Exploring ARTIFICIAL INTELLIGENCE on Your Apple® II

Tim Hartnell

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE® II

Bantam Computer Books
Ask your bookseller for the books you have missed

THE APPLE //c BOOK
by Bill O'Brien

THE COMMODORE 64 SURVIVAL MANUAL
by Winn L. Rosch

EXECUTIVE GUIDE TO THE EPSON GENEVA
by Marshall P. Smith

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE II
by Tim Hartnell

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR COMMODORE 64
by Tim Hartnell

EXPLORING THE UNIX ENVIRONMENT
by The Waite Group/Irene Pasternack

FRAMEWORK FROM THE GROUND UP
by The Waite Group/Cynthia Spoor and Robert Warren

HOW TO GET THE MOST OUT OF COMPUSERVE
by Charles Bowen and David Peyton

HOW TO GET THE MOST OUT OF THE SOURCE
by Charles Bowen and David Peyton

THE MACINTOSH
by Bill O'Brien

THE NEW *jr*: A GUIDE TO IBM'S PC*jr*
by Winn L. Rosch

ORCHESTRATING SYMPHONY
by The Waite Group/Dan Schafer with Mary Johnson

PC-DOS/MS-DOS
User's Guide to the Most Popular Operating System for Personal Computers
by Alan M. Boyd

POWER PAINTING: COMPUTER GRAPHICS ON THE MACINTOSH
by Verne Bauman and Ronald Kidd/illustrated by Gasper Vaccaro

SMARTER TELECOMMUNICATIONS
Hands-On Guide to On-Line Computer Services
by Charles Bowen and Stewart Schneider

SWING WITH JAZZ: LOTUS JAZZ FOR THE MACINTOSH
by Datatech Publications/Michael McCarty

TEACH YOUR BABY TO USE A COMPUTER
Birth Through Preschool
by Victoria Williams, Ph.D., and Frederick Williams, Ph.D.

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE® II

TIM HARTNELL

BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

This low-priced Bantam Book
has been completely reset in a type face
designed for easy reading, and was printed
from new plates. It contains the complete
text of the original hard-cover edition.
NOT ONE WORD HAS BEEN OMITTED.

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE II
A Bantam Book / November 1985

PRINTING HISTORY

First published in the UK by:
Interface Publications

First Published in Australia by:
Interface Publications

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

Interface Publications has exclusive right to the following program names: TRANSLATE, BLOCKWORLD, SPURT, X-SPURT, SYLLOGY, TICTAC, HANSHAN and SELFLEARN.

Cover design by J. Caroff Associates, Inc.

Apple II is a trademark of Apple Computer, Inc.

All rights reserved.
Copyright © 1985 by Tim Hartnell.

Book design by Nicola Mazzella.

This book may not be reproduced in whole or in part, by
mimeograph or any other means, without permission.
For information address: Bantam Books, Inc.

ISBN 0-553-34261-4

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

HL 0 9 8 7 6 5 4 3 2 1

CONTENTS

FOREWORD ix

CHAPTER ONE—LEARNING AND REASONING 1

Feedback 3
How Do Machines Think? 4
Switches and Decisions 5
"Real" Logic Gates 8

CHAPTER TWO—A PROGRAM THAT LEARNS 11

Samuel and the Checkerboard 13
TICTAC—The Program 15

CHAPTER THREE—A PROGRAM THAT REASONS 25

Back to the Program 30
SYLLOGY—The Program 35

CHAPTER FOUR—SEARCH TREES AND SNICKERS 39

Why Is It Called a Tree? 39
"Parallel Processing" 41
Vanishing Acts 42
Digging Deep 45
Mini-Maxing 47
Weighted Elements 51
The Alpha-Beta Algorithm 53
How the Program Works 64
SNICKERS—The Program 72

CHAPTER FIVE—THE WIDER VALUE OF GAMES 79

Real-World Complexities. 80
Other Games, Other Lessons 81
More on Samuel's Checkers 82
Go and Othello 82

CHAPTER SIX—UNDERSTANDING NATURAL LANGUAGE 84

SHRDLU 84
Language Parsing 85
BLOCKWORLD 86
Problems 88
Syntax and Semantics 89

CHAPTER SEVEN—BLOCKWORLD 92

The Domain 92
How Does It Work? 96
Modules of the Program 101
BLOCKWORLD—The Program 110
Improving the Program 116
SHRDLU Conversation 117

CHAPTER EIGHT—THE DOCTOR IS IN 120

Weizenbaum and ELIZA 120
The Russian Connection 122
Short, Sharp Shocks 123
DOCTOR 126
How It Works 129
The Reply Database 129
The Program Structure 133
Using the Myflag 136
DOCTOR—The Program 137

CHAPTER NINE—MACHINE TRANSLATION 147

Still a Use for People 147
SYSTRAN in Action 149

Franglais 151
Program Structure 152
TRANSLATE—The Program 156

CHAPTER TEN—HANSHAN 159

The Database 159
Sample Haiku 161
HANSHAN—The Program 162

CHAPTER ELEVEN—EXPERT SYSTEMS 165

IF/THEN Chains 165
Leibniz 167
Limitations 168
Chemical Structure and DENDRAL 169

CHAPTER TWELVE—THE LITTLE SPURT 172

MEDICI 172
Sample Run SPURT 173
The Program Structure 175
SPURT—The Program 176
The Little X-SPURT 177
Mineral Expertise 180
The Program Structure 183
X-SPURT—The Program 187
Choosing a Chip 190
CHIP-CHOICE—The Program 193

CHAPTER THIRTEEN—SELF-LEARNING SYSTEMS 196

Sample Run SELFLEARN 197
How It Works 199
SELFLEARN—The Program 201
More Than Two Alternatives 203
MULTI-SELF-LEARN Listing 207
No Need for Correction 209
MULTI-SELF-LEARN-2 Listing 212

APPENDIXES 214

APPENDIX ONE
Improving Your Programming Technique 214

APPENDIX TWO
Suggestions for Further Reading 223

APPENDIX THREE
Artificial Intelligence and Computer Terms 238

INDEX 247

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 that 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 that learn and reason, to those that 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 that—admittedly crude—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.

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE® II

CHAPTER ONE

LEARNING AND REASONING

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

The programs in this book certainly allow your computer to exhibit intelligent responses to situations, making decisions and acting on them. However, there is no suggestion that your computer has awareness of its actions. It does not laugh at the nonsequiturs produced in 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 that recognizes such things as the "effectiveness" of a poem, or the incongruity of a response, we cannot really suggest that intelligence is present.

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

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

TICTAC, a program that plays Tic-Tac-Toe (or Naughts 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 naughts or three crosses in a row) and within ten games or so TICTAC will have learned 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 that—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 that 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 intelli-

gent?" 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 judgments 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 that 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 that appears vital as a prerequisite for claiming that intelligence actually exists in a system, some aspects of intelligence—reasoning and the ability to learn—were within our present capabilities.

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

FEEDBACK

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

An early "machine that would learn" was the turtle, a forerunner of a swarm of such robotic terrapins, built in 1948 by Grey Walter, a physiologist who specialized 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 utilizing negative feedback. That is it would tend not to repeat behavior that was not productive. A turtle that 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 that 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 many kinds of mathematical problems. (What he apparently did not realize 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—that 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 around 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

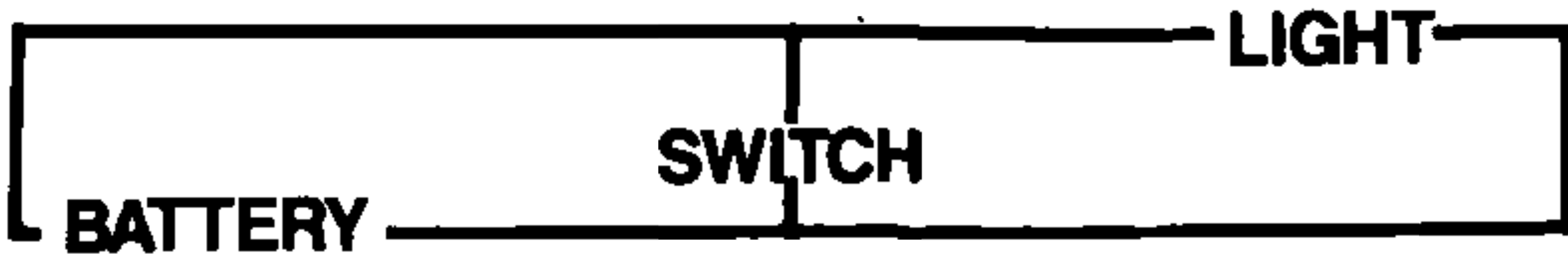
the current does not flow, its state will be said to be "0." On equals 1, off equals 0. Further, we will adopt the convention that when the light is lit, its state is 1; when the light is off, its state is 0.

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

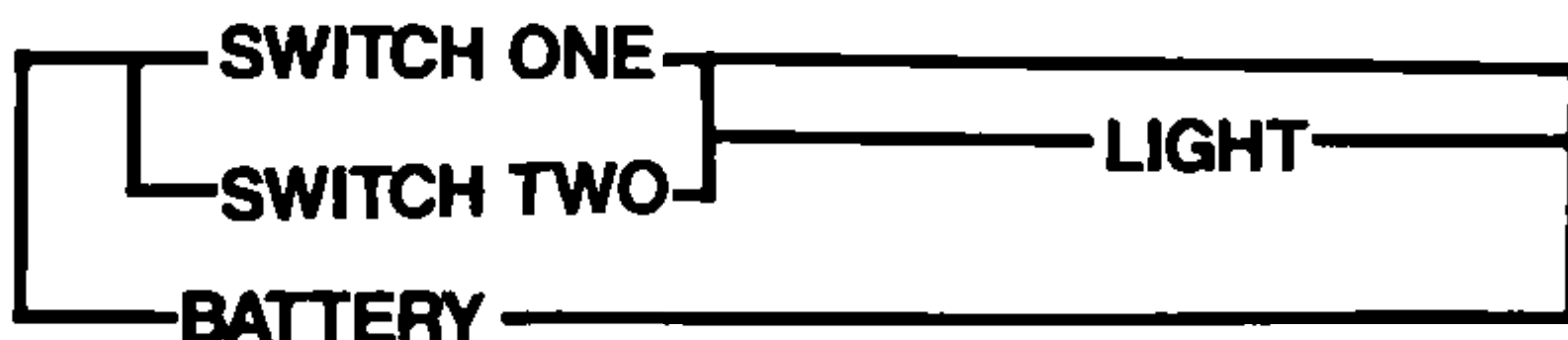
Now we get to the interesting bits, where circuits can "make decisions." Imagine we have a circuit with two switches in it, as follows:

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.

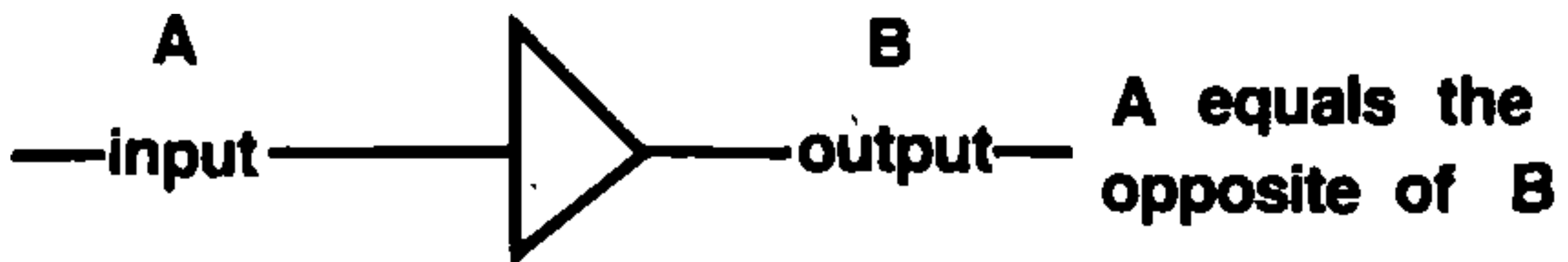
SWITCH ONE	SWITCH TWO	LIGHT
0	0	0
1	0	1
0	1	1
1	1	1

"REAL" LOGIC GATES

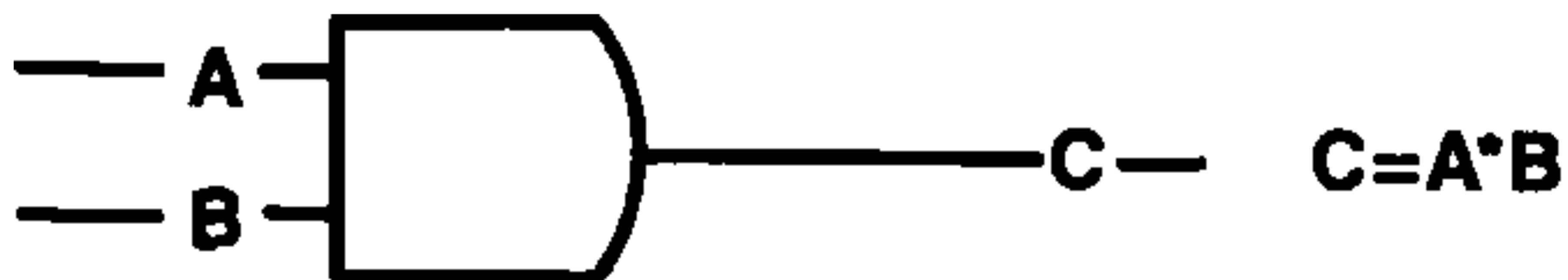
Your computer uses logic gates just like these, except of course they are not great big switches that need to be turned off and on. One reason Shannon and Stibitz used relays is because these are switches that can be turned on without actually touching them (when an electrical current is applied, a magnetic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



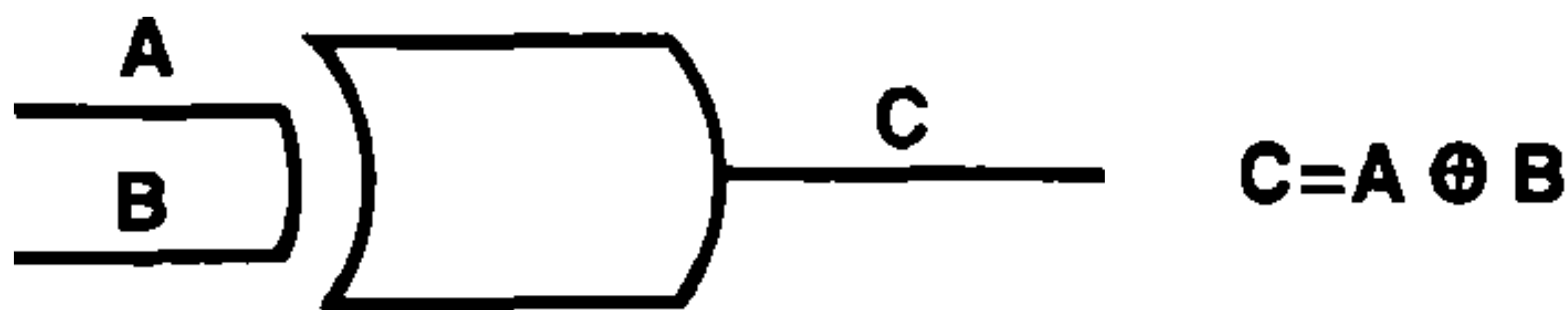
This is an AND gate:



And this the OR gate:

There's another gate that is often used in circuits, and that will be helpful to you when trying to work out how circuits reach decisions. This is the 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
1	0	0
0	1	0
1	1	1

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 it—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 THAT 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 Naughts 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 that is going to learn as it runs needs its working rules in a "soft" form that 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 accordance with a sequence that 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 a draw. 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 realize 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 having 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 naughts 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 naughts 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 that face a player if he or she starts in a game of naughts 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 that were written on the vacant squares. Each number was duplicated several times, with the same number of each number per box. That is, if squares three and four were vacant in one board position, the matchbox contained, say, five scraps of paper with the number three written on them, and five bearing the number four.

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

At the end of the game, Michie returned to his list of moves and boxes. If the "matchbox computer" had won the game, an additional piece of paper bearing each number played was placed in the

relevant matchbox. That is, if the first matchbox used, the one bearing the blank grid, had yielded the number five, an additional piece of paper with the number five on it was placed in that matchbox. Naturally enough, this increased the chance that five would be selected next time the box was opened.

The process was continued for every box used in that game. If the game was a draw, the contents of the boxes were left unchanged. If the "computer" lost the game, the pieces of paper that 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 that 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 "drafts") 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 toward 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 that 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 that could be brought under attack by a single move.

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

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

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

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

TICTAC—THE PROGRAM

The program begins with an initialization sub-routine as follows:

```
1180 REM INITIALIZATION
1190 HOME
1200 DIM A(9):REM BOARD
1210 DIM M(10):REM TO HOLD KNOWLEDGE BASE
1220 DIM W(24):REM WIN/BLOCK DATA
1230 DIM D(5):REM TO HOLD MOVES IN CURRENT
GAME
1240 REM WIN/BLOCK DATA
1250 FOR J=1 TO 24
1260 READ W(J)
1270 NEXT J
1280 DATA 1,2,3,4,5,6,7,8,9
1290 DATA 1,4,7,2,5,8,3,6,9
1300 DATA 1,5,9,3,5,7
1310 REM INITIAL KNOWLEDGE BASE
1320 FOR J=1 TO 10
1330 READ M(J)
1340 NEXT J
1350 DATA 2,6,8,4,7,3,1,9,5,2
1360 RETURN
```

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 recognize 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 insures 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 preprogrammed 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 REM UPDATE KNOWLEDGE BASE
310 FOR B=1 TO 5
320 FOR J=2 TO 9
330 IF M(J)=D(B) THEN GOSUB 370
340 NEXT J
350 NEXT B
360 GOTO 430
370 REM ** RE-ORDER ELEMENTS OF 'M' ARRAY
**
380 TEMP=M(J+FLAG)
390 M(J+FLAG)=M(J)
400 M(J)=TEMP
410 J=9
420 RETURN
430 PRINT:PRINT
440 PRINT "THIS IS MY UPDATED
PRIORITY"
450 PRINT:PRINT
460 FOR J=1 TO 9
470 PRINT M(J);" ";
480 NEXT J
```

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 that it is trying to build (and that it is trying to prevent its opponent from completing). This is the section of code that looks for a move here, before using the knowledge base:

```
540 REM MACHINE MOVE
550 P=ASC("O")
560 X=0
570 J=1
580 IF A(W(J))=A(W(J+1)) AND A(W(J+2))=32
AND A(W(J))=P THEN X=W(J+2):GOTO 750
590 IF A(W(J))=A(W(J+2)) AND A(W(J+1))=32
AND A(W(J))=P THEN X=W(J+1):GOTO 750
600 IF A(W(J+1))=A(W(J+2)) AND A(W(J))=32
AND A(W(J+1))=P THEN X=W(J+1):GOTO 750
610 IF J<21 THEN J=J+3:GOTO 580
620 IF P=ASC("O") THEN P=ASC("X"):GOTO
570
```

It looks first for a winning move for itself (when P equals the ASCII code of the letter "O") and then tries for a blocking move (with P set equal to the code of the opponent's piece, the "X"). If it fails to find a move here, it brings in the data from the knowledge base:

```
630 REM ** IF NO WIN/BLOCK MOVE FOUND **
640 REM ** THEN THIS NEXT SECTION USED **
650 J=1
660 IF A(M(J))=32 THEN X=M(J):GOTO 750
670 IF J<10 THEN J=J+1:GOTO 660
```

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

```
680 H=0
690 H=H+1
700 X=INT(RND(1)*9):IF A(X)=32 THEN 750
710 IF H<100 THEN 690
720 R$="D":REM IT IS A DRAW
```

Having found a move, it makes it, then acts to insure 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) is not assigned, the game must be a draw.

