINT
440  INPUT "How many factors?"!factors
450  DIM a$(outcomes,12)
460  DIM b$(factors,24)
470  DIM c(factors), d(factors)
480  CLS
490  FOR j=1 TO factors
500   PRINT: PRINT
510   PRINT "Enter factor"!j
520   INPUT "?"!b$(j)
530  END FOR j
540  PRINT: PRINT
550  CLS
560  FOR j=1 TO outcomes
570   PRINT: PRINT
580   PRINT "Enter outcome"!j
590   INPUT "?"!a$(j)
600  END FOR j
610 END DEFine initialise
620 :
630 REMark =============
640 REMark END SELFLEARN
650 REMark =============
```

MORE THAN TWO ALTERNATIVES

Although it is facinating to have an expert system which teaches itself in this way, it is a major drawback to be able to choose from just two alternatives. Our next program, MULTI-SELF-LEARN, is designed to allow for more than two outcomes.

The program starts much as you would, by now, expect:

```
HOW MANY OUTCOMES? 3

HOW MANY FACTORS? 3


PLEASE ENTER NAME OF OUTCOME 1
? HUMAN
PLEASE ENTER NAME OF OUTCOME 2
? HORSE
PLEASE ENTER NAME OF OUTCOME 3
? SPARROW



PLEASE ENTER NAME OF FACTOR 1
? A SINGLE PAIR OF LEGS
PLEASE ENTER NAME OF FACTOR 2
? CAN FLY UNAIDED
PLEASE ENTER NAME OF FACTOR 3
? BREATHES OXYGEN
```

```
THESE ARE THE POSSIBLE OUTCOMES:

HUMAN
 HORSE
  SPARROW

PLEASE THINK OF ONE OF THEM

PRESS RETURN WHEN YOU ARE READY
?
```

However, when you run it, you'll see that it asks you questions, then makes a guess as to what you were thinking of. If it was wrong, it asks you what the correct answer should have been:

```
PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF

A SINGLE PAIR OF LEGS
? Y
CAN FLY UNAIDED
? N
BREATHES OXYGEN
? Y
YOU WERE THINKING OF HORSE

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? N
WHAT SHOULD THE ANSWER HAVE BEEN
? HUMAN
```

Run it long enough (which is not very long with only three outcomes, and three factors) and it gets them right every time:

```
PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF

A SINGLE PAIR OF LEGS
? N
CAN FLY UNAIDED
? N
BREATHES OXYGEN
? Y
YOU WERE THINKING OF HORSE

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? Y

PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF

A SINGLE PAIR OF LEGS
? Y
CAN FLY UNAIDED
? Y
BREATHES OXYGEN
? Y
YOU WERE THINKING OF SPARROW

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? Y
```

The important part of this program lies between lines 150 and 520. The first section of this accepts the 'Y' answers, and increments a variable called COUNT in terms of your answer (adding 1 for the first 'Y', 2 for the next, 4 for the next, then 8, 16, 32 and so on):

```
150  PRINT "Please answer 'y' or 'n' for
 each of"
160  PRINT "the following in respect of"
170  PRINT "the outcome you have thought
 of"
180  PRINT
190  count=0
200  x=.5
210  FOR j=1 TO fact
220   x=x+x
230   PRINT f$(j)
235   REPeat control
240    INPUT "?"!z$
250    IF z$="y" OR z$="n"
252     EXIT control
254    END IF
256   END REPeat control
260   IF z$="y": count=count+x
270  END FOR j
```

After the program has been running for a while, it will have assigned values to many elements of the B array. B(1) will be the total when A$(1) is the outcome, B(6) will be the total for an outcome of A$(6) and so on. A small loop is traversed after the 'Y' answers are given, to see if the total obtained equals any previously-stored B(J) value. If it does, then the variable X is set equal to the relevant J:

```
280  x=0
290  FOR j=1 TO otco
```

```
300    IF count=b(j): x=j
310  END FOR j
```

If such a value has been assigned, X is no longer equal to zero, and the system has made a decision:

```
320  IF x=0
```

If no definite answer has been obtained here, the computer generates a random number between 1 and the number of outcomes, in order to make a guess. But it is not enough to then say "WERE YOU THINKING OF";A$ (random number generated). Although your program may not yet have assigned an A$(n) to the n total you received, it may well have assigned some elements of A$. Therefore it can, and must, reject some 'guesses' produced by the random number generator:

```
330     x=RND(1,otco)
340     REMark rejects answers known to b
e wrong
350     flag=0
355     j=0
360     REPeat j_loop
365       j=j+1
370       IF b(j)=0: EXIT j_loop
380       IF x=j AND count<>b(j)
382         flag=1
384       END IF
386       IF j>=otco: EXIT j_loop
390     END REPeat j_loop
400     IF flag<>1: EXIT x_loop
```

If the variable FLAG equals anything except zero, then the guess (the element of A$ represented by the randomly-generated value of X) cannot be used, as the system

already knows that answer is wrong. Therefore, using line 400, it goes back to choose a new value for X.

Having done so, the program checks on its guess:

```
410  PRINT "You were thinking of"!a$(x)
420  PRINT
430  PRINT "Enter 'y' if I'm right, 'n'
if wrong"
435  REPeat control
440   INPUT "?"!z$
450   IF z$="y" OR z$="n"
452    EXIT control
454   END IF
456  END REPeat control
460  IF z$="y"
462   b(x)=count
464   NEXT main_loop
466  END IF
470  PRINT "What should the answer have
been"
480   INPUT "?"!z$
490   FOR j=1 TO otco
500    IF a$(j)=z$: b(j)=count
510   END FOR j
520 END REPeat main_loop
```

If the guess was correct, the system uses the loop from 490 to 510 to find out which element of A$ corresponds to the total generated in that run. This ensures that, when it meets the same total next time, it will be able to identify the relevant element of A$.

Here's the complete listing, so you can set up a major expert system, which will teach itself:

```
10 REMark ================
12 REMark MULTI-SELF-LEARN
14 REMark ================
16 :
20 initialise
25 REPeat main_loop
30  CLS
40  PRINT "These are the possible outcom
es:"
50  PRINT
60  FOR j=1 TO otco
70   PRINT "    ";a$(j)
80  END FOR j
90  PRINT
100  PRINT "Please think of one of them"
110  PRINT
120  PRINT "Press ENTER when you are rea
dy"
125  PRINT "or any other key then ENTER
to quit"
130  INPUT "?"!z$
135  IF z$<>"": EXIT main_loop
140  CLS
150  PRINT "Please answer 'y' or 'n' for
 each of"
160  PRINT "the following in respect of"
170  PRINT "the outcome you have thought
 of"
180  PRINT
190  count=0
200  x=.5
210  FOR j=1 TO fact
220   x=x+x
230   PRINT f$(j)
235   REPeat control
240    INPUT "?"!z$
```

```
250     IF z$="y" OR z$="n"
252      EXIT control
254     END IF
256    END REPeat control
260    IF z$="y": count=count+x
270   END FOR j
280   x=0
290   FOR j=1 TO otco
300    IF count=b(j): x=j
310   END FOR j
320   IF x=0
325    REPeat x_loop
330     x=RND(1,otco)
340     REMark rejects answers known to b
e wrong
350     flag=0
355     j=0
360     REPeat j_loop
365      j=j+1
370      IF b(j)=0: EXIT j_loop
380      IF x=j AND count<>b(j)
382       flag=1
384      END IF
386      IF j>=otco: EXIT j_loop
390     END REPeat j_loop
400     IF flag<>1: EXIT x_loop
402    END REPeat x_loop
404   END IF
410   PRINT "You were thinking of"!a$(x)
420   PRINT
430   PRINT "Enter 'y' if I'm right, 'n'
if wrong"
435   REPeat control
440    INPUT "?"!z$
450    IF z$="y" OR z$="n"
```