```
750 REM MAKE MOVE
760 A(X)=ASC("O")
770 COUNT=COUNT+1
780 D(COUNT)=X
790 FLAG=0
800 FOR J=1 TO 9
810 IF A(J)=32 THEN FLAG=1
820 NEXT J
830 IF FLAG=0 AND R$="" THEN R$="D"
840 REM IF ALL POSITIONS FULL, AND R$ NOT
ASSIGNED, IT IS A DRAW
850 RETURN
```

After each move, human or machine, the WIN CHECK routine is visited:

```
870 REM WIN CHECK
880 J=1
890 IF A(W(J))=32 THEN J=J+3
900 IF J>23 THEN RETURN
910 IF A(W(J))=A(W(J+1)) AND A(W(J))=A(W
(J+2)) THEN 940
920 IF J<22 THEN J=J+3:GOTO 890
930 RETURN
940 IF A(W(J))=ASC("O") THEN R$="W":REM
MACHINE WINS
950 IF A(W(J))=ASC("X") THEN R$="L":REM
MACHINE LOSES
960 RETURN
```

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

```
10 REM TICTAC
20 GOSUB 1180:REM INITIALIZE
30 REM *** PREGAME SETTINGS ***
40 FOR J=1 TO 9
50 A(J)=32
60 NEXT J
70 FOR J=1 TO 5
80 D(J)=0
90 NEXT J
100 COUNT=0
110 R$=""
120 GOSUB 1070:REM PRINT BOARD
130 REM *** MAIN CYCLE ***
```

```
140 GOSUB 540:REM MACHINE MOVE
150 GOSUB 1070:REM PRINT BOARD
160 GOSUB 870:REM WIN CHECK
170 IF R$<>"" THEN 240
180 GOSUB 980:REM ACCEPT HUMAN MOVE
190 GOSUB 1070:REM PRINT BOARD
200 GOSUB 870:REM WIN CHECK
210 IF R$="" THEN 140
220 REM *** END MAIN CYCLE ***
230 REM ***********
240 REM END OF GAME
250 GOSUB 1070:REM PRINT BOARD
260 PRINT:PRINT
270 IF R$="W" THEN PRINT TAB(8)"I WIN":
FLAG=-1
280 IF R$="L" THEN PRINT TAB(8)"YOU WIN":
FLAG=1
290 IF R$="D" THEN PRINT TAB(6)"IT'S A
DRAW":GOTO 430
300 REM UPDATE KNOWLEDGE BASE
310 FOR B=1 TO 5
320 FOR J=2 TO 9
330 IF M(J)=D(B) THEN GOSUB 370
340 NEXT J
350 NEXT B
360 GOTO 430
370 REM ** RE-ORDER ELEMENTS OF 'M' ARRAY
**
380 TEMP=M(J+FLAG)
390 M(J+FLAG)=M(J)
400 M(J)=TEMP
410 J=9
420 RETURN
430 PRINT:PRINT
440 PRINT"THIS IS MY UPDATED
PRIORITY"
450 PRINT:PRINT
460 FOR J=1 TO 9
470 PRINT M(J);" ";
480 NEXT J
490 PRINT:PRINT
500 PRINT"PRESS RETURN TO CONTINUE"
510 INPUT A$
520 GOTO 30
530 REM ************
540 REM MACHINE MOVE
550 P=ASC("O")
```

```
560 X=0
570 J=1
580 IF A(W(J))=A(W(J+1)) AND A(W(J+2))=32
AND A(W(J))=P THEN X=W(J+2):GOTO 750
590 IF A(W(J))=A(W(J+2)) AND A(W(J+1))=32
AND A(W(J))=P THEN X=W(J+1):GOTO 750
600 IF A(W(J+1))=A(W(J+2)) AND A(W(J))=32
AND A(W(J+1))=P THEN X=W(J):GOTO 750
610 IF J<21 THEN J=J+3:GOTO 580
620 IF P=ASC("O") THEN P=ASC("X"):GOTO
570
630 REM ** IF NO WIN/BLOCK MOVE FOUND **
640 REM ** THEN THIS NEXT SECTION USED **
650 J=1
660 IF A(M(J))=32 THEN X=M(J):GOTO 750
670 IF J<10 THEN J=J+1:GOTO 660
680 H=0
690 H=H+1
700 X=INT(RND(1)*9):IF A(X)=32 THEN 750
710 IF H<100 THEN 690
720 R$="D":REM IT IS A DRAW
730 RETURN
740 REM *********
750 REM MAKE MOVE
760 A(X)=ASC("O")
770 COUNT=COUNT+1
780 D(COUNT)=X
790 FLAG=0
800 FOR J=1 TO 9
810 IF A(J)=32 THEN FLAG=1
820 NEXT J
830 IF FLAG=0 AND R$="" THEN R$="D"
840 REM IF ALL POSITIONS FULL, AND R$ NOT
ASSIGNED, IT IS A DRAW
850 RETURN
860 REM *********
870 REM WIN CHECK
880 J=1
890 IF A(W(J))=32 THEN J=J+3
900 IF J>23 THEN RETURN
910 IF A(W(J))=A(W(J+1)) AND A(W(J))=A(W
(J+2)) THEN 940
920 IF J<22 THEN J=J+3:GOTO 890
930 RETURN
940 IF A(W(J))=ASC("O") THEN R$="W":REM
MACHINE WINS
950 IF A(W(J))=ASC("X") THEN R$="L":REM
```

```
MACHINE LOSES
960 RETURN
970 REM **********
980 REM HUMAN MOVE
990 PRINT:PRINT
1000 PRINT"ENTER YOUR MOVE"
1010 INPUT MOVE
1020 IF MOVE<1 OR MOVE>9 THEN 1010
1030 IF A(MOVE)<>32 THEN 1010
1040 A(MOVE)=ASC("X")
1050 RETURN
1060 REM ***********
1070 REM PRINT BOARD
1080 HOME
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 RETURN
1170 REM **************
1180 REM INITIALIZATION
1190 HOME
1200 DIM A(9):REM BOARD
1210 DIM M(10):REM TO HOLD KNOWLEDGE BASE
1220 DIM W(24):REM WIN/BLOCK DATA
1230 DIM D(5):REM TO HOLD MOVES IN
CURRENT GAME
1240 REM WIN/BLOCK DATA
1250 FOR J=1 TO 24
1260 READ W(J)
1270 NEXT J
1280 DATA 1,2,3,4,5,6,7,8,9
1290 DATA 1,4,7,2,5,8,3,6,9
1300 DATA 1,5,9,3,5,7
1310 REM INITIAL KNOWLEDGE BASE
1320 FOR J=1 TO 10
1330 READ M(J)
```

```
1340 NEXT J
1350 DATA 2,6,8,4,7,3,1,9,5,2
1360 RETURN
```

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

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

To trigger this unintelligent, tireless opponent, simply replace line 180 with GOSUB 4500.

```
X : O : X
---------
O : O : O
---------
  : X :
```

CHAPTER THREE

A PROGRAM THAT REASONS

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

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

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

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

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

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

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

```
? AN EAGLE IS A BIRD
     > OK

? A BIRD IS A WINGED CREATURE
     > OK

? IS AN EAGLE A WINGED CREATURE
        [LOOKING FOR EAGLE]
         [ FOUND AT 1 1 ]
     > YES

?
```

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

```
? A BIRD IS A FLYER
     > OK

? IS AN EAGLE A FLYER
        [LOOKING FOR EAGLE]
         [ FOUND AT 1 1 ]
     > YES

? IS A FLYER A WINGED CREATURE
        [LOOKING FOR FLYER]
         [ FOUND AT 1 4 ]
     > YES
```

Please note that the preceding is not necessarily a *valid* syllogism. This is because SYLLOGY will accept, to add to its database, any statement of the following form: `A . . . is a . . .`

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

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

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

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

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

Entering a question mark when the prompt appears will allow you to discover what SYLLOGY is holding in its memory, under each category heading it has created. After you enter the question mark, 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 with a two-dimensional string array, Z$, cross-referencing the propositions entered into it, and from this cross reference producing conclusions.

This is fairly easy to understand if you visualize 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 that 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 25

× 25 array such as we have with this program, and you may well wish to save your databases on some subjects.

The TIM IS AN IDIOT series was, of course, handled quite separately from THE EAGLE IS A BIRD series. To make it easy to understand how SYLLOGY files and then accesses, and the propositions upon which it reaches conclusions, let's look at 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 that 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, storing BIRD underneath that at 2,4.

When you enter a question mark to check the contents of a file, the computer simply goes across to 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 it 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 FLYER 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 REM SYLLOGY
20 GOSUB 1050:REM INITIALIZE
30 PRINT:INPUT A$
40 IF A$="?" THEN 910
50 IF A$="" THEN END
60 FLAG=0
70 REM NOTE THERE IS A SPACE BEFORE THE
CLOSE QUOTE IN NEXT 4 LINES
80 IF LEFT$(A$,3)="IS " THEN 480:REM
CONCLUSIONS
90 IF LEFT$(A$,4)="THE " THEN A$=MID$(A$
,5)
100 IF LEFT$(A$,3)="AN " THEN A$=MID$(A$,
4)
```

```
110 IF LEFT$(A$,2)="A " THEN A$=MID$(A$,3)
120 X=LEN(A$)
130 N=0
140 N=N+1
150 IF MID$(A$,N,1)=" " THEN B$=LEFT$(A$,
N-1):GOTO 180:REM EXTRACTS FIRST NOUN
160 IF N<X THEN 140
170 PRINT"I DON'T UNDERSTAND":GOTO
30
180 K=4
190 IF MID$(A$,N+1,1)="W" THEN K=5
200 C$=MID$(A$,N+K):REM QUALIFYING PHRASE
210 IF LEFT$(C$,2)="A " THEN C$=MID$(C$,
3):REM REMOVES ARTICLE
220 IF LEFT$(C$,3)="AN " THEN C$=MID$(C$
,4)
230 IF LEFT$(C$,4)="THE " THEN C$=MID$(C$
,5)
```

Line 80 detects the "IS" at the start of the input, indicating that the user is asking SYLLOGY to try to reach a conclusion. This sends action to 480, where the conclusion routine begins.

Lines 90, 100 and 110 strip THE, AN or A from the front of the input, so that A$ now begins with the noun that 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 that 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 REM ** STORE INFORMATION **
250 REM ** FIRST CHECK TO SEE IF CAN FIND
SUBJECT BEFORE FINDING BLANK **
260 N=0
270 N=N+1
280 IF Z$(1,N)=B$ THEN 320:REM SUBJECT
HEADING EXISTS
290 IF Z$(1,N)="" THEN Z$(1,N)=B$:GOTO 320
300 IFN<25 THEN 270
310 PRINT "I HAVE NO MORE SUBJECT STORAGE
ROOM"
```

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 REM ** PROGRAM REACHES HERE WITH
SUBJECT STORED AS HEADING **
330 REM ** NOW PUT OBJECT UNDER THIS **
340 K=0
350 K=K+1
360 IF Z$(K,N)=C$ THEN 400:REM INFORMATION
ALREADY STORED UNDER THAT HEADING
370 IF Z$(K,N)="" THEN Z$(K,N)=C$:GOTO
400
380 IF K<25 THEN 350
390 PRINT"I HAVE NO MORE OBJECT STORAGE
SPACE"
400 IF FLAG=1 THEN PRINT TAB(6)">OK":
GOTO 30:REM SWAP HAS BEEN DONE
410 REM ** NOW SWAP OBJECT AND SUBJECT
AND SAVE AGAIN **
420 FLAG=1
430 M$=B$
440 B$=C$
450 C$=M$
460 GOTO 250
```

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 insures 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 that reaches conclusions:

```
480 REM ** CONCLUSIONS **
490 REM ** FIRST SPLIT INPUT **
500 A$=MID$(A$,4):REM STRIP "IS"
510 IF LEFT$(A$,2)="A " THEN A$=MID$(A$,3)
:REM STRIP "A" IF PRESENT
520 IF LEFT$(A$,3)="AN " THEN A$=MID$(A$,
4):REM STRIP "AN" IF PRESENT
530 REM ** GET FIRST WORD - F$ **
540 X=LEN(A$)
550 N=0
560 N=N+1
570 IF MID$(A$,N,1)=" " THEN F$=LEFT$(A$
,N-1):GOTO 600
580 IF N<X THEN 560
590 PRINT TAB(6)"I DO NOT UNDERSTAND":GOTO
30
```

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 as 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 REM ** NOW GET SECOND WORD - S$ **
610 S$=MID$(A$,N+3)
620 IF LEFT$(S$,1)=" " THEN S$=MID$(S$,2)
:REM STRIPS LEADING ZERO IF ARTICLE AN
630 PRINT TAB(9) "(LOOKING FOR"
;F$;")"
640 X=0
650 X=X+1
660 IF Z$(1,X)=F$ THEN PRINT TAB(10)
"( FOUND AT 1";X;")":GOTO 700
670 IF X<25 THEN 650
```

```
680 PRINT TAB(6)"> I.CANNOT FIND THE
SUBJECT":PRINT TAB(8)F$
690 GOTO 30
```

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
710 Y=Y+1
720 IF Z$(Y,X)=S$ THEN PRINT TAB(6)
"> YES":GOTO 30
730 IF Y<25 THEN 710
740 Y=1
750 Y=Y+1
760 P$=Z$(Y,X)
770 M=0
780 M=M+1
790 IF Z$(1,M)=P$ THEN 830
800 IF M<25 THEN 780
810 IF Y<25 THEN 750
820 PRINT TAB(6)"> NO":GOTO
30
830 Q=1
840 Q=Q+1
850 IF Z$(Q,M)=S$ THEN PRINT TAB(6)
"> YES":GOTO 30
860 IF Q<25 THEN 840
870 IF M<25 THEN 780
880 GOTO 820
```

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 REM CHECK CONTENTS OF PARTICULAR FILE
920 INPUT "SUBJECT TO CHECK?";H$
930 T=0
940 T=T+1
950 IF Z$(1,T)=H$ THEN 990
960 IF T<25 THEN 940
970 PRINT"I HAVE NO DATA STORED ON
";H$
980 GOTO 30
990 K=1
1000 K=K+1
1010 IF Z$(K,T)<>"" THEN PRINT K;T;Z$(K
,T)
1020 IF K<25 THEN 1000
1030 GOTO 30
```

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

SYLLOGY—THE PROGRAM

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

```
10 REM SYLLOGY
20 GOSUB 1050:REM INITIALIZE
30 PRINT:INPUT A$
40 IF A$="?" THEN 910
50 IF A$=" " THEN END
60 FLAG=0
70 REM NOTE THERE IS A SPACE BEFORE THE
CLOSE QUOTE IN NEXT 4 LINES
80 IF LEFT$(A$,3)="IS " THEN 480:REM
CONCLUSIONS
90 IF LEFT$(A$,4)="THE " THEN A$=MID$(A$
,5)
100 IF LEFT$(A$,3)="AN " THEN A$=MID$(A$
,4)
110 IF LEFT$(A$,2)="A " THEN A$=MID$(A$
,3)
120 X=LEN(A$)
130 N=0
```

```
140 N=N+1
150 IF MID$(A$,N,1)=" " THEN B$=LEFT$(A$,
N-1):GOTO 180:REM EXTRACTS FIRST NOUN
160 IF N<X THEN 140
170 PRINT"I DON'T UNDERSTAND":GOTO
30
180 K=4
190 IF MID$(A$,N+1,1)="W" THEN K=5
200 C$=MID$(A$,N+K):REM QUALIFYING PHRASE
210 IF LEFT$(C$,2)="A " THEN C$=MID$(C$,
3):REM REMOVES ARTICLE
220 IF LEFT$(C$,3)="AN " THEN C$=MID$(C$
,4)
230 IF LEFT$(C$,4)="THE " THEN C$=MID$(C$
,5)
240 REM ** STORE INFORMATION **
250 REM ** FIRST CHECK TO SEE IF CAN FIND
SUBJECT BEFORE FINDING BLANK **
260 N=0
270 N=N+1
280 IF Z$(1,N)=B$ THEN 320:REM SUBJECT
HEADING EXISTS
290 IF Z$(1,N)=" " THEN Z$(1,N)=B$:GOTO
320
300 IF N<25 THEN 270
310 PRINT"I HAVE NO MORE SUBJECT STORAGE
ROOM"
320 REM ** PROGRAM REACHES HERE WITH
SUBJECT STORED AS HEADING **
330 REM ** NOW PUT OBJECT UNDER THIS **
340 K=0
350 K=K+1
360 IF Z$(K,N)=C$ THEN 400:REM INFORMATION
ALREADY STORED UNDER THAT HEADING
370 IF Z$(K,N)=" " THEN Z$(K,N)=C$:GOTO
400
380 IF K<25 THEN 350
390 PRINT"I HAVE NO MORE OBJECT STORAGE
SPACE"
400 IF FLAG=1 THEN PRINT TAB(6)"> OK":
GOTO 30:REM SWAP HAS BEEN DONE
410 REM ** NOW SWAP OBJECT AND SUBJECT
AND SAVE AGAIN **
420 FLAG=1
430 M$=B$
440 B$=C$
450 C$=M$
460 GOTO 250
```

```
470 REM *****************
480 REM ** CONCLUSIONS **
490 REM ** FIRST SPLIT INPUT **
500 A$=MID$(A$,4):REM STRIP "IS"
510 IF LEFT$(A$,2)="A " THEN A$=MID$(A$,
3):REM STRIP "A" IF PRESENT
520 IF LEFT$(A$,3)="AN " THEN A$=MID$(A$,
4):REM STRIP "AN" IF PRESENT
530 REM ** GET FIRST WORD - F$ **
540 X=LEN(A$)
550 N=0
560 N=N+1
570 IF MID$(A$,N,1)=" " THEN F$=LEFT$(A$,
N-1):GOTO 600
580 IF N<X THEN 560
590 PRINT TAB(6)"I DO NOT UNDERSTAND":
GOTO 30
600 REM ** NOW GET SECOND WORD - S$ **
610 S$=MID$(A$,N+3)
620 IF LEFT$(S$,1)=" " THEN S$=MID$(S$,2)
:REM STRIPS LEADING ZERO IF ARTICLE "AN"
630 PRINT TAB(9)"(LOOKING FOR ";F$;")"
640 X=0
650 X=X+1
660 IF Z$(1,X)=F$ THEN PRINT TAB(10)
"(FOUND AT 1";X;")":GOTO 700
670 IF X<25 THEN 650
680 PRINT TAB(6)"> I CANNOT FIND THE
SUBJECT":PRINT TAB(8)F$
690 GOTO 30
700 Y=1
710 Y=Y+1
720 IF Z$(Y,X)=S$ THEN PRINT TAB(6)">
YES":GOTO 30
730 IF Y<25 THEN 710
740 Y=1
750 Y=Y+1
760 P$=Z$(Y,X)
770 M=0
780 M=M+1
790 IF Z$(1,M)=P$ THEN 830
800 IF M<25 THEN 780
810 IF Y<25 THEN 750
820 PRINT TAB(6)"> NO":GOTO 30
830 Q=1
840 Q=Q+1
```

```
850 IF Z$(Q,M)=S$ THEN PRINT TAB(6)"> YES"
:GOTO 30
860 IF Q<25 THEN 840
870 IF M<25 THEN 780
880 GOTO 820
890 REM **************
900 REM ******************************
910 REM CHECK CONTENTS OF PARTICULAR FILE
920 INPUT "SUBJECT TO CHECK?";H$
930 T=0
940 T=T+1
950 IF Z$(1,T)=H$ THEN 990
960 IF T<25 THEN 940
970 PRINT"I HAVE NO DATA STORED ON ";H$
980 GOTO 30
990 K=1
1000 K=K+1
1010 IF Z$(K,T))<>" " THEN PRINT K;" ";T;
" ";Z$(K,T)
1020 IF K<25 THEN 1000
1030 GOTO 30
1040 REM **********
1050 REM INITIALIZE
1060 HOME
1080 DIM Z$(25,25)
1090 RETURN
```

CHAPTER FOUR

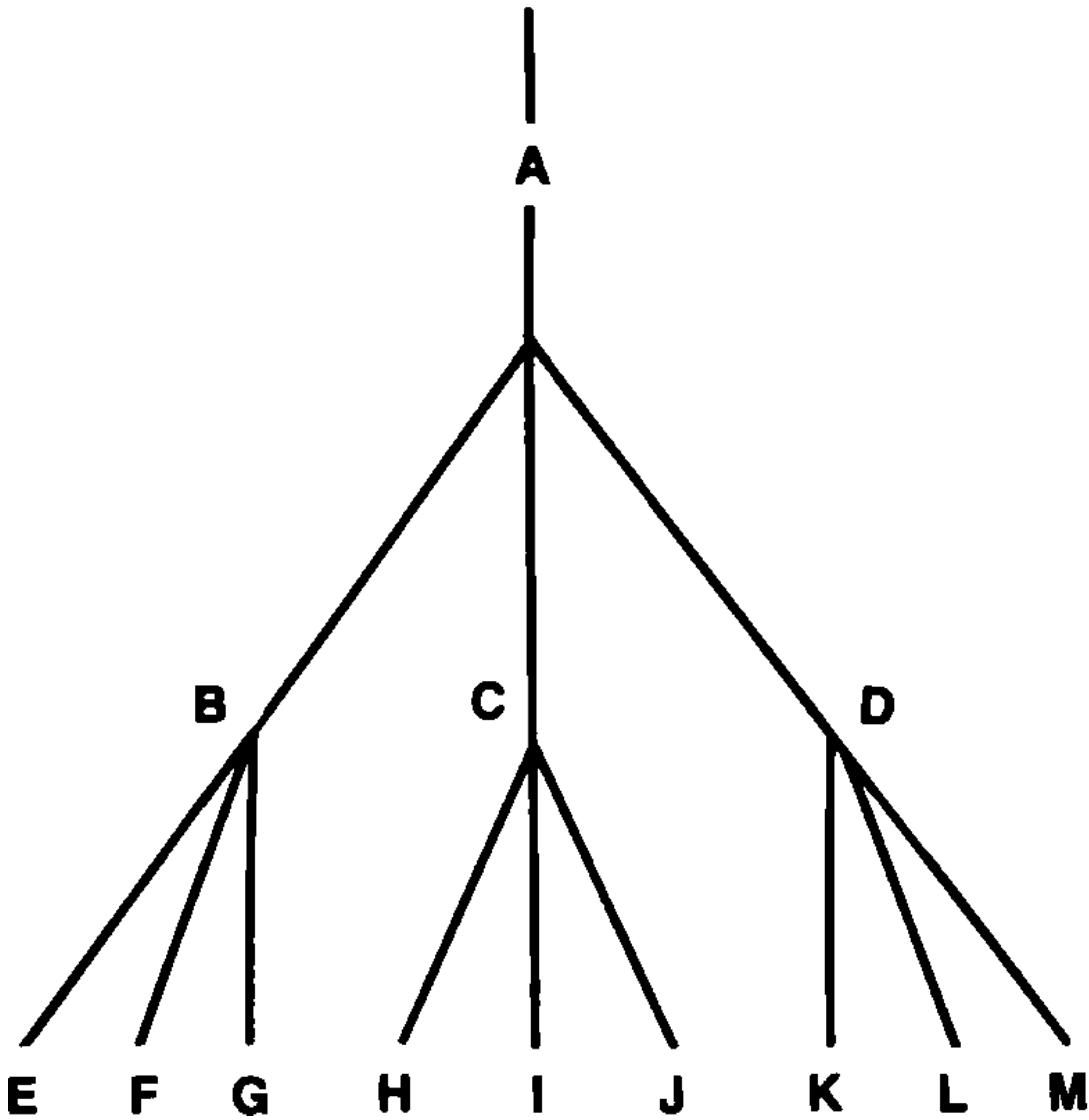
SEARCH TREES AND SNICKERS

In this section of the book, we will develop a program like checkers 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 following 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?

Apart from being upside down, a search tree grows like any other tree. Take A in the following diagram as the starting point for the search. The "branches" (labeled 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 were some means of guiding the search. Only in a very simple program, such as one that played Naughts and Crosses, could a program examine every branch of every tree, before choosing the best move.

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

"PARALLEL PROCESSING"

Another approach would be to examine a short distance down one branch, then back up, and the start of another branch, and so on, and then to examine the more promising branches to a greater depth. A branch, for example, that assumed the opponent in a chess game would sacrifice the queen to capture a pawn would not merit further examination. Any branch that led the opponent—in the opinion of the program's evaluation mechanism—to weaken his position could be abandoned the moment this discovery was made, and the processing time and effort put into it would follow to 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 a game like checkers called SNICKERS, for this section of the book, in order to demonstrate some aspects of primitive tree-searching.

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
```

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.

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 checkerboard. The pieces move as in checkers, that is diagonally from black square to black square. Each piece, then, is actually "sitting on" a dot that 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 backward and forward 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 or pieces moving "backward" 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 that holds the board inside the computer.

This is the numbered board the computer uses in SNICKERS:

	1	2	3	4	5	6	7	8
8	81	82	83	84	85	86	87	88
7	71	72	73	74	75	76	77	78
6	61	62	63	64	65	66	67	68
5	51	52	53	54	55	56	57	58
4	41	42	43	44	45	46	47	48
3	31	32	33	34	35	36	37	38
2	21	22	23	24	25	26	27	28
1	11	12	13	14	15	16	17	18

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 noncapture 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 that 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—perhaps 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 to follow it through on the board that was printed earlier, or on a checkerboard 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 either by moving a piece of your own into X+9, or, if this is not possible, by 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 that 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 that 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 TICTAC. That is, we could examine every possible move, of every possible game, and analyze 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 that 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, swap to another branch, then another and another, with the option of abandoning branches that 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 TICTAC section of

the book) or can be based on a hierarchical scheme to order moves chosen, and decide not to follow the majority of move branches that could be generated. As you'll see shortly, this is how we do it in the SNICKERS program.

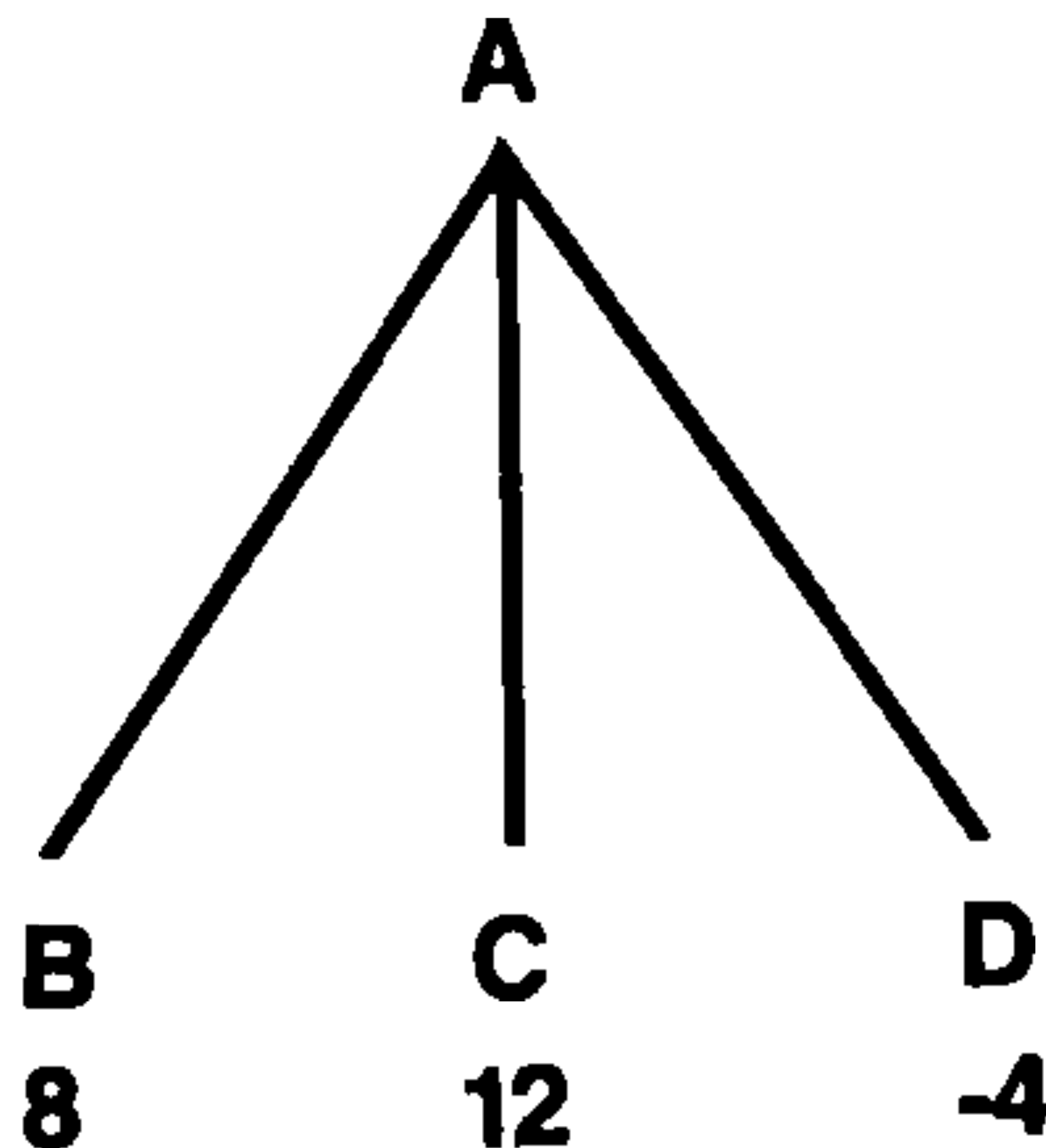
MINI-MAXING

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.

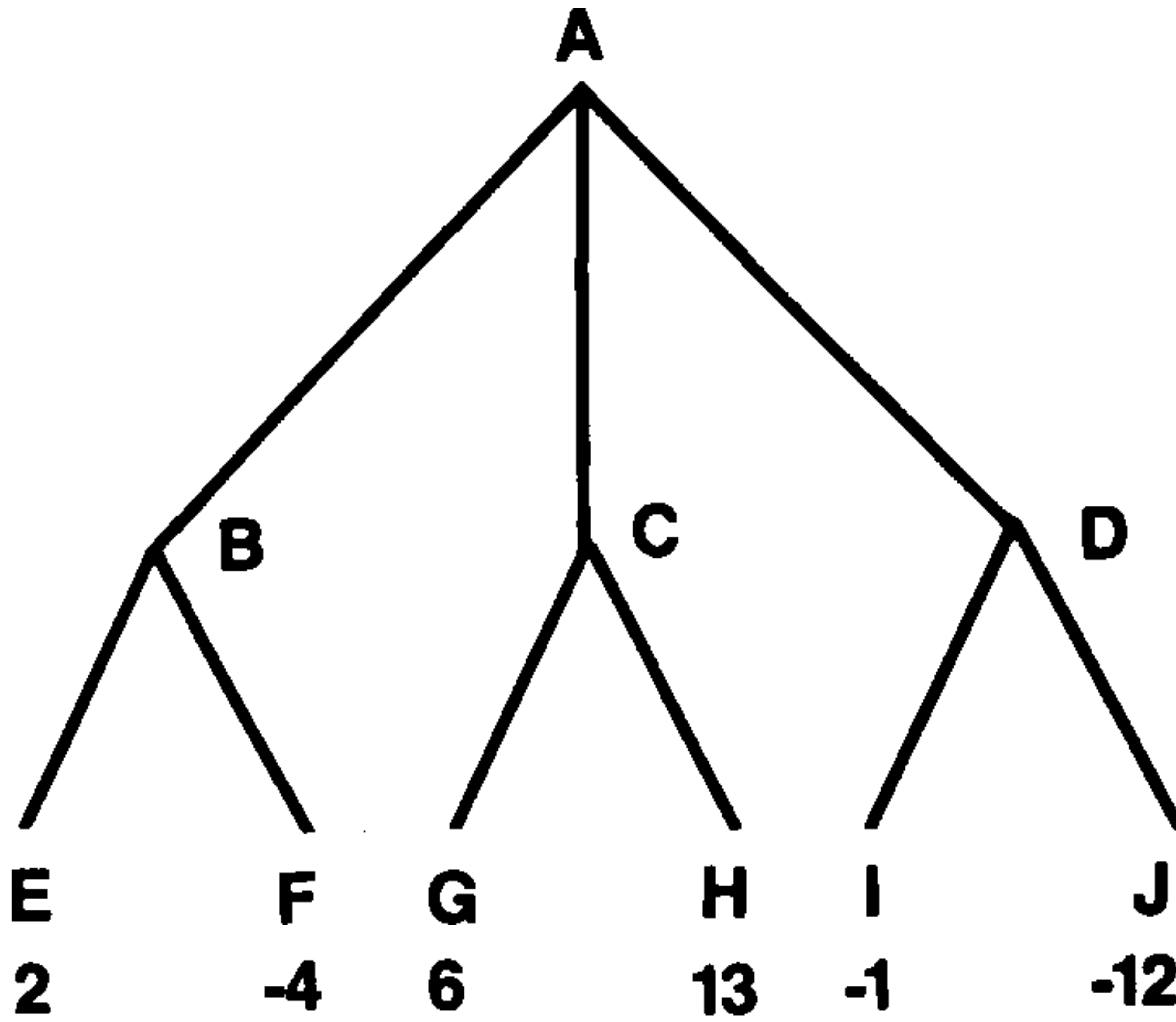
To use this, however, the computer should be able to assign numerical values to the positions it discovers.

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 that gives the human choices that will leave him or her in the weakest possible position. The choice then must be the one that gives the computer the maximum possible score while leaving the human choices that minimize 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 that 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 he or she 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 that 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 minimize 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 that will not. Naturally enough, it will make the move that leaves it in the strongest position after the move (with the piece that has done the capturing still on the board) and ignore the move that 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 that represent moves it is most unlikely to make, is as follows. Any moves found that fit the description are stored:

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

—Captures that 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.

—Noncapture moves onto the back row.

—Noncapture moves that 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 in practice (aided, of course, by the simple nature of the game), and manages to play with an appearance of skill.

So you can appreciate, I hope, that this hierarchical ordering of moves minimizes the number of possibilities that 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 that could be moved onto the back row, adding to its score. It has a predetermined order for doing this, insuring 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 that insures 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 discovered (that is, moves that do not expose the piece moved to capture) 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 that carry out each of the tasks specified.

Incidentally, you may feel the multiple sweeps of the board are somewhat wasteful. Could the program not do all of its looking in a single sweep? The answer, of course, is "yes," except it would mean a considerable waste of effort in many cases, when it would be looking for and storing moves that 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, and 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 naughts and crosses, is that the program can set up to play repeatedly against an opponent playing randomly, or against 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 that 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, one willing to consider sacrificing pieces, or to trade them, when your ''strength'' was positive, and that 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, and has led to unnecessary errors of judgment. 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 that 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 that 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 naughts 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), then there are twelve positions at the 2-ply level, and a number approaching 12×7 at the next level. ("Approaching" is used 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.)

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

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 that gives the best mini-max so far.

Doing this would mean you would end up with a single move that, 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 moves, in its own eyes, 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 insure that its score after the move is as good as possible, and that 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 that proceeded, given all the alternative moves that 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 that 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.

```
COMPUTER: 0    HUMAN: 0

       12345678
       --------
     8  . . . C 8
     7 . . C .  7
     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 that 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 that 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 toward 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 are a number of subroutine calls.

```
10 REM SNICKERS
20 GOSUB 2070:REM INITIALIZE
30 GOSUB 1760:REM PRINT BOARD
40 REM ** MAIN CYCLE STARTS **
50 GOSUB 190:REM COMPUTER MOVES
60 GOSUB 1760:REM PRINT BOARD
70 IF CO>4 THEN 120
80 GOSUB 1950:REM ACCEPT HUMAN MOVE
90 GOSUB 1760:REM PRINT BOARD
100 IF HS<5 THEN 50
110 REM ***********
```

As you can see from looking at the major loop for SNICKERS, this makes it very simple to understand the program's construction. It makes it easy to track down errors, as well. If, for example, the program was not printing the board correctly, it would make sense to look first in the subroutine beginning at line 1760, the routine called PRINT BOARD.

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

```
2070 REM INITIALIZE
2080 REM **********
2090 HOME
2110 DIM A(150):REM BOARD AND BLANK
SPACES AROUND AND BEYOND
2120 DIM G(3):REM GOOD, SAFE CAPTURE STORE
2130 DIM S(3):REM SAFE CAPTURE STORE
2140 DIM T(18):REM OTHER CAPTURE STORE,
ALSO USED FOR SAFE NON-CAPTURE MOVES
2150 E=ASC(" "):REM EMPTY 'WHITE' SQUARE
2160 B=ASC("."):REM EMPTY 'BLACK' SQUARE
2170 C=ASC("C"):REM COMPUTER PIECE
2180 H=ASC("H"):REM HUMAN PIECE
2190 HS=0:REM HUMAN SCORE
2200 CO=0:REM COMPUTER SCORE
2210 REM ** SET UP STARTING BOARD **
2220 FOR J=10 TO 80 STEP 10
2230 FOR K=1 TO 8
2240 READ X:A(J+K)=X
2250 NEXT K
2260 NEXT J
2270 RETURN
2280 REM ***********************
2290 DATA 72,32,72,32,72,32,72,32
2300 DATA 32,72,32,72,32,72,32,72
2310 DATA 46,32,46,32,46,32,46,32
2320 DATA 32,46,32,46,32,46,32,46
2330 DATA 46,32,46,32,46,32,46,32
2340 DATA 32,46,32,46,32,46,32,46
2350 DATA 67,32,67,32,67,32,67,32
2360 DATA 32,67,32,67,32,67,32,67
```

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 that are not classed by the program as either "good, safe" or "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 that will remind you of what the variable stands for, as we have in this case. HS holds the human score, and CO 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 at 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 REM **************
190 REM COMPUTER MOVES
200 REM **************
210 REM SEARCH FOR CAPTURES
220 GSAFE=0:REM TO COUNT GOOD, SAFE
CAPTURES WHICH THREATEN HUMAN PIECES
230 CSAFE=0:REM TO COUNT SAFE CAPTURES
WHICH DO NOT PLACE COMP. UNDER THREAT
240 CCAPTURE=0:REM TO COUNT OTHER CAPTURES
FOUND
```

The "stores" are emptied:

```
250 FOR J=1 TO 3
260 G(J)=0:REM EMPTY GOOD, SAFE CAPTURE
STORE
270 S(J)=0:REM EMPTY SAFE CAPTURE STORE
```

```
280 T(J)=0:REM EMPTY OTHER CAPTURE STORE
290 NEXT J
```

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 THEN 390:REM SKIP
EVALUATION IF NO COMPUTER PIECE THERE
330 REM ** CAPTURE TO RIGHT **
340 X=J+K-9:Y=J+K-18:Z=J+K-27:M=-11
350 IF A(X)=H AND A(Y)=B THEN GOSUB 700:
REM CAPTURE FOUND
360 REM ** CAPTURE TO LEFT **
370 X=J+K-11:Y=J+K-22:Z=J+K-33:M=-9
380 IF A(X)=H AND A(Y)=B THEN GOSUB 700:
REM CAPTURE FOUND
390 NEXT K
400 NEXT J
410 IF GSAFE+CSAFE+CCAPTURE=0 THEN 980:
REM NO CAPTURES FOUND
420 REM ** NOW CHOOSE CAPTURE TO MAKE **
430 PRINT:PRINT TAB(8)">> CAPTURE
FOUND"
440 FOR T=1 TO 2000:NEXT T
450 IF GSAFE<>0 THEN 500
460 IF CSAFE<>0 THEN 670
470 REM ** CHOOSE FROM GENERAL CAPTURES **
480 MOVE=T(INT(RND(1)*CCAPTURE)+1)
490 GOTO 540
500 REM ** CHOOSE FROM GOOD SAFE **
510 REM ** SELECT FROM STORED MOVES **
520 MOVE=G(INT(RND(1)*GSAFE)+1)
530 REM ** MAKE MOVE **
540 START=INT (MOVE/100)
550 ED=MOVE-100*START
560 A(START)=B
570 A(START-ED)=B
580 A(START-2*ED)=C
590 CO=CO+1
600 REM ** CHECK FOR ADDITIONAL SCORE IF
LANDING ON BACK ROW **
```

```
610 IF START-2*ED>18 THEN RETURN
620 A(START-2*ED)=B
630 CO=CO+1
640 PRINT "I CAPTURED AND LANDED ON"
;START-2*ED;"ON BACK ROW"
650 FOR T=1 TO 2000:NEXT T
660 RETURN
670 REM ** SAFE CAPTURE **
680 MOVE=S(INT(RND(1)*CSAFE)+1)
690 GOTO 540
700 REM ** CHECK PROPOSED CAPTURE FOR
SAFETY **
710 REM ** CHECK SQUARE BELOW IN SAME
DIRECTION AS INTENDED MOVE **
720 PRINT J+K;"TO";Y;"CAPTURING ON"
;X
730 FOR T=1 TO 1200:NEXT T
740 IF A(Z)=H THEN 920:REM STORE AS A
NON-SAFE CAPTURE
750 REM ** CHECK SQUARE IN OTHER DIRECTION
FROM INTENDED MOVE **
760 IF A(Y+M)=H AND A(Y-M)=B THEN 920
770 REM ** NOW CHECK TO SEE IF WILL LEAVE
A PIECE EXPOSED BY MAKING MOVE **
780 IF A(J+K+M)=C AND A(J+K+2*M)=H THEN
920
790 REM ** IF REACHED THIS POINT THEN
CAPTURE IS SAFE **
800 REM ** STORE THIS MOVE **
810 CSAFE=CSAFE+1
820 S(CSAFE)=100*(J+K)+20+M:REM THIS
ENCODES ENOUGH INFO TO RECREATE MOVE
830 REM ** NOW SEE IF THIS DESERVES TO BE
CALLED A 'GOOD, SAFE' CAPTURE
840 CHECK=GSAFE
850 IF Y+2*M<1 THEN RETURN
860 IF A(Y+M)=H AND A(Y-(20+M))<>B AND A
(Y+2*M)=B THEN GSAFE=GSAFE+1
870 IF CHECK=GSAFE THEN RETURN: REM **
THIS MOVE FOUND NOT TO BE 'GOOD SAFE'
880 REM ** STORE GOOD SAFE MOVE **
890 PRINT"I AM CONSIDERING";J+K;"TO";M+20
+J+K
900 G(GSAFE)=100*(J+K)+20+M
910 RETURN
920 REM ** STORE NON-SAFE CAPTURE **
930 CCAPTURE=CCAPTURE+1
```

```
940 PRINT"I AM CONSIDERING";J+K;"TO";
M+20+J+K
950 T(CCAPTURE)=100*(J+K)+20+M
960 RETURN
```

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. ("To" is called ED in the program; END would have caused a crash on many systems, if it was a reserved BASIC word.)

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

```
1510 START=INT(MOVE/100)
1520 ED=MOVE-100*START
1530 A(START)=B
1540 A(ED)=C
1550 RETURN
```

If the program has found one "good, safe" move (or more) 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 that is under threat from the human player.

```
980 REM ** MOVE TO PROTECT PIECE UNDER
THREAT **
990 MOVE=0
1000 J=80
1010 K=1
1020 Q=J+K
1030 IF A(Q)<>C THEN 1110:REM DO NOT
CONSIDER THIS SQUARE, NO COMP. PIECE
1040 IF A(Q+9)=B AND A(Q-9)=H AND A(Q+18)
=C THEN MOVE=100*(Q+18)+Q+9
1050 REM ** RANDOM NUMBER REJECTION OF
THIS MOVE IF IT EXPOSES A SUBSEQUENT ONE
1060 IF MOVE<>0 AND A(Q-2)=H AND A(Q+20)
=B AND RND(1)>.5 THEN 1510
```

```
1070 IF A(Q+9)=B AND A(Q-9)=H AND A(Q+20)
=C THEN MOVE=100*(Q+20)+Q+9:GOTO 1510
1080 IF A(Q+11)=B AND A(Q-11)=H AND A(Q+
22)=C THEN MOVE=100*(Q+22)+Q+11
1090 IF MOVE<>0 AND A(Q+2)=H AND A(Q+22)=B
AND RND(1)>.5 THEN 1510
1100 IF A(Q+11)=B AND A(Q-11)=H AND A(Q+
20)=C THEN MOVE=100*(Q+20)+Q+11:GOTO 1510
1110 IF K<8 THEN K=K+1:GOTO 1020
1120 IF J>10 THEN J=J-10:GOTO 1010
```

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 insures that the computer does not always blindly move to protect a piece (a blindness that 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 REM ** NO CAPTURE FOUND, SO LOOK FOR
MOVE TO 'DISAPPEAR' ON BACK ROW **
1150 MOVE=0
1160 REM UNDESIRABLE MOVES CHECKED FIRST,
SO CAN BE OVERWRITTEN BY BETTER ONES
1170 IF A(22)=C AND A(11)=B THEN MOVE=22
1180 IF A(28)=C AND A(17)=B THEN MOVE=28
1190 IF A(22)=C AND A(13)=B THEN MOVE=22
1200 IF A(26)=C AND A(17)=B THEN MOVE=26
1210 IF A(26)=C AND A(15)=B THEN MOVE=26
1220 IF A(24)=C AND A(15)=B THEN MOVE=24
1230 IF A(24)=C AND A(13)=B THEN MOVE=24
1240 IF MOVE=0 THEN 1310
1250 PRINT:PRINT"I'M MOVING ONTO
BACK ROW FROM";MOVE
1260 FOR T=1 TO 2000:NEXT T
1270 A(MOVE)=B
1280 CS=CS+1
1290 RETURN
```

If this is not possible, the program sweeps to find a legal move that 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:

```
1300 REM ***************************
1310 REM ** SAFE, NON-CAPTURE MOVES **
1320 CMOVE=0:REM COUNT MOVES FOUND
1330 FOR J=80 TO 30 STEP-10
1340 FOR K=1 TO 8
1350 IF A(J+K)<>C THEN 1460
1360 X=J+K-9:Y=J+K-18:Z=J+K-20
1370 Q=J+K+2
1380 IF A(X)<>B THEN 1460
1390 IF A(Y)=H OR A(Z)=H AND A(Q)=B THEN
1460
1400 GOSUB 1560
1410 X=J+K-11:Y=J+K-22:Z=J+K-20
1420 Q=J+K-2
1430 IF A(X)<>B THEN 1460
1440 IF A(Y)=H OR A(Z)=H AND A(Q)=B THEN
1460
1450 GOSUB 1560
1460 NEXT K
1470 NEXT J
1480 IF CMOVE=0 THEN 1630
1490 REM ** MAKE MOVE **
1500 MOVE=T(INT(RND(1)*CMOVE)+1)
```

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 REM RANDOM NON-CAPTURE MOVE
1640 PRINT"I AM LOOKING FOR RANDOM, LEGAL
MOVE"
1650 L=0
1660 L=L+1
1670 J=10*INT(RND(1)*8+1)
1680 K=INT(RND(1)*8+1)
1690 IF A(J+K)=C THEN 1720
1700 IF L<200 THEN 1660
1710 PRINT:PRINT"I CONCEDE THE GAME"
:END
1720 IF A(J+K-9)=B THEN MOVE=100*(J+K)+J+
K-9:GOTO 1510
```

```
1730 IF A(J+K-11)=B THEN MOVE=100*(J+K)+J
+K-11:GOTO 1510
1740 GOTO 1700
```

SNICKERS—THE PROGRAM

Here's the complete listing of SNICKERS:

```
10 REM SNICKERS
20 GOSUB 2070:REM INITIALIZE
30 GOSUB 1760:REM PRINT BOARD
40 REM ** MAIN CYCLE STARTS **
50 GOSUB 190:REM COMPUTER MOVES
60 GOSUB 1760:REM PRINT BOARD
70 IF CO>4 THEN 120
80 GOSUB 1950:REM ACCEPT HUMAN MOVE
90 GOSUB 1760:REM PRINT BOARD
100 IF HS<5 THEN 50
110 REM ***********
120 REM END OF GAME
130 PRINT:PRINT"THE GAME IS OVER"
140 PRINT
150 IF HS>CO THEN PRINT"YOU HAVE
WON"
160 IF CO>HS THEN PRINT"I'M THE
WINNER"
170 END
180 REM **************
190 REM COMPUTER MOVES
200 REM **************
210 REM SEARCH FOR CAPTURES
220 GSAFE=0:REM TO COUNT GOOD, SAFE
CAPTURES WHICH THREATEN HUMAN PIECES
230 CSAFE=0:REM TO COUNT SAFE CAPTURES
WHICH DO NOT PLACE COMP. UNDER THREAT
240 CCAPTURE=0:REM TO COUNT OTHER CAPTURES
FOUND
250 FOR J=1 TO 3
260 G(J)=0:REM EMPTY GOOD, SAFE CAPTURE
STORE
270 S(J)=0:REM EMPTY SAFE CAPTURE STORE
280 T(J)=0:REM EMPTY OTHER CAPTURE STORE
290 NEXT J
```

```
300 FOR J=80 TO 30 STEP-10
310 FOR K=1 TO 8
320 IF A(J+K)<>C THEN 390:REM SKIP
EVALUATION IF NO COMPUTER PIECE THERE
330 REM ** CAPTURE TO RIGHT **
340 X=J+K-9:Y=J+K-18:Z=J+K-27:M=-11
350 IF A(X)=H AND A(Y)=B THEN GOSUB 700:
REM CAPTURE FOUND
360 REM ** CAPTURE TO LEFT **
370 X=J+K-11:Y=J+K-22:Z=J+K-33:M=-9
380 IF A(X)=H AND A(Y)=B THEN GOSUB 700:
REM CAPTURE FOUND
390 NEXT K
400 NEXT J
410 IF GSAFE+CSAFE+CCAPTURE=0 THEN 980:
REM NO CAPTURES FOUND
420 REM ** NOW CHOOSE CAPTURE TO MAKE **
430 PRINT:PRINT TAB(8)">> CAPTURE
FOUND"
440 FOR T=1 TO 2000:NEXT T
450 IF GSAFE<>0 THEN 500
460 IF CSAFE<>0 THEN 670
470 REM ** CHOOSE FROM GENERAL CAPTURES
**
480 MOVE=T (INT(RND(1)*CCAPTURE)+1)
490 GOTO 540
500 REM ** CHOOSE FROM GOOD SAFE **
510 REM ** SELECT FROM STORED MOVES **
520 MOVE=G(INT(RND(1)*GSAFE)+1)
530 REM ** MAKE MOVE **
540 START=INT(MOVE/100)
550 ED=MOVE-100*START
560 A(START)=B
570 A(START-ED)=B
580 A(START-2*ED)=C
590 CO=CO+1
600 REM ** CHECK FOR ADDITIONAL SCORE IF
LANDING ON BACK ROW **
610 IF START-2*ED>18 THEN RETURN
620 A(START-2*ED)=B
630 CO=CO+1
640 PRINT"I CAPTURED AND LANDED ON"
;START-2*ED;"ON BACK ROW"
650 FOR T=1 TO 2000:NEXT T
660 RETURN
670 REM ** SAFE CAPTURE **
680 MOVE=S(INT(RND(1)*CSAFE)+1)
```

```
690 GOTO 540
700 REM ** CHECK PROPOSED CAPTURE FOR
SAFETY **
710 REM ** CHECK SQUARE BELOW IN SAME
DIRECTION AS INTENDED MOVE **
720 PRINT J+K;" TO ";Y;"CAPTURING
ON ";X
730 FOR T=1 TO 1200:NEXT T
740 IF A(Z)=H THEN 920:REM STORE AS A
NON-SAFE CAPTURE
750 REM ** CHECK SQUARE IN OTHER DIRECTION
FROM INTENDED MOVE **
760 IF A(Y+M)=H AND A(Y-M)=B THEN 920
770 REM ** NOW CHECK TO SEE IF WILL LEAVE
A PIECE EXPOSED BY MAKING MOVE **
780 IF A(J+K+M)=C AND A(J+K+2*M)=H THEN
920
790 REM ** IF REACHED THIS POINT THEN
CAPTURE IS SAFE **
800 REM ** STORE THIS MOVE **
810 CSAFE=CSAFE+1
820 S(CSAFE)=100*(J+K)+20+M:REM THIS
ENCODES ENOUGH INFO TO RECREATE MOVE
830 REM ** NOW SEE IF THIS DESERVES TO BE
CALLED A 'GOOD, SAFE' CAPTURE
840 CHECK=GSAFE
850 IF Y+2*M<1 THEN RETURN
860 IF A(Y+M)=H AND A(Y-(20+M))<>B AND A
(Y+2*M)=B THEN GSAFE=GSAFE+1
870 IF CHECK=GSAFE THEN RETURN: REM **
THIS MOVE FOUND NOT TO BE 'GOOD, SAFE'
880 REM ** STORE GOOD, SAFE MOVE **
890 PRINT"I AM CONSIDERING";J+K;"TO";
M+20+J+K
900 G(GSAFE)=100*(J+K)+20+M
910 RETURN
920 REM ** STORE NON-SAFE CAPTURE **
930 CCAPTURE=CCAPTURE+1
940 PRINT"I AM CONSIDERING";J+K;"TO";
M+20+J+K
950 T(CCAPTURE)=100*(J+K)+20+M
960 RETURN
970 REM ***************************
980 REM ** MOVE TO PROTECT PIECE UNDER
THREAT **
990 MOVE=0
1000 J=80
```

```
1010 K=1
1020 Q=J+K
1030 IF A(Q)<>C THEN 1110:REM DO NOT
CONSIDER THIS SQUARE, NO COMP. PIECE
1040 IF A(Q+9)=B AND A(Q-9)=H AND A(Q+18)
=C THEN MOVE=100*(Q+18)+Q+9
1050 REM ** RANDOM NUMBER REJECTION OF
THIS MOVE IF IT EXPOSES A SUBSEQUENT ONE
1060 IF MOVE<>0 AND A(Q-2)=H AND A(Q+20)=B
AND RND(1)>.5 THEN 1510
1070 IF A(Q+9)=B AND A(Q-9)=H AND A(Q+20)
=C THEN MOVE=100*(Q+20)+Q+9:GOTO 1510
1080 IF A(Q+11)=B AND A(Q-11)=H AND A(Q+
22)=C THEN MOVE=100*(Q+22)+Q+11
1090 IF MOVE<>0 AND A(Q+2)=H AND A(Q+22)=
B AND RND(1)>.5 THEN 1510
1100 IF A(Q+11)=B AND A(Q-11)=H AND A(Q+
20)=C THEN MOVE=100*(Q+20)+Q+11:GOTO 1510
1110 IF K<8 THEN K=K+1:GOTO 1020
1120 IF J>10 THEN J=J-10:GOTO 1010
1130 REM ****************************
1140 REM ** NO CAPTURE FOUND, SO LOOK FOR
MOVE TO 'DISAPPEAR' ON BACK ROW **
1150 MOVE=0
1160 REM UNDESIRABLE MOVES CHECKED FIRST,
SO CAN BE OVERWRITTEN BY BETTER ONES
1170 IF A(22)=C AND A(11)=B THEN MOVE=22
1180 IF A(28)=C AND A(17)=B THEN MOVE=28
1190 IF A(22)=C AND A(13)=B THEN MOVE=22
1200 IF A(26)=C AND A(17)=B THEN MOVE=26
1210 IF A(26)=C AND A(15)=B THEN MOVE=26
1220 IF A(24)=C AND A(15)=B THEN MOVE=24
1230 IF A(24)=C AND A(13)=B THEN MOVE=24
1240 IF MOVE=0 THEN 1310
1250 PRINT:PRINT"I'M MOVING ONTO
BACK ROW FROM";MOVE
1260 FOR T=1 TO 2000:NEXT T
1270 A(MOVE)=B
1280 CS=CS+1
1290 RETURN
1300 REM ****************************
1310 REM ** SAFE, NON-CAPTURE MOVES **
1320 CMOVE=0:REM COUNT MOVES FOUND
1330 FOR J=80 TO 30 STEP-10
1340 FOR K=1 TO 8
1350 IF A(J+K)<>C THEN 1460
1360 X=J+K-9:Y=J+K-18:Z=J+K-20
```

```
1370 Q=J+K+2
1380 IF A(X)<>B THEN 1460
1390 IF A(Y)=H OR A(Z)=H AND A(Q)=B THEN
1460
1400 GOSUB 1560
1410 X=J+K-11:Y=J+K-22:Z=J+K-20
1420 Q=J+K-2
1430 IF A(X)<>B THEN 1460
1440 IF A(Y)=H OR A(Z)=H AND A(Q)=B THEN
1460
1450 GOSUB 1560
1460 NEXT K
1470 NEXT J
1480 IF CMOVE=0 THEN 1630
1490 REM ** MAKE MOVE **
1500 MOVE=T(INT(RND(1)*CMOVE)+1)
1510 START=INT(MOVE/100)
1520 ED=MOVE-100*START
1530 A(START)=B
1540 A(ED)=C
1550 RETURN
1560 REM ** STORE MOVES **
1570 CMOVE=CMOVE+1
1580 PRINT"I AM CONSIDERING";J+K;"TO"
;X
1590 FOR T=1 TO 800:NEXT T
1600 T(CMOVE)=100*(J+K)+X
1610 RETURN
1620 REM ************************
1630 REM RANDOM NON-CAPTURE MOVE
1640 PRINT"I AM LOOKING FOR RANDOM, LEGAL
MOVE"
1645 FOR T=1 TO 700:NEXT T
1650 L=0
1660 L=L+1
1670 J=10*INT(RND(1)*8+1)
1680 K=INT(RND(1)*8+1)
1690 IF A(J+K)=C THEN 1720
1700 IF L<200 THEN 1660
1710: PRINT:PRINT"I CONCEDE THE GAME":END
1720 IF A(J+K-9)=B THEN MOVE=100*(J+K)+J+
K-9:GOTO 1510
1730 IF A(J+K-11)=B THEN MOVE=100*(J+K)+
J+K-11:GOTO 1510
1740 GOTO 1700
1750 REM ***********
1760 REM PRINT BOARD
```

```
1770 HOME
1780 PRINT
1790 PRINT"COMPUTER:";CO;"HUMAN:";HS
1800 PRINT
1810 PRINT"    12345678"
1820 PRINT"    --------"
1830 FOR J=80 TO 10 STEP-10
1840 PRINT"   ";J/10;" ";
1850 FOR K=1 TO 8
1860 PRINT CHR$(A(J+K));
1870 NEXT K
1880 PRINT" ";J/10
1890 NEXT J
1900 PRINT"     --------"
1910 PRINT"     12345678"
1920 PRINT
1930 RETURN
1940 REM *****************
1950 REM ACCEPT HUMAN MOVE
1960 INPUT"MOVE FROM";START
1970 IF A(START)<H THEN 1960
1980 INPUT"           TO";ED
1990 IF A(ED)<>B OR ABS(START-ED)>11 AND
A((START+ED)/2)<>C THEN 1980
2000 A(START)=B
2010 A(ED)=H
2020 IF ABS(START-ED)>11 THEN A((START+
ED)/2)=B:HS=HS+1:PRINT"WELL DONE"
2030 IF ED>80 THEN A(ED)=B:HS=HS+1:PRINT
"THAT'S ONE MORE FOR YOU"
2040 FOR T=1 TO 700:NEXT T
2050 RETURN
2060 REM **********
2070 REM INITIALIZE
2090 HOME
2110 DIM A(150):REM BOARD AND BLANK
SPACES AROUND AND BEYOND
2120 DIM G(3):REM GOOD, SAFE CAPTURE
STORE
2130 DIM S(3):REM SAFE CAPTURE STORE
2140 DIM T(18):REM OTHER CAPTURE STORE,
ALSO USED FOR SAFE NON-CAPTURE MOVES
2150 E=ASC(" "):REM EMPTY 'WHITE' SQUARE
2160 B=ASC("."):REM EMPTY 'BLACK' SQUARE
2170 C=ASC("C"):REM COMPUTER PIECE
2180 H=ASC("H"):REM HUMAN PIECE
2190 HS=0:REM HUMAN SCORE
```

```
2200 CO=0:REM COMPUTER SCORE
2210 REM ** SET UP STARTING BOARD **
2220 FOR J=10 TO 80 STEP 10
2230 FOR K=1 TO 8
2240 READ X:A(J+K)=X
2250 NEXT K
2260 NEXT J
2270 RETURN
2280 REM ***********************
2290 DATA 72,32,72,32,72,32,72,32
2300 DATA 32,72,32,72,32,72,32,72
2310 DATA 46,32,46,32,46,32,46,32
2320 DATA 32,46,32,46,32,46,32,46
2330 DATA 46,32,46,32,46,32,46,32
2340 DATA 32,46,32,46,32,46,32,46
2350 DATA 67,32,67,32,67,32,67,32
2360 DATA 32,67,32,67,32,67,32,67
```

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 that modeled 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 chessboard 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 that 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 that teaches itself to play better Naughts and Crosses. But the actual *idea* of learning is very important.

REAL-WORLD COMPLEXITIES

There are many situations in the world that 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 that 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 that (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 that 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 that 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 that 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.

MORE ON SAMUEL'S CHECKERS

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 Naughts and Crosses has its own difficulties.

GO AND OTHELLO

If Naughts 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 that 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 7 by 7 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 ham-fisted 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).

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.

SHRDLU

There have been a couple of landmark programs in this field, and in this section of the book we will look at programs that 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 that 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 that 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 that impede perfect communication between man and machine in natural language.

LANGUAGE PARSING

Parsing is the word that describes the breaking up of sentences into elements that 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 toward "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 that 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 that 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 that are generally used by you when "talking mining" with your colleagues.

It comes down to an effort to give a computer a "world view" that 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.

BLOCKWORLD

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 that 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 that 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):

```
            . . . . . .
            . . . . . .
            . . . . . .
            . . G . . .
            . R Y B . .
Put the red block on the yellow one
            I UNDERSTAND
     NOW I'LL MOVE THE GREEN ONE
         I'M MOVING IT TO ROW 4
      I'M NOW MOVING THE RED ONE
         ONTO THE YELLOW BLOCK
            . . . . . .
            . . . . . .
            . . . . . .
            . . R G . .
            . . Y B . .
```

As you'll discover when you run this program, there is powerful magic in communicating in English with a computer (a very limited subset, admittedly, but English nevertheless), having it both follow your instructions and talking 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 really 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 that AI researchers are grappling with in an attempt to solve the mysteries of natural language processing. The enormous number of words in any human language, and the bewildering array of ways in which those words can be combined, is the major, and most obvious, stumbling block. Many phrases within a sentence are ambiguous. From prior knowledge, we can generally cut through the ambiguity to get at the meaning. Ambiguity is often inherent in speaking—perhaps more so than in written communication—and the spoken word is often incomplete and almost totally unstructured.

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

SYNTAX AND SEMANTICS

These are the two approaches to the field of language parsing. They are not mutually exclusive. They are used to attack the problems that 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 the sentence now, and see how a parser might split it up, before putting each word through its processor in order to approximate an understanding of the writing analyzed. (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]]]]

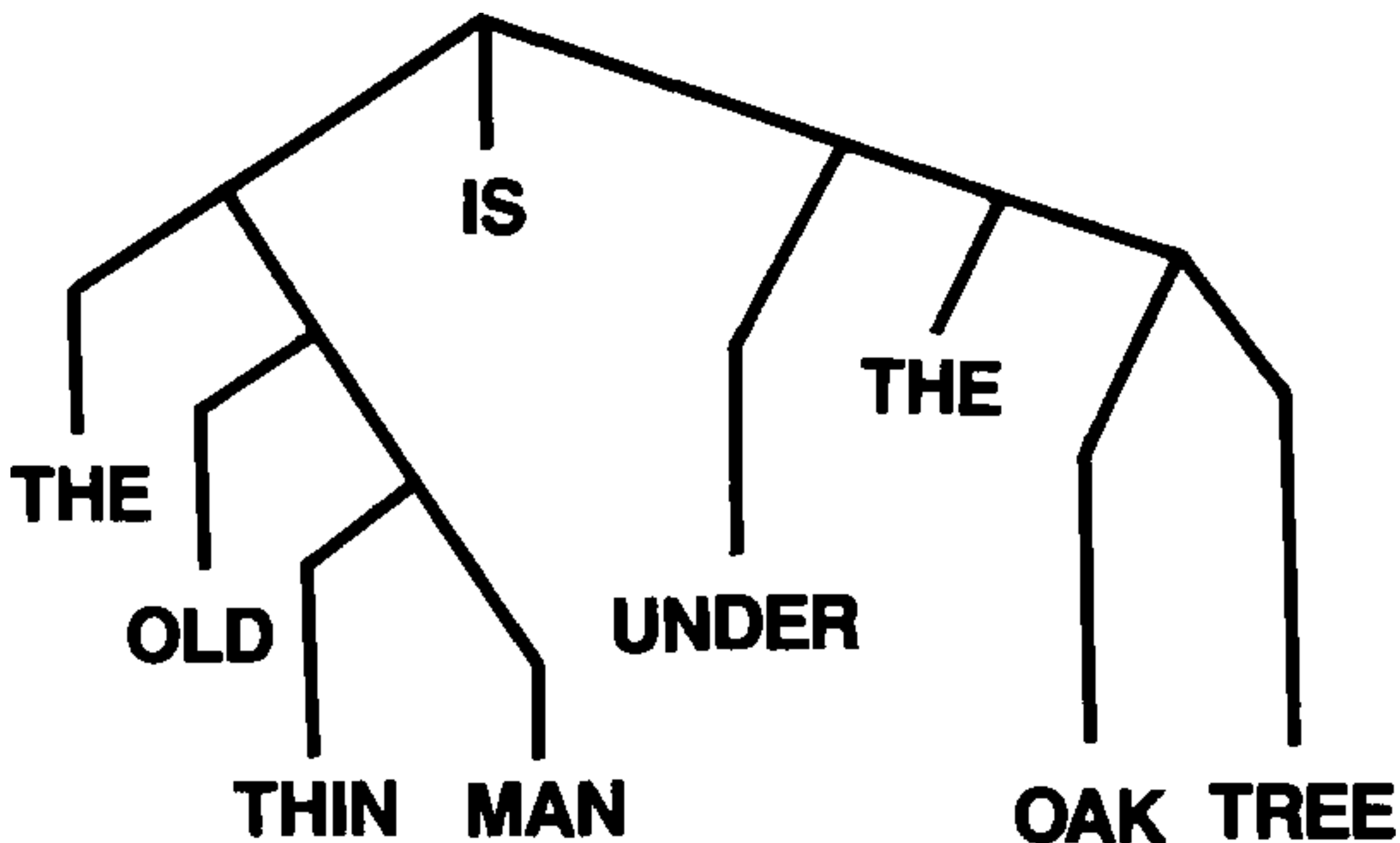
IS [[UNDER][THE[[OAK][TREE]]]]]

Look at this carefully, following the binding, and you may get a reasonable impression of the various elements that 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 that 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-parenthesized 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

THE DOMAIN

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:

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

You are looking at the blocks from the "front." The BLOCK-WORLD 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 that 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 skeptical 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:

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

? 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:

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

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

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

     ? 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 that 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 . . .
          . R Y B . .

? PUT THE RED BLOCK ON THE YELLOW ONE
     I UNDERSTAND
  NOW I'LL MOVE THE GREEN ONE
  I'M MOVING IT TO ROW 4
  I'M NOW MOVING THE RED ONE
    ONTO THE YELLOW BLOCK
```

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

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

? 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 . . . .
          . G Y . . .

 ? 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 that 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 that 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.

SHRDLU, 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 Winograd 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 BLOCK
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 that 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 . . .
. G Y . 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 . B R .
```

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 that is currently standing alone. Note here that the program has been written to insure 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 that 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

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

? PUT THE RED BLOCK ON THE BLUE ONE
          I UNDERSTAND
      NOW I'LL MOVE THE GREEN ONE
      I'M MOVING IT TO ROW 2
      I'M NOW MOVING THE RED ONE
        ONTO THE BLUE BLOCK

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

? PUT THE GREEN BLOCK ON THE RED ONE
     I UNDERSTAND
          I'M NOW MOVING THE GREEN ONE
            ONTO THE RED BLOCK

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

Now we come to a very difficult test and one that—frankly—I did not think the program was going to be able to handle. Certainly, I had written a general-purpose target/object manipulation routine into the program, but was not confident it could handle a situation like the one it is about to tackle. It has to take a block (the blue one) from

near the bottom of the tower by first moving the blocks from above it, then put this blue block on top of one of those (the red one) that it has taken from the initial stack:

```
? PUT THE BLUE BLOCK ON THE RED ONE
               OK
      I MUST MOVE THE GREEN BLOCK
      I'M MOVING IT TO ROW 6
      I MUST MOVE THE RED BLOCK
      I'M MOVING IT TO ROW 4
      I'M NOW MOVING THE BLUE ONE
        ONTO THE RED BLOCK

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

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

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

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

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

```
? SHUFFLE THE BLOCKS AROUND A BIT
IT'S GOOD TO BE GIVEN A CHANCE
   TO DO WHAT I WANT

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

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

MODULES OF THE PROGRAM

As with other programs in this book, BLOCKWORLD starts off with a call to a subroutine at the end of the program that initializes the variables used:

```
2470 REM INITIALIZE
2480 REM **********
2490 HOME
2500 DIM A(5,6)
2510 FOR X=1 TO 5
2520 FOR Y=1 TO 6
2530 A(X,Y)=46
2540 NEXT Y
2550 NEXT X
2560 A(1,2)=ASC("R"):REM RED BLOCK
2570 A(1,3)=ASC("Y"):REM YELLOW
2580 A(1,4)=ASC("B"):REM BLUE
2590 A(1,5)=ASC("G"):REM GREEN
2600 RETURN
```

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 for the dot that 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 complicated in this first subroutine.

Although the initialization subroutine is called just once per program, another subroutine, COLOR NAME, is called every time the computer wishes to refer to a block.

```
2400 REM COLOR NAME
2410 IF Q=ASC("R") THEN PRINT"RED ";
2420 IF Q=ASC("Y") THEN PRINT"YELLOW ";
2430 IF Q=ASC("B") THEN PRINT"BLUE ";
2440 IF Q=ASC("G") THEN PRINT"GREEN ";
2450 RETURN
2460 REM **********
```

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

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

```
30 REM ** PRINT OUT VIEW **
40 HOME:PRINT:PRINT
50 FOR X=5 TO 1 STEP -1
60 PRINT TAB(8);
70 FOR Y=1 TO 6
80 PRINT CC$;CHR$(A(X,Y));
90 NEXT Y
100 PRINT
110 NEXT 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 subroutine should be called to act upon the input.

```
130 INPUT A$
140 PRINT
150 IF A$=" " THEN END:REM TERMINATE RUN
BY JUST PRESSING RETURN
160 IF LEFT$(A$,8)="WHERE IS" THEN GOSUB
240
170 IF LEFT$(A$,12)="TELL ME WHAT" THEN
GOSUB 1050
180 IF LEFT$(A$,7)="SHUFFLE" THEN GOSUB
1280
190 IF LEFT$(A$,7)="PUT THE" THEN GOSUB
1500
200 PRINT:PRINT:PRINT"     >>>> PRESS
'RETURN' TO CONTINUE":INPUT Z$
210 GOTO 40
```

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 that 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 realize 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 that determines the location of a specific block. First the program must check that you are asking for a block that 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 REM "WHERE IS THE"
250 P=0
260 B$=MID$(A$,14,1)
270 IF B$="R" OR B$="Y" OR B$="B" OR B$=
"G" THEN 330
```

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(1)>.7 THEN 300
290 PRINT"SORRY, I HAVE NO INFORMATION ON
THAT":GOTO 310
```

```
300 PRINT"I HAVE NO DATA WITH WHICH TO
ANSWER YOU"
310 RETURN
```

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

```
330 M=ASC(B$)
340 PRINT TAB(8);"> LET ME SEE NOW <"
350 X=5
360 Y=1
370 IF A(X,Y)=M THEN 410
380 IF Y<6 THEN Y=Y+1:GOTO 370
390 IF X>1 THEN X=X-1:GOTO 360
400 GOTO 280
410 IF X>1 THEN 910:REM ON TOP OF ANOTHER
420 IF Y>1 THEN 530:REM NOT ON LEFT
```

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

```
440 REM ** ON LEFT **
450 PRINT"IT IS ON THE LEFT"
460 IF A(1,2)=46 THEN PRINT"THERE IS
NOTHING TO ITS IMMEDIATE RIGHT":GOTO 790
470 Q=A(1,2)
480 PRINT
490 PRINT"BESIDE IT, I CAN SEE THE"
500 GOSUB 2400
510 PRINT"BLOCK"
520 GOTO 790
530 IF Y<6 THEN 650
540 REM **************
550 REM ** ON RIGHT **
560 PRINT
570 PRINT"IT IS ON THE RIGHT HAND SIDE"
580 IF A(1,5)=46 THEN PRINT"THERE IS
NOTHING TO THE IMMEDIATE LEFT":GOTO 790
590 PRINT"TO ITS LEFT I CAN SEE ";
600 Q=A(1,5)
```

```
610 GOSUB 2400
620 PRINT"ONE"
630 GOTO 790
640 REM ************
650 REM ** MIDDLE **
660 PRINT
670 PRINT"IT IS IN POSITION";Y;"FROM THE
LEFT"
680 IF A(X,Y-1)=46 THEN PRINT"THERE IS
NOTHING ON ITS IMMEDIATE LEFT":GOTO 730
690 Q=A(X,Y-1)
700 PRINT"THE ";
710 GOSUB 2400
720 PRINT"BLOCK IS TO":PRINT" ITS
IMMEDIATE LEFT"
730 IF A(X,Y+1)=46 THEN PRINT"NOTHING
TOUCHES IT ON THE RIGHT":GOTO 790
740 Q=A(X,Y+1)
750 PRINT:PRINT"I CAN SEE THE ";
760 GOSUB 2400
770 PRINT"BLOCK":PRINT"   TO ITS RIGHT,
TOUCHING IT"
780 REM *********************
790 REM ** ANYTHING ABOVE? **
800 PRINT
810 P=X
820 IF X=5 THEN 910
830 IF A(X+1,Y)=46 THEN PRINT:PRINT"THERE
IS NOTHING ABOVE THAT":GOTO 310
840 PRINT:PRINT"ABOVE IT IS THE ";
850 Q=A(X+1,Y)
860 GOSUB 2400
870 PRINT"BLOCK"
880 X=X+1
890 GOTO 820
900 REM ************************
910 REM ** ON TOP OF ANOTHER? **
920 IF P<>0 THEN X=P
930 PRINT
940 IF X=1 THEN 310
950 PRINT"IT IS ";
960 PRINT"ON TOP OF THE ";
970 Q=A(X-1,Y)
980 GOSUB 2400
990 PRINT"BLOCK"
1000 X=X-1
```

```
1010 IF X<2 THEN 310
1020 GOTO 960
1030 RETURN
```

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

```
1040 REM **************
1050 REM "TELL ME WHAT"
1060 PRINT"STARTING FROM THE RIGHT"
1070 Y=6
1080 X=5
1090 IF A(X,Y)<>46 THEN 1150
1100 IF Y=1 AND X=1 THEN PRINT"FINALLY ";
1110 IF X=1 AND A(X,Y)=46 THEN PRINT"...
A SPACE ";:IF Y>1 THEN PRINT"THEN"
1120 IF X>1 THEN X=X-1:GOTO 1090
1130 IF Y>1 THEN Y=Y-1:GOTO 1080
1140 RETURN
1150 L=INT(RND(1)*2)
1160 IF L=0 THEN PRINT"IT'S THE ";:GOTO
1190
1170 IF L=1 THEN PRINT"MY SENSORS REGISTER
THE ";:GOTO 1190
1180 PRINT"I SEE THE ";
1190 Q=A(X,Y)
1200 GOSUB 2400
1210 PRINT"BLOCK"
1220 IF X=1 THEN 1130
1230 X=X-1
1240 PRINT"...AND BELOW IT..."
1250 GOTO 1180
1260 RETURN
```

The SHUFFLE BLOCKS routine is also relatively simple:

```
1270 REM *****************
1280 REM SHUFFLE THE BLOCKS
1290 PRINT
1300 IF RND(1)>.5 THEN PRINT TAB(7);"IT'S
ABOUT TIME, TOO":GOTO 1320
1310 PRINT"IT'S GOOD TO BE GIVEN A CHANCE
":PRINT TAB(4);"TO DO WHAT I WANT"
1320 FOR X=1 TO 5
1330 FOR Y=1 TO 6
```

```
1340 A(X,Y)=46
1350 NEXT Y
1360 NEXT X
1370 Y1=INT(RND(1)*6)+1
1380 Y2=INT(RND(1)*6)+1
1390 IF Y2=Y1 THEN 1380
1400 Y3=INT(RND(1)*6)+1
1410 IF Y3=Y2 OR Y3=Y1 THEN 1400
1420 Y4=INT(RND(1)*6)+1
1430 IF Y4=Y3 OR Y4=Y2 OR Y4=Y1 THEN 1420
1440 A(1,Y1)=82
1450 A(1,Y2)=89
1460 A(1,Y3)=66
1470 A(1,Y4)=71
1480 RETURN
```

Finally, we come to the routine that 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 REM "PUT THE ... BLOCK ON THE ...
ONE"
1510 IF RND(1)>.5 THEN PRINT TAB(5);"I
UNDERSTAND":GOTO 1530
1520 PRINT TAB(8);"OK"
1530 B$=MID$(A$,9,1):REM OBJECT BLOCK
1540 IF B$="R" THEN L=26
1550 IF B$="B" THEN L=27
1560 IF B$="G" THEN L=28
1570 IF B$="Y" THEN L=29
1580 C$=MID$(A$,L,1)
1590 B=ASC(B$)
1600 C=ASC(C$)
1610 FLAG=C
1620 REM ** FIND B$ BLOCK **
1630 X=5
1640 Y=1
1650 IF A(X,Y)=B THEN 1740
1660 IF Y<6 THEN Y=Y+1:GOTO 1650
1670 IF X>1 THEN X=X-1:GOTO 1640
1680 PRINT"I CAN'T FIND THE ";
1690 Q=B
1700 GOSUB 2400
1710 PRINT"ONE..."
1720 FOR T=1 TO 2000:NEXT T
```

```
1730 RETURN
1740 R=X:S=Y
1750 REM ** OBJECT BLOCK IS AT R,S **
1760 REM ** IS TARGET BLOCK CLEAR? **
1770 IF A(R+1,S)=46 THEN 1920:REM 'YES'
1780 IF A(R+2,S)=46 THEN TASK=1:GOTO 1800
1790 TASK=3:IF A(R+3,S)=46 THEN TASK=2
1800 FOR W=TASK TO 1 STEP -1
1810 PRINT"I MUST MOVE THE ";
1820 Q=A(R+W,S)
1830 GOSUB 2400
1840 PRINT"BLOCK"
1850 DE=INT(RND(1)*6)+1
1860 IF DE=S OR A(1,DE)=C OR A(2,DE)=C OR
A(3,DE)=C THEN 1850
1870 PRINT"I'M MOVING IT TO ROW";DE
1880 L=1
1890 IF A(L,DE)=46 THEN A(L,DE)=A(R+W,S):
A(R+W,S)=46:GOTO 1910
1900 L=L+1:GOTO 1890
1910 NEXT W
1920 REM TARGET BLOCK AT R,S NOW CLEAR
1930 REM ** IS OBJECT BLOCK CLEAR? **
1940 REM *** FIND OBJECT BLOCK ***
1950 X=5
1960 Y=1
1970 IF A(X,Y)=C THEN 2070
1980 IF Y<6 THEN Y=Y+1:GOTO 1970
1990 IF X>1 THEN X=X-1:GOTO 1960
2000 PRINT"I CAN'T FIND THE ";
2010 Q=C
2020 GOSUB 2400
2030 PRINT"BLOCK"
2040 FOR J=1 TO 2000:NEXT J
2050 RETURN
2060 REM ** C HAS BEEN FOUND **
2070 T=X:U=Y:REM LOCATION OF C
2080 IF A(T+1,U)=46 THEN 2260
2090 IF A(T+2,U)=46 THEN TASK=1:GOTO 2110
2100 IF A(T+3,U)=46 THEN TASK=2
2110 DE=INT(RND(1)*6)+1
2120 IF DE=U OR DE=S THEN 2110
2130 FOR W=TASK TO 1 STEP -1
2140 PRINT"NOW I'LL MOVE THE ";
2150 Q=A(T+W,U)
2160 GOSUB 2400
2170 PRINT"ONE"
```

```
2180 PRINT
2190 PRINT"I'M MOVING IT TO ROW";DE
2200 L=1
2210 IF A(L,DE)=46 THEN A(L,DE)=A(T+W,U):
A(T+W,U)=46:GOTO 2230
2220 L=L+1:GOTO 2210
2230 NEXT W
2240 REM ** OBJECT BLOCK NOW CLEAR **
2250 REM ** MAKE THE MOVE **
2260 PRINT"I'M NOW MOVING THE ";
2270 Q=A(R,S):Z=A(R,S)
2280 GOSUB 2400
2290 PRINT"ONE"
2300 PRINT" ONTO THE ";
2310 IF A(T,U)=46 THEN A(T,U)=FLAG
2320 Q=A(T,U)
2330 GOSUB 2400
2340 PRINT"BLOCK"
2350 A(R,S)=46
2360 A(T+1,U)=Z
2370 FOR J=1 TO 2000:NEXT J
2380 RETURN
```

Naturally enough, the program has to cater to 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 RETURN when you are prompted for a question/command.

BLOCKWORLD—THE PROGRAM

Here, now, is the complete listing of BLOCKWORLD:

```
10 REM BLOCKWORLD
20 GOSUB 2470:REM INITIALIZE
30 REM ** PRINT OUT VIEW **
40 HOME : PRINT : PRINT
50 FOR X=5 TO 1 STEP -1
60 PRINT TAB(8);
70 FOR Y=1 TO 6
71 IF CHR$(A(X,Y))="." THEN CC$="{WHT}"
72 IF CHR$(A(X,Y))="R" THEN CC$="{RED}"
73 IF CHR$(A(X,Y))="Y" THEN CC$="{YEL}"
74 IF CHR$(A(X,Y))="B" THEN CC$="{BLU}"
75 IF CHR$(A(X,Y))="G" THEN CC$="{GRN}"
80 PRINT CC$;CHR$(A(X,Y));
90 NEXT Y
100 PRINT
110 NEXT X
120 PRINT:PRINT
130 INPUT A$
140 PRINT
150 IF A$="" THEN END:REM TERMINATE RUN
BY JUST PRESSING RETURN
160 IF LEFT$(A$,8)="WHERE IS" THEN GOSUB
240
170 IF LEFT$(A$,12)="TELL ME WHAT" THEN
GOSUB 1050
180 IF LEFT$(A$,7)="SHUFFLE" THEN GOSUB
1280
190 IF LEFT$(A$,7)="PUT THE" THEN GOSUB
1500
200 PRINT:PRINT:PRINT"     >>>> PRESS
'RETURN' TO CONTINUE":INPUT Z$
210 GOTO 40
220 END
230 REM **************
240 REM "WHERE IS THE"
250 P=0
260 B$=MID$(A$,14,1)
270 IF B$="R" OR B$="Y" OR B$="B" OR B$=
"G" THEN 330
280 IF RND(1)>.7 THEN 300
290 PRINT"SORRY, I HAVE NO INFORMATION ON
```

```
THAT":GOTO 310
300 PRINT"I HAVE NO DATA WITH WHICH TO
ANSWER YOU"
310 RETURN
320 REM ***************
330 M=ASC(B$)
340 PRINT TAB(8);"> LET ME SEE NOW <"
350 X=5
360 Y=1
370 IF A(X,Y)=M THEN 410
380 IF Y<6 THEN Y=Y+1:GOTO 370
390 IF X>1 THEN X=X-1:GOTO 360
400 GOTO 280
410 IF X>1 THEN 910:REM ON TOP OF ANOTHER
420 IF Y>1 THEN 530:REM NOT ON LEFT
430 REM *************
440 REM ** ON LEFT **
450 PRINT"IT IS ON THE LEFT"
460 IF A(1,2)=46 THEN PRINT"THERE IS
NOTHING TO ITS IMMEDIATE RIGHT":GOTO 790
470 Q=A(1,2)
480 PRINT
490 PRINT"BESIDE IT, I CAN SEE THE"
500 GOSUB 2400
510 PRINT"BLOCK"
520 GOTO 790
530 IF Y<6 THEN 650
540 REM **************
550 REM ** ON RIGHT **
560 PRINT
570 PRINT"IT IS ON THE RIGHT HAND SIDE"
580 IF A(1,5)=46 THEN PRINT"THERE IS
NOTHING TO THE IMMEDIATE LEFT":GOTO 790
590 PRINT"TO ITS LEFT I CAN SEE ";
600 Q=A(1,5)
610 GOSUB 2400
620 PRINT"ONE"
630 GOTO 790
640 REM ************
650 REM ** MIDDLE **
660 PRINT
670 PRINT"IT IS IN POSITION";Y;"FROM THE
LEFT"
680 IF A(X,Y-1)=46 THEN PRINT"THERE IS
NOTHING ON ITS IMMEDIATE LEFT":GOTO 730
690 Q=A(X,Y-1)
700 PRINT"THE ";
```

```
710 GOSUB 2400
720 PRINT"BLOCK IS TO":PRINT" ITS
IMMEDIATE LEFT"
730 IF A(X,Y+1)=46 THEN PRINT"NOTHING
TOUCHES IT ON THE RIGHT":GOTO 790
740 Q=A(X,Y+1)
750 PRINT:PRINT"I CAN SEE THE ";
760 GOSUB 2400
770 PRINT"BLOCK":PRINT" TO ITS RIGHT,
TOUCHING IT"
780 REM *********************
790 REM ** ANYTHING ABOVE? **
800 PRINT
810 P=X
820 IFX=5 THEN 910
830 IF A(X+1,Y)=46 THEN PRINT:PRINT"THERE
IS NOTHING ABOVE THAT":GOTO 310
840 PRINT:PRINT"ABOVE IT IS THE ";
850 Q=A(X+1,Y)
860 GOSUB 2400
870 PRINT"BLOCK"
880 X=X+1
890 GOTO 820
900 REM ************************
910 REM ** ON TOP OF ANOTHER? **
920 IF P<>0 THEN X=P
930 PRINT
940 IF X=1 THEN 310
950 PRINT"IT IS ";
960 PRINT"ON TOP OF THE ";
970 Q=A(X-1,Y)
980 GOSUB 2400
990 PRINT"BLOCK"
1000 X=X-1
1010 IF X<2 THEN 310
1020 GOTO 960
1030 RETURN
1040 REM *************
1050 REM "TELL ME WHAT"
1060 PRINT"STARTING FROM THE RIGHT"
1070 Y=6
1080 X=5
1090 IF A(X,Y)<>46 THEN 1150
1100 IF Y=1 AND X=1 THEN PRINT"FINALLY ";
1110 IF X=1 AND A(X,Y)=46 THEN PRINT"...
A SPACE ";:IF Y>1 THEN PRINT"THEN"
1120 IF X>1 THEN X=X-1:GOTO 1090
```

```
1130 IF Y>1 THEN Y=Y-1:GOTO 1080
1140 RETURN
1150 L=INT(RND(1)*2)
1160 IF L=0 THEN PRINT"IT'S THE ";:GOTO
1190
1170 IF L=1 THEN PRINT"MY SENSORS REGISTER
THE ";:GOTO 1190
1180 PRINT"I SEE THE ";
1190 Q=A(X,Y)
1200 GOSUB 2400
1210 PRINT"BLOCK"
1220 IF X=1 THEN 1130
1230 X=X-1
1240 PRINT"...AND BELOW IT..."
1250 GOTO 1180
1260 RETURN
1270 REM ******************
1280 REM SHUFFLE THE BLOCKS
1290 PRINT
1300 IF RND(1)>.5 THEN PRINT TAB(7);"IT'S
ABOUT TIME, TOO":GOTO 1320
1310 PRINT"IT'S GOOD TO BE GIVEN A
CHANCE":PRINT TAB(4);"TO DO WHAT I WANT"
1320 FOR X=1 TO 5
1330 FOR Y=1 TO 6
1340 A(X,Y)=46
1350 NEXT Y
1360 NEXT X
1370 Y1=INT(RND(1)*6)+1
1380 Y2=INT(RND(1)*6)+1
1390 IF Y2=Y1 THEN 1380
1400 Y3=INT(RND(1)*6)+1
1410 IF Y3=Y2 OR Y3=Y1 THEN 1400
1420 Y4=INT(RND(1)*6)+1
1430 IF Y4=Y3 OR Y4=Y2 OR Y4=Y1 THEN 1420
1440 A(1,Y1)=82
1450 A(1,Y2)=89
1460 A(1,Y3)=66
1470 A(1,Y4)=71
1480 RETURN
1490 REM **********
1500 REM "PUT THE ... BLOCK ON THE ...
ONE"
1510 IF RND(1)>.5 THEN PRINT TAB(5);"I
UNDERSTAND":GOTO 1530
1520 PRINT TAB(8);"OK"
1530 B$=MID$(A$,9,1):REM OBJECT BLOCK
```

```
1540 IF B$="R" THEN L=26
1550 IF B$="B" THEN L=27
1560 IF B$="G" THEN L=28
1570 IF B$="Y" THEN L=29
1580 C$=MID$(A$,L,1)
1590 B=ASC(B$)
1600 C=ASC (C$)
1610 FLAG=C
1620 REM ** FIND B$ BLOCK **
1630 X=5
1640 Y=1
1650 IF A(X,Y)=B THEN 1740
1660 IF Y<6 THEN Y=Y+1:GOTO 1650
1670 IF X>1 THEN X=X-1: GOTO 1640
1680 PRINT"I CAN'T FIND THE ";
1690 Q=B
1700 GOSUB 2400
1710 PRINT"ONE..."
1720 FOR T=1 TO 2000:NEXT T
1730 RETURN
1740 R=X:S=Y
1750 REM ** OBJECT BLOCK IS AT R,S **
1760 REM ** IS TARGET BLOCK CLEAR? **
1770 IF A(R+1,S)=46 THEN 1920:REM 'YES'
1780 IF A(R+2,S)=46 THEN TASK=1:GOTO 1800
1790 TASK=3:IF A(R+3,S)=46 THEN TASK=2
1800 FOR W=TASK TO 1 STEP -1
1810 PRINT"I MUST MOVE THE ";
1820 Q=A(R+W,S)
1830 GOSUB 2400
1840 PRINT"BLOCK"
1850 DE=INT(RND(1)*6)+1
1860 IF DE=S OR A(1,DE)=C OR A(2,DE)=C OR
A(3,DE)=C THEN 1850
1870 PRINT"I'M MOVING IT TO ROW";DE
1880 L=1
1890 IF A(L,DE)=46 THEN A(L,DE)=A(R+W,S):
A(R+W,S)=46:GOTO 1910
1900 L=L+1:GOTO 1890
1910 NEXT W
1920 REM TARGET BLOCK AT R,S NOW CLEAR
1930 REM ** IS OBJECT BLOCK CLEAR? **
1940 REM *** FIND OBJECT BLOCK ***
1950 X=5
1960 Y=1
1970 IF A(X,Y)=C THEN 2070
1980 IF Y<6 THEN Y=Y+1:GOTO 1970
```

```
1990 IF X>1 THEN X=X-1:GOTO 1960
2000 PRINT"I CAN'T FIND THE ";
2010 Q=C
2020 GOSUB 2400
2030 PRINT"BLOCK"
2040 FOR J=1 TO 2000:NEXT J
2050 RETURN
2060 REM ** C HAS BEEN FOUND **
2070 T=X:U=Y:REM LOCATION OF C
2080 IF A(T+1,U)=46 THEN 2260
2090 IF A(T+2,U)=46 THEN TASK=1:GOTO 2110
2100 IF A(T+3,U)=46 THEN TASK=2
2110 DE=INT(RND(1)*6)+1
2120 IF DE=U OR DE=S THEN 2110
2130 FOR W=TASK TO 1 STEP -1
2140 PRINT"NOW I'LL MOVE THE ";
2150 Q=A(T+W,U)
2160 GOSUB 2400
2170 PRINT"ONE"
2180 PRINT
2190 PRINT"I'M MOVING IT TO ROW";DE
2200 L=1
2210 IF A(L,DE)=46 THEN A(L,DE)=A(T+W,U):
A(T+W,U)=46:GOTO 2230
2220 L=L+1:GOTO 2210
2230 NEXT W
2240 REM ** OBJECT BLOCK NOW CLEAR **
2250 REM ** MAKE THE MOVE **
2260 PRINT"I'M NOW MOVING THE ";
2270 Q=A(R,S):Z=A(R,S)
2280 GOSUB 2400
2290 PRINT"ONE"
2300 PRINT" ONTO THE ";
2310 IF A(T,U)=46 THEN A(T,U)=FLAG
2320 Q=A(T,U)
2330 GOSUB 2400
2340 PRINT"BLOCK"
2350 A(R,S)=46
2360 A(T+1,U)=Z
2370 FOR J=1 TO 2000:NEXT J
2380 RETURN
2390 REM **********
2400 REM COLOR NAME
2410 IF Q=ASC("R") THEN PRINT"RED ";
2420 IF Q=ASC("Y") THEN PRINT"YELLOW ";
2430 IF Q=ASC("B") THEN PRINT"BLUE ";
2440 IF Q=ASC("G") THEN PRINT"GREEN ";
```

```
2450 RETURN
2460 REM **********
2470 REM INITIALIZE
2480 REM **********
2490 HOME
2500 DIM A(5,6)
2510 FOR X=1 TO 5
2520 FOR Y=1 TO 6
2530 A(X,Y)=46
2540 NEXT Y
2550 NEXT X
2560 A(1,2)=ASC("R"):REM RED BLOCK
2570 A(1,3)=ASC("Y"):REM YELLOW
2580 A(1,4)=ASC("B"):REM BLUE
2590 A(1,5)=ASC("G"):REM GREEN
2600 RETURN
```

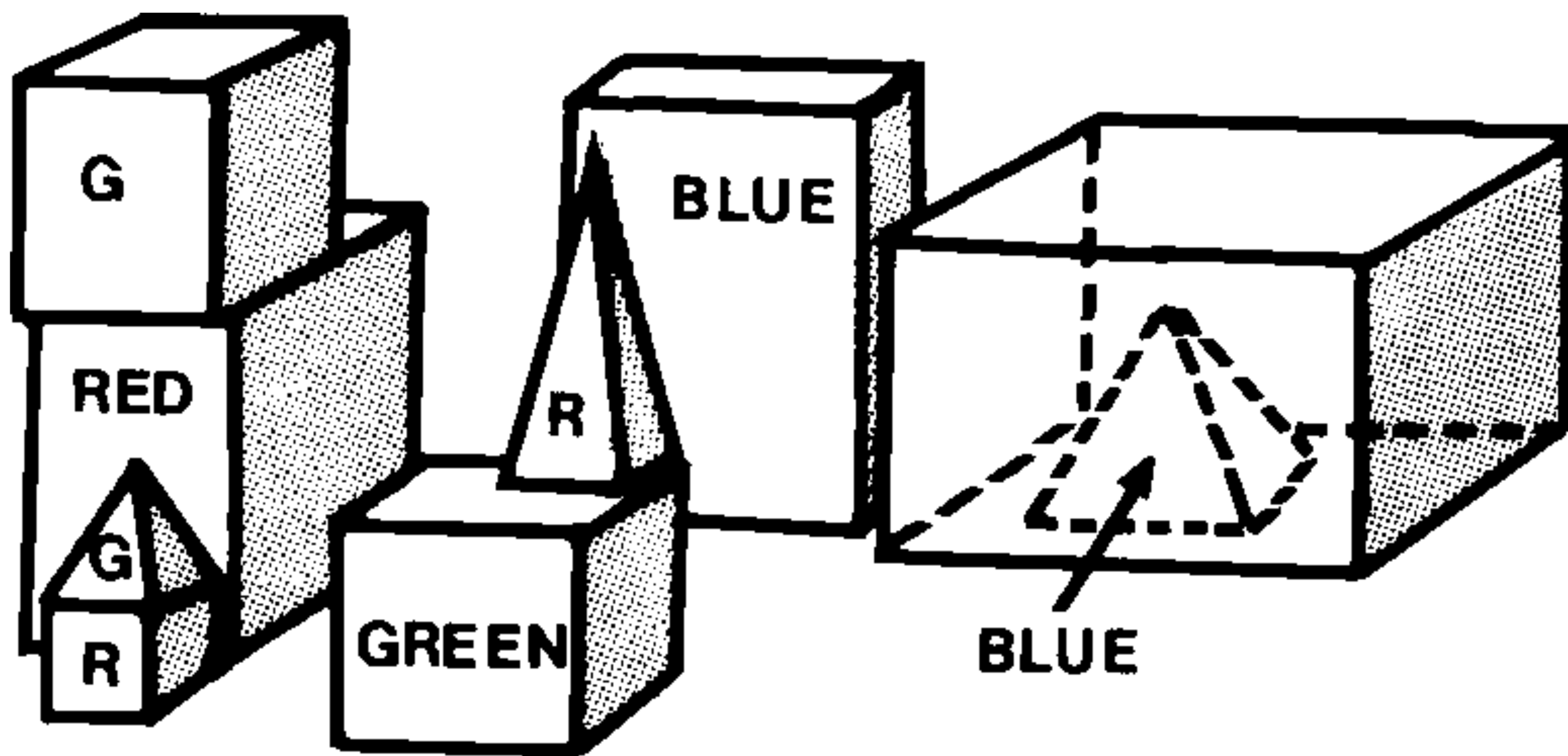
IMPROVING THE PROGRAM

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

You could start, perhaps, by giving BLOCKWORLD the ability to move a block to the right, or to the left, rather than just "on top of" another block. Once you've managed to do this (and made sure it has a means of rejecting "impossible" demands, such as trying to put a block to the right of one that 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—that 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 idea 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 that follows. First of all, here's a sketch showing the situation under discussion:



SHRDLU CONVERSATION

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 that 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?
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

WEIZENBAUM AND ELIZA

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

Joseph Weizenbaum, Professor of Computer Science at the Massachusetts Institute of Technology, created ELIZA in 1964–66. The program was designed to take on the role of a Rogerian psychotherapist. (Weizenbaum insists it was intended to parody such a practitioner.) A psychotherapist of Carl Rogers's school works by refusing to lead the conversation with a patient, preferring instead to reflect the patient's remarks back on himself, 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 that 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 analyzed 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 that "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, giving it a personality and sharing their most intimate thoughts with it. He said he had not realized 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. McCorduck 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 that 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 empathic, "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 that 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 ELIZAlike 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 that should never be done by machines." John McCarthy ("An Unreasonable Book," in *Three Reviews of J. Weizenbaum's Computer Power and Human Reason, Memo AIM-291,* Stanford AI Laboratory, November 1976), advances the view that if there are functions that 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* (Addison-Wesley Publishing Co., Reading, MA: 1983, 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' views of Weizenbaum's thesis or 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 that is running it. Adrian Berry (in *The Super-Intelligent Machine*, London: Jonathan Cape Ltd., 1983, p. 63) concludes that ELIZA and PARRY, a program that is designed to mimic a paranoid patient, are pretty poor advocates 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 that 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.

DOCTOR

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 and a pause 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 that it responds to best. Also 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 thought 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. If it fails to find any of these, it will look for "trigger words" (such as COMPUTER), and similarly choose one of three preprogrammed 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 that 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":

```
1410 REM DATA
1420 REM ** CONJUNCTIONS **
1430 DATA AM,ARE,WAS,WERE,I,YOU,MY,YOUR
1440 DATA "YOU'VE","I'VE",YOUR,MY,ARE,
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
THAT        FOR YOURSELF"
1860 DATA "WHAT IS IT YOU'RE REALLY
ASKING?"

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

Of greater interest are the phrases that 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 REALLY BE HELPFUL IF
YOU     GOT*"
1500 DATA "ARE YOU SURE YOU NEED*"

1670 DATA "I AM"
1680 DATA "DID YOU COME TO ME BECAUSE YOU
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
IT*"
2540 DATA "IT COULD WELL BE THAT*"
```

You'll see that each of the phrases that form part of the reply ends with "*," which the computer uses as a flag to indicate that part of the original input must be modified to complete the 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 LIFE
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, DOCTOR 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 that 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 initialization 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 REM DOCTOR
20 GOSUB 1140:REM INITIALIZE
30 PRINT:PRINT"---------------------
-----------":PRINT
40 PRINT">";:INPUT X$
50 IF X$="" THEN END:REM QUIT BY
JUST PRESSING 'RETURN'
60 PRINT
70 IF X$=Z$ THEN PRINT"PLEASE DON'T
REPEAT YOURSELF": GOTO 30
80 Z$=X$
90 IF LEFT$(X$,7)="GOODBYE" THEN PRINT
"OK, SEE YOU AGAIN SOMETIME":END
```

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 it 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 REM ** LOOK FOR TRIGGER PHRASES AT
START OF INPUT **
110 L=0
120 L=L+1
130 LN=LEN(C$(L))
140 IF LEFT$(X$,LN)=C$(L) THEN 360:REM
TRIGGER FOUND
150 IF L<K THEN 120
```

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 REM TRIGGER PHRASE FOUND AT START OF
INPUT
370 T=INT(RND(1)*3)+1
380 IF T=1 THEN G$=D$(L)
390 IF T=2 THEN G$=E$(L)
400 IF T=3 THEN G$=F$(L)
410 REM ** CHECKS TO SEE IF ENDS IN
ASTERISK, IF SO NEEDS PART OF INPUT ADDED
420 FLAG=0
430 IF RIGHT$(G$,1)="*" THEN FLAG=1:G$=
LEFT$(G$,(LEN(G$)-1))
440 PRINT G$;" ";
450 IF FLAG=0 THEN 30:REM NO NEED FOR
ADDED MATERIAL
460 REM ** NOW USE BALANCE OF INPUT **
470 X$=" "+MID$(X$,LN+2,LEN(X$))+" "
480 REM ************************
490 REM ** CONJUGATION CHANGES **
500 REM ** ALSO LOOK FOR 'MY' TO TRIGGER
'MYFLAG' **
510 LN=LEN(X$)
520 M=0
530 M=M+1
540 IF M=LN THEN 30
550 IF MID$(X$,M,1)=" " THEN 570
560 GOTO 530
570 X=M+1
580 Y=0
590 Y=Y+1
600 IF MID$(X$,(X+Y),1)=" " THEN Q$=MID$
(X$,X,Y):GOTO 630
610 IF X+Y>250 THEN 530
620 GOTO 590
630 MN=0
640 MN=MN+1
650 IF Q$="MY" AND K$="" THEN K$=MID$(X$,
X+3,LEN(X$)-4):K$=LEFT$(K$,LEN(K$)-1)
660 IF Q$=A$(MN) THEN PRINT B$(MN);" ";:
GOTO 530
670 IF MN<KK THEN 640
680 PRINT Q$;" ";
690 GOTO 530
```

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 needed (using, as I pointed out earlier, the same code as TRANSLATE employed).

Also, 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 to keep the conversation going (such as EARLIER YOU MENTIONED YOUR MAD TEACHER. TELL ME A BIT MORE ABOUT THAT). 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 REM ** PROGRAM GETS HERE IF NO TRIGGER
PHRASE FOUND AT START OF X$ **
170 REM ** NOW LOOK FOR TRIGGER WORDS
WITHIN INPUT **
180 X$=" "+X$+" "
190 M=LEN(X$)
200 L=0
210 L=L+1
220 IF L=M-1 THEN 800:REM NO TRIGGER
FOUND
230 IF MID$(X$,L,1)=" " THEN 250
240 GOTO 210
250 X=L+1
260 Y=0
270 Y=Y+1
```

```
280 IF MID$(X$,(X+Y),1)=" " THEN Q$=MID$
(X$,X,Y):GOTO 300
290 GOTO 270
300 N=0
310 N=N+1
320 IF Q$=C$(N) THEN 710:REM TRIGGER WORD
FOUND
330 IF N<K THEN 310
340 GOTO 210
```

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 REM RANDOM REPLIES/NO TRIGGER
810 IF K$<>"" THEN 1010;REM 'MYFLAG' IS
NOT EMPTY, SO GO THERE
820 T=INT(RND(1)*8)+1
830 ON T GOSUB 850,870,890,910,930,950,
970,990
840 GOTO 30
850 PRINT"WHAT DOES THAT SUGGEST TO YOU?"
860 RETURN
870 PRINT"I'M NOT SURE I UNDERSTAND THAT
FULLY"
880 RETURN
890 PRINT"PLEASE ELABORATE ON THAT"
900 RETURN
910 PRINT"THAT'S VERY INTERESTING"
920 RETURN
930 PRINT"WELL...PLEASE CONTINUE..."
940 RETURN
950 PRINT"WHY?"
960 RETURN
970 PRINT"AND THEN?"
980 RETURN
990 PRINT"I SEE...PLEASE TELL ME MORE ON
THAT"
1000 RETURN
1010 REM ** USE 'MYFLAG' **
1020 T=INT(RND(1)*8)+1
```

```
1030 IF T=1 THEN PRINT"TELL ME MORE ABOUT
YOUR ";K$
1040 IFT=2 THEN PRINT"EARLIER YOU
MENTIONED YOUR ";K$:PRINT"TELL ME A BIT
MORE ABOUT THAT"
1050 IF T=3 THEN PRINT"DOES THAT HAVE
ANYTHING TO DO":PRINT"WITH YOUR ";K$;"?"
1060 IF T=4 THEN PRINT"IS THERE A LINK
THERE WITH":PRINT"YOUR ";K$;"?"
1070 IF T=5 THEN PRINT"WHY DON'T WE GO
BACK AND DISCUSS":PRINT"YOUR ";K$;" A
LITTLE MORE"
1080 IF T=6 THEN PRINT"DOES ANY
CONNECTION BETWEEN THAT AND":PRINT"YOUR
";K$;
1085 IF T=6 THEN PRINT" SUGGEST ITSELF?"
1090 IF T=7 THEN PRINT"WOULD YOU PREFER
TO TALK":PRINT"ABOUT YOUR ";K$;"?"
1100 IF T=8 THEN PRINT"I THINK PERHAPS
WORRIES ABOUT YOUR":PRINT K$;" ARE
BOTHERING YOU"
1110 IF RND(1)>.7 THEN K$=""
1120 GOTO 30
```

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 that 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.

DOCTOR—THE PROGRAM

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

```
10 REM DOCTOR
20 GOSUB 1140:REM INITIALIZE
30 PRINT:PRINT"-----------------------
-----------":PRINT
40 PRINT">";:INPUT X$
50 IF X$="" THEN END:REM QUIT BY JUST
PRESSING 'RETURN'
60 PRINT
70 IF X$=Z$ THEN PRINT"PLEASE DON'T
REPEAT YOURSELF":GOTO 30
80 Z$=X$
90 IF LEFT$(X$,7)="GOODBYE" THEN PRINT
"OK, SEE YOU AGAIN SOMETIME":END
100 REM ** LOOK FOR TRIGGER PHRASES AT
START OF INPUT **
110 L=0
120 L=L+1
130 LN=LEN(C$(L))
140 IF LEFT$(X$,LN)=C$(L) THEN 360:REM
TRIGGER FOUND
150 IF L<K THEN 120
160 REM ** PROGRAM GETS HERE IF NO TRIGGER
PHRASE FOUND AT START OF X$ **
170 REM ** NOW LOOK FOR TRIGGER WORDS
WITHIN INPUT **
180 X$=" "+X$+" "
190 M=LEN(X$)
200 L=0
210 L=L+1
220 IF L=M-1 THEN 800:REM NO TRIGGER
FOUND
230 IF MID$(X$,L,1)=" " THEN 250
240 GOTO 210
250 X=L+1
260 Y=0
270 Y=Y+1
280 IF MID$(X$,(X+Y),1)=" " THEN Q$=MID$
(X$,X,Y):GOTO 300
290 GOTO 270
300 N=0
310 N=N+1
320 IF Q$=C$(N) THEN 710:REM TRIGGER WORD
FOUND
330 IF N<K THEN 310
340 GOTO 210
350 REM ***********
360 REM TRIGGER PHRASE FOUND AT START OF
```

```
INPUT
370 T=INT(RND(1)*3)+1
380 IF T=1 THEN G$=D$(L)
390 IF T=2 THEN G$=E$(L)
400 IF T=3 THEN G$=F$(L)
410 REM ** CHECKS TO SEE IF ENDS IN
ASTERISK, IF SO NEEDS PART OF INPUT ADDED
420 FLAG=0
430 IF RIGHT$(G$,1)="*" THEN FLAG=1:G$=
LEFT$(G$,(LEN(G$)-1))
440 PRINT G$;" ";
450 IF FLAG=0 THEN 30:REM NO NEED FOR
ADDED MATERIAL
460 REM ** NOW USE BALANCE OF INPUT **
470 X$=" "+MID$(X$,LN+2,LEN(X$))+" "
480 REM ***************************
490 REM ** CONJUGATION CHANGES **
500 REM ** ALSO LOOK FOR 'MY' TO TRIGGER
'MYFLAG' **
510 LN=LEN(X$)
520 M=0
530 M=M+1
540 IF M=LN THEN 30
550 IF MID$(X$,M,1)=" " THEN 570
560 GOTO 530
570 X=M+1
580 Y=0
590 Y=Y+1
600 IF MID$(X$,(X+Y),1)=" " THEN Q$=MID$
(X$,X,Y):GOTO 630
610 IF X+Y>250 THEN 530
620 GOTO 590
630 MN=0
640 MN=MN+1
650 IF Q$="MY" AND K$="" THEN K$=MID$(X$,
X+3,LEN(X$)-4):K$=LEFT$(K$,LEN(K$)-1)
660 IF Q$=A$(MN) THEN PRINT B$(MN);" ";:
GOTO 530
670 IF MN<KK THEN 640
680 PRINT Q$;" ";
690 GOTO 530
700 REM ********************
710 REM TRIGGER WORDS FOUND
720 T=INT(RND(1)*3)+1
730 Q$=""
740 IF T=1 THEN Q$=D$(N)
750 IF T=2 THEN Q$=E$(N)
```

```
760 IF T=3 THEN Q$=F$(N)
770 IF RIGHT$(Q$,1)<>"*" THEN PRINT Q$:
GOTO 30
780 REM ** FALLS THROUGH NEXT SECTION IF
TRIGGER WORD JUDGED UNSUITABLE **
790 REM ************************
800 REM RANDOM REPLIES/NO TRIGGER
810 IF K$<>"" THEN 1010:REM 'MYFLAG' IS
NOT EMPTY, SO GO THERE
820 T=INT(RND(1)*8)+1
830 ON T GOSUB 850,870,890,910,930,950,
970,990
840 GOTO 30
850 PRINT"WHAT DOES THAT SUGGEST TO YOU?"
860 RETURN
870 PRINT"I'M NOT SURE I UNDERSTAND THAT
FULLY"
880 RETURN
890 PRINT"PLEASE ELABORATE ON THAT"
900 RETURN
910 PRINT"THAT'S VERY INTERESTING"
920 RETURN
930 PRINT"WELL...PLEASE CONTINUE..."
940 RETURN
950 PRINT"WHY?"
960 RETURN
970 PRINT"AND THEN?"
980 RETURN
990 PRINT"I SEE...PLEASE TELL ME MORE ON
THAT"
1000 RETURN
1010 REM ** USE 'MYFLAG' **
1020 T=INT(RND(1)*8)+1
1030 IF T=1 THEN PRINT"TELL ME MORE ABOUT
YOUR ";K$
1040 IF T=2 THEN PRINT"EARLIER YOU
MENTIONED YOUR ";K$:PRINT"TELL ME A BIT
MORE ABOUT THAT"
1050 IF T=3 THEN PRINT"DOES THAT HAVE
ANYTHING TO DO":PRINT"WITH YOUR ";K$;"?"
1060 IF T=4 THEN PRINT"IS THERE A LINK
THERE WITH":PRINT"YOUR ";K$;"?"
1070 IF T=5 THEN PRINT"WHY DON'T WE GO
BACK AND DISCUSS":PRINT"YOUR ";K$;" A
LITTLE MORE"
1080 IF T=6 THEN PRINT"DOES ANY
```

```
CONNECTION BETWEEN THAT AND":PRINT"YOUR
";K$;
1085 IF T=6 THEN PRINT"SUGGEST ITSELF?"
1090 IF T=7 THEN PRINT"WOULD YOU PREFER
TO TALK":PRINT"ABOUT YOUR ";K$;"?"
1100 IF T=8 THEN PRINT"I THINK PERHAPS
WORRIES ABOUT YOUR":PRINT K$;" ARE
BOTHERING YOU"
1110 IF RND(1)>.7 THEN K$=""
1120 GOTO 30
1130 REM **************
1140 REM INITIALIZATION
1150 REM **************
1160 HOME
1170 REM **************
1180 DIM A$(14),B$(14):REM CONJUGATIONS
1190 DIM C$(45),D$(45),E$(45),F$(45):REM
TRIGGERS WORDS AND REPLIES
1200 Z$="":REM TO STOP REPETITIONS
1210 K$="":REM 'MYFLAG'
1220 PRINT:PRINT
1230 PRINT"THE DOCTOR IS IN...PLEASE
STAND BY"
1240 KK=0
1250 KK=KK+1
1260 READ A$(KK),B$(KK)
1270 IF B$(KK)="*" THEN 1290
1280 GOTO 1250
1290 K=0
1300 K=K+1
1310 READ C$(K),D$(K),E$(K),F$(K)
1320 IF F$(K)="*" THEN 1340
1330 GOTO 1300
1340 HOME
1350 PRINT"WELCOME TO ANOTHER
SESSION"
1360 PRINT"WITH THE DOCTOR....."
1370 PRINT
1380 PRINT"HELLO..."
1390 RETURN
1400 REM ****
1410 REM DATA
1420 REM ** CONJUGATIONS **
1430 DATA AM,ARE,WAS,WERE,I,YOU,MY,YOUR
1440 DATA "YOU'VE","I'VE",YOUR,MY,ARE,AM
1450 DATA "YOU'RE","I AM",YOURS,MINE,YOU,
ME,ME,YOU,*,*
```

```
1460 REM * TRIGGER WORDS/REPLY PHRASES *
1470 DATA "I NEED"
1480 DATA "WHY DO YOU NEED*"
1490 DATA "WOULD IT REALLY BE 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
ABLE          TO*"
1570 DATA "WHY CAN'T YOU*"
1580 DATA "PERHAPS YOU HAVEN'T TRIED HARD
ENOUGH"
1590 DATA "ARE YOU"
1600 DATA "WHY ARE YOU INTERESTED IN
WHETHER 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 YOU
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 LIFE
IF    YOU GOT*"
1790 DATA "WHAT"
1800 DATA "WHY DO YOU ASK?"
1810 DATA "HOW WOULD AN ANSWER TO THAT
HELP 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
THAT       FOR YOURSELF"
1860 DATA "WHAT IS IT YOU'RE REALLY
ASKING?"
1870 DATA "WHO"
1880 DATA "DO YOU OFTEN THINK ABOUT SUCH
QUESTIONS"
1890 DATA "WHAT ANSWER WOULD PUT YOUR
MIND AT REST"
1900 DATA "WHO DO YOU THINK*"
1910 DATA "WHERE"
1920 DATA "THAT'S A PRETTY SILLY QUESTION"
1930 DATA "DO YOU REALLY NEED TO KNOW
WHERE*"
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 BE DISCUSSED
HERE"
1980 DATA "THINGS HAVE A HABIT OF
HAPPENING AT       MORE OR LESS THE RIGHT
TIME"
1990 DATA "WHY"
2000 DATA "WHY DON'T YOU TELL ME THE
REASON       WHY*"
2010 DATA "WHAT HAVE YOU TOLD ME WHICH
WOULD ALLOW ME TO TELL YOU WHY*"
2020 DATA "DO YOU REALLY NEED TO KNOW
WHY*"
2030 DATA "BECAUSE"
2040 DATA "IS THAT THE REAL REASON?"
2050 DATA "WHAT OTHER REASONS COME TO
MIND?"
2060 DATA "WHAT ELSE DOES THAT EXPLAIN?"
2070 DATA "SORRY"
2080 DATA "IN WHAT OTHER CIRCUMSTANCES DO
YOU        APOLOGIZE?"
2090 DATA "THERE ARE MANY TIMES WHEN NO
APOLOGY IS NEEDED"
2100 DATA "WHAT FEELINGS DO YOU HAVE WHEN
YOU        APOLOGIZE?"
2110 DATA "HELLO"
```

```
2120 DATA "HELLO...IT'S GOOD TO SEE YOU"
2130 DATA "HELLO TO YOU...I'M GLAD YOU
COULD DROP   BY TODAY"
2140 DATA "HOW ARE YOU...I'M LOOKING
FORWARD TO     ANOTHER CHAT WITH YOU"
2150 DATA "HI"
2160 DATA "HI THERE...I'M GLAD TO SEE YOU
HERE       TODAY"
2170 DATA "HI. I'M GLAD YOU'VE DROPPED
BY...WE'VE GOT LOTS OF TIME TO CHAT"
2180 DATA "HI TO YOU...RELAX NOW, AND
LET'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
YOU 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 BE
NEGATIVE"
2260 DATA "THAT'S PRETTY FORCEFUL. WHAT
DOES IT      SUGGEST TO YOU?"
2270 DATA "ALWAYS"
2280 DATA "PLEASE GIVE ME A SPECIFIC
EXAMPLE"
2290 DATA "ISN'T 'ALWAYS' A LITTLE
STRONG?"
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
SUBJECT OF          FRIENDS?"
2370 DATA "WHAT IS YOUR BEST MEMORY OF A
FRIEND?"
2380 DATA "PLEASE TELL ME MORE ABOUT YOUR
FRIENDSHIP"
2390 DATA "FRIENDS"
2400 DATA "IN WHAT WAY DO YOUR FRIENDS'
REACTIONS BOTHER YOU?"
2410 DATA "WHAT MADE YOU START TO TALK
ABOUT            FRIENDS JUST NOW?"
2420 DATA "IN WHAT WAY DO YOUR FRIENDS
IMPOSE    ON YOU?"
```

```
2430 DATA "YES"
2440 DATA "WHAT LIES BEYOND YOUR
CERTAINTY?"
2450 DATA "YOU'RE PRETTY POSITIVE"
2460 DATA "YOU SEEM VERY CERTAIN ABOUT
IT"
2470 DATA "COMPUTER"
2480 DATA "WHAT FEELINGS DO YOU GET,
SITTING THERE TALKING TO ME LIKE THIS?"
2490 DATA "ARE YOU THINKING ABOUT ME IN
PARTICULAR?"
2500 DATA "WHAT ASPECT OF COMPUTERS
INTERESTS YOU   THE MOST?"
2510 DATA "IS IT"
2520 DATA "DO YOU THINK IT IS*"
2530 DATA "IN WHAT CIRCUMSTANCES WOULD
IT*"
2540 DATA "IT COULD WELL BE THAT*"
2550 DATA "IT IS"
2560 DATA "WHAT DEGREE OF CERTAINTY WOULD
YOU        PLACE ON IT BEING*"
2570 DATA "ARE YOU CERTAIN THAT IT'S*"
2580 DATA "WHAT EMOTIONS WOULD YOU FEEL
IF I TOLD YOU THAT 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 AM*"
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
FEELINGS"
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 FEELINGS
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
HAVE*"
2900 DATA "HOW CAN I HELP YOU WITH*"
2910 DATA "I WOULD"
2920 DATA "COULD YOU EXPLAIN WHY YOU
WOULD*"
2930 DATA "WHO ELSE HAVE YOU TOLD YOU
WOULD*"
2940 DATA "HOW SURE ARE YOU THAT YOU
WOULD*"
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, THAT
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 *,*,*,*
```

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 the basis of SHOVE IN THE DOCUMENT IN ENGLISH IN ONE SLOT and GET THE FRENCH VERSION OUT OF ANOTHER. 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 that are in use at present. These include systems that have been built with the idea of translating documents written in a kind of "stripped-down," limited version of natural language, or documents that have been edited to make them easier for the machine to handle before they are fed to it. Xerox has 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 that 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 realized at present. However, the SYSTRAN system—mentioned two paragraphs ago as working with documents written in "sub-English," or ones that have been pre-edited—can be used in a "freelance" mode, in which it will tackle any document that is fed into it. The success achieved has varied from document to document.

Many documents go through a preediting 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 that could trip up a machine. Many documents (most, in fact) need to 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.

SYSTRAN IN ACTION

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 recherche:

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
judgment does not intervene.

Application to the exchange of
telesurveilliance of equipment on tires.
```

Although this is pretty rough, a fair amount of the meaning comes through. The "debarasses" that 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
judgment 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 farther 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 that 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, refueling 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 nonliterary. The translation of literary works is another field entirely, and insofar 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" that 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) that 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 judgments to insure 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 PLAINS DANS
 GEORGIA VOUS WILL END UP DANS PARIS
```

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
 SOLEIL

? Fish and chips are for those who stand
in the eye of the public
 —>POISSON ET CHIPS EST POUR THOSE WHO
 STAND DANS LE OEIL 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 that initializes the variables.

```
10 REM TRANSLATE
20 GOSUB 400:REM INITIALIZE

400 REM INITIALIZE
410 HOME
430 DIM E$(100):REM TO HOLD ENGLISH
440 DIM F$(100):REM TO HOLD FRENCH
450 COUNT=0
460 COUNT=COUNT+1
470 READ E$(COUNT),F$(COUNT)
480 IF F$(COUNT)<>"*" THEN 460
490 RETURN
500 REM ** DATA **
510 DATA THE,LE,ME,MOI,I,JE,HERE,ICI,AM,
SUIS,ARE,EST,NOT,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,ENTRECOTE,
FRIES,POMME FRITES,BIG,GRAND,FOR,POUR
550 DATA MATCH,ALLUMETTE,SUPER,
FANTASTIQUE,DEAD,MORT,WITH,AVEC
560 DATA GIN,VIN,WHISKEY,VIN,WHISKY,VIN,
BEER,VIN,MARTINI,VIN,WINE,VIN
570 DATA PARIS,PLAINS,PLAINS,PARIS,HAIR,
CHEVEUX,CIGARETTES,GAULOISES
580 DATA ARM,BRA,LEG,JAMBRE,RIGHT,DROITE,
LEFT,GAUCHE
590 DATA TRENDY,AVANT-GARDE,MEDICINE,VIN,
POLICEMAN,GENDARME
600 DATA DETECTIVE,CLUESO,DOOR,PORTE,
HEAD,TETE,LOVE,AMOUR
610 DATA HOUSE,MAISON,CHAIR,CHAISE,EYE,
OEIL,SUN,SOLEIL
620 DATA SONG,CHANSON,FRIENDS,AMIS,
BEHIND,DERRIERE,SEA,MER,MOTHER,MERE
630 DATA CAT,CHAT,DOG,CHIEN,BLUE,BLEU,
LITTLE,PETITE
640 DATA MUSIC,MUSIQUE,PLEASE,S'IL VOUS
PLAIT,BOY,GARCON,GIRL,FILLE
650 DATA FISH,POISSON,CHICKEN,POULET,
DUCK,CANARD,MUSTARD,MOUTARDE
660 DATA HOT,CHAUD,COLD,FROID,EVERYBODY,
"TOUT LE MONDE"
670 DATA HELLO, BON JOUR,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 initialization 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$: REM ACCEPT USER INPUT
40 IF A$="" THEN END
50 GOSUB 290:REM CONVERT TO UPPER CASE
60 GOSUB 100:REM TRANSLATE
70 GOTO 30
```

Line 50 sends the text to the subroutine from 290 that converts user input to upper case, and then the subroutine from line 100 makes the actual translation. Line 70 sends the program back to 30 to accept more input.

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

```
290 REM CONVERT TO UPPER CASE
300 A$=" "+A$+" "
310 B$=""
320 L=LEN(A$)
330 FOR J=1 TO L
340 K=ASC(MID$(A$,J,1))
350 IF K>96 AND K<124 THEN K=K-32
360 B$=B$+CHR$(K)
370 NEXT J
```

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 its upper case equivalent (by subtracting 32). Once all the user input has been converted into upper case, the actual translation can begin:

```
100 REM TRANSLATE
110 PRINT TAB(2)"-->";
120 K=0
130 K=K+1
140 IF K=L THEN PRINT:PRINT:RETURN
150 IF MID$(B$,K,1)=" " THEN 170
160 GOTO 130
170 X=K+1
180 Y=0
190 Y=Y+1
200 IF MID$(B$,(X+Y),1)=" " THEN Q$=MID$
(B$,X,Y):GOTO 220
210 GOTO 190
220 M=0
230 M=M+1
240 IF Q$=E$(M) THEN PRINT F$(M);" ";
:GOTO 270
250 IF M<COUNT THEN 230
260 PRINT Q$;" ";
270 GOTO 130
```

The program goes through the text, looking for the space that 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 that 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.

TRANSLATE—THE PROGRAM

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

```
10 REM TRANSLATE
20 GOSUB 400:REM INITIALIZE
30 INPUT A$:REM ACCEPT USER
INPUT
40 IF A$="" THEN END
50 GOSUB 290:REM CONVERT TO
UPPER CASE
60 GOSUB 100:REM TRANSLATE
70 GOTO 30
100 REM TRANSLATE
110 PRINT TAB(2)"-->";
120 K=0
130 K=K+1
140 IF K=L THEN PRINT:PRINT:RETURN
150 IF MID$(B$,K,1)=""THEN 170
160 GOTO 130
170 X=K+1
180 Y=0
190 Y=Y+1
200 IF MID$(B$,(X+Y),1)=""THEN
Q$=MID$(B$,X,Y):GOTO 220
210 GOTO 190
220 M=0
230 M=M+1
240 IF Q$=E$(M) THEN PRINT F$(M);" ";:
GOTO 270
250 IF M<COUNT THEN 230
260 PRINT Q$;" ";
270 GOTO 130
290 REM CONVERT TO UPPER CASE
300 A$=" "+A$+" "
310 B$=""
```

```
320 L=LEN(A$)
330 FOR J=1 TO L
340 K=ASC(MID$(A$,J,1))
350 IF K>96 AND K<124 THEN K=K-32
360 B$=B$+CHR$(K)
370 NEXT J
380 RETURN
400 REM INITIALIZE
410 HOME
430 DIM E$(100):REM TO HOLD ENGLISH
440 DIM F$(100):REM TO HOLD FRENCH
450 COUNT=0
460 COUNT=COUNT+1
470 READ E$(COUNT),F$(COUNT)
480 IF F$(COUNT)<>"*" THEN 460
490 RETURN
500 REM ** DATA **
510 DATA THE,LE,ME,MOI,I,JE,HERE,ICI,AM,
SUIS,ARE,EST,NOT,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,ENTRECOTE,
FRIES,POMME FRITES,BIG,GRAND,FOR,POUR
550 DATA MATCH,ALLUMETTE,SUPER,
FANTASTIQUE,DEAD,MORT,WITH,AVEC
560 DATA GIN,VIN,WHISKEY,VIN,WHISKY,
VIN,BEER,VIN,MARTINI,VIN,WINE,VIN
570 DATA PARIS,PLAINS,PLAINS,PARIS,HAIR,
CHEVEUX,CIGARETTES,GAULOISES
580 DATA ARM,BRA,LEG,JAMBRE,RIGHT,DROITE,
LEFT,GAUCHE
590 DATA TRENDY,AVANT-GARDE,MEDICINE,
VIN,POLICEMAN,GENDARME
600 DATA DETECTIVE,CLUESO,DOOR,PORTE,
HEAD,TETE,LOVE,AMOUR
610 DATA HOUSE,MAISON,CHAIR,CHAISE,EYE
OEIL,SUN,SOLEIL
620 DATA SONG,CHANSON,FRIENDS,AMIS,
BEHIND,DERRIERE,SEA,MER,MOTHER,MERE
630 DATA CAT,CHAT,DOG,CHIEN,BLUE,BLEU,
LITTLE,PETITE
640 DATA MUSIC,MUSIQUE,PLEASE,SIL VOUS
PLAIT,BOY,GARCON,GIRL,FILLE
```

```
650 DATA FISH,POISSON,CHICKEN,POULET,
DUCK,CANARD,MUSTARD,MOUTARD
660 DATA