```
452     EXIT control
454    END IF
456   END REPeat control
460   IF z$="y"
462    b(x)=count
464    NEXT main_loop
466   END IF
470   PRINT "What should the answer have
been"
480   INPUT "?"!z$
490   FOR j=1 TO otco
500    IF a$(j)=z$: b(j)=count
510   END FOR j
520 END REPeat main_loop
530 STOP
535 :
540 REMark ==========
542 REMark definition
544 REMark ==========
546 :
550 DEFine PROCedure initialise
552   CLS #0
554   BORDER 2,3
556   PAPER 7
558   INK 3
560   CLS
565   PRINT
570   INPUT "How many outcomes?"!otco
580   PRINT
590   INPUT "How many factors?"!fact
600   CLS
610   x=otco+fact
620   DIM a$(otco,12)
630   DIM f$(fact,30), b(x)
640   FOR j=1 TO otco
```

```
650    PRINT "Please enter name of outcom
e"!j
660    INPUT "?"!a$(j)
670   END FOR j
680   CLS
690   FOR j=1 TO fact
700    PRINT "Please enter name of factor
"!j
710    INPUT "?"!f$(j)
720   END FOR j
730 END DEFine initialise
740 :
750 REMark ====================
760 REMark END MULTI-SELF-LEARN
770 REMark ====================
```

NO NEED FOR CORRECTION

Our final program in this section is a variation of the one you have just been studying. The only difference between this one, and the preceding one, is that this one does not need to be told what the correct answer should have been if it makes a mistake. The program works out for itself, fairly quickly how to distinguish between the various values.

I have not renumbered this program, so it will be easy to modify MULTI-SELF-LEARN so that it becomes MULTI-SELF-LEARN-2. Here is the program up and running:

```
HOW MANY OUTCOMES? 5

HOW MANY FACTORS? 5
```

```
PLEASE ENTER NAME OF OUTCOME 1
? DOG
PLEASE ENTER NAME OF OUTCOME 2
? HORSE
PLEASE ENTER NAME OF OUTCOME 3
? SHEEP
PLEASE ENTER NAME OF OUTCOME 4
? CROW
PLEASE ENTER NAME OF OUTCOME 5
? HUMAN


PLEASE ENTER NAME OF FACTOR 1
? POWER OF SPEECH
PLEASE ENTER NAME OF FACTOR 2
? BARKS
PLEASE ENTER NAME OF FACTOR 3
? ABLE TO FLY
PLEASE ENTER NAME OF FACTOR 4
? FOUR LEGS
PLEASE ENTER NAME OF FACTOR 5
? PRODUCES WOOL
```

Once the knowledge base is in place, the program proceeds as follows:

```
THESE ARE THE POSSIBLE OUTCOMES:

DOG
 HORSE
  SHEEP
   CROW
    HUMAN
```

```
PLEASE THINK OF ONE OF THEM

PRESS RETURN WHEN YOU ARE READY
?

PLEASE ANSWER 'Y' OR 'N' FOR EACH OF
THE FOLLOWING IN RESPECT OF
THE OUTCOME YOU HAVE THOUGHT OF
```

While most of its answers will be wrong when the run begins, the correct guesses will become more frequent as time goes on:

```
POWER OF SPEECH
? N
BARKS
? Y
ABLE TO FLY
? N
FOUR LEGS
? Y
PRODUCES WOOL
? N
YOU WERE THINKING OF CROW

ENTER 'Y' IF I'M RIGHT, 'N' IF WRONG
? N
```

Eventually, it will not make any mistakes. It works in much the same way as the previous program, assigning values to the elements of the B array. However, this time it has an array called C, which ensures that it will never make the same wrong guess from the same total. This acts to increasingly limit the possible outcomes which could be obtained, so the number of things it can guess from is

reduced each time. It does not take it long to build up a 'world view' which ensures it will be right every time. The number of trials it will take to obtain perfection depends, of course, on the number of outcomes.

After the program has managed to teach itself to distinguish between the outcomes without error, it will be holding a knowledge base it refers to for each subsequent trial. This is the base that MULTI-SELF-LEARN-2 built up during that run:

```
J= 1  B(J)= 10   A$(J)=DOG
J= 2  B(J)= 8   A$(J)=HORSE
J= 3  B(J)= 24   A$(J)=SHEEP
J= 4  B(J)= 4   A$(J)=CROW
J= 5  B(J)= 1   A$(J)=HUMAN
```

And this is the listing:

```
 10 REMark ==================
 12 REMark MULTI-SELF-LEARN-2
 14 REMark ==================
 16 :
 20 initialise
 25 REPeat main_loop
 30  CLS
 40  PRINT "These are the possible outcom
es:"
 50  PRINT
 60  FOR j=1 TO otco
 70   PRINT "    ";a$(j)
 80  END FOR j
 90  PRINT
100  PRINT "Please think of one of them"
110  PRINT
120  PRINT "Press ENTER when you are rea
dy"
```

```
125  PRINT "or any other key then ENTER
to quit"
130  INPUT "?"!z$
135  IF z$<>"": EXIT main_loop
140  CLS
150  PRINT "Please answer 'y' or 'n' for
 each of"
160  PRINT "the following in respect of"
170  PRINT "the outcome you have thought
 of"
180  PRINT
190  count=0
200  x=.5
210  FOR j=1 TO fact
220   x=x+x
230   PRINT f$(j)
235   REPeat control
240    INPUT "?"!z$
250    IF z$="y" OR z$="n"
252     EXIT control
254    END IF
256   END REPeat control
260   IF z$="y": count=count+x
270  END FOR j
280  x=0
290  FOR j=1 TO otco
300   IF count=b(j): x=j
310  END FOR j
320  IF x=0
325   REPeat x_loop
330    x=RND(1,otco)
340    REMark rejects answers known to b
e wrong
350     flag=0
355     j=0
```

```
360     REPeat j_loop
365       j=j+1
370       IF b(j)=0: EXIT j_loop
380       IF x=j AND count<>b(j)
381         flag=1
382       END IF
383       IF j>=otco: EXIT j_loop
385       IF c(x)=count OR d(x)=count
386         flag=1
387       END IF
390     END REPeat j_loop
400     IF flag<>1: EXIT x_loop
402   END REPeat x_loop
404  END IF
410  PRINT "You were thinking of"!a$(x)
420  PRINT
430  PRINT "Enter 'y' if I'm right, 'n'
if wrong"
435  REPeat control
440    INPUT "?"!z$
450    IF z$="y" OR z$="n"
452      EXIT control
454    END IF
456  END REPeat control
460  IF z$="y"
462    b(x)=count
464    NEXT main_loop
466  END IF
470  IF c(x)=0
472    c(x)=count
474    NEXT main_loop
476  END IF
510  d(x)ount
520 END REPeat main_loop
530 STOP
535 :
```

```
540 REMark ==========
542 REMark definition
544 REMark ==========
546 :
550 DEFine PROCedure initialise
552  CLS #0
554  BORDER 1,6
556  PAPER 0
558  INK 6
560  CLS
565  PRINT
570  INPUT "How many outcomes?"!otco
580  PRINT
590  INPUT "How many factors?"!fact
600  CLS
610  x=otco+fact
620  DIM a$(otco,12)
630  DIM f$(fact,30),b(x),c(x),d(x)
640  FOR j=1 TO otco
650   PRINT "Please enter name of outcom
e"!j
660   INPUT "?"!a$(j)
670  END FOR j
680  CLS
690  FOR j=1 TO fact
700   PRINT "Please enter name of factor
"!j
710   INPUT "?"!f$(j)
720  END FOR j
730 END DEFine initialise
740 :
750 REMark ====================
760 REMark END MULTI-SELF-LEARN-2
770 REMark ====================
```

You can now demonstrate Artificial Intelligence in action on your microcomputer.

From programs which learn and reason, to those which will talk to you, obey you and advise you, this book will guide you into the fascinating world of AI – where science fact interacts with science fiction.

Can a machine really think?

What is the nature of intelligence, and will a machine ever be built to partake of that nature?

Answer these questions for yourself, as you explore AI with a host of ready-to-run programs including BLOCKWORLD (communicate in ordinary English with your computer, and watch it follow your orders), DOCTOR (the most powerful version of ELIZA ever published in BASIC), X-SPERT (an expert system ready to help you with any subject you can imagine), SYLLOGY (a program which reasons) and TIC (noughts and crosses game which learns as it plays, improving its skill from game to game).

This book is part of the 'Interface Artificial Intelligence Library'.

another great book from Interface

£6.95

ISBN 0-947695-18-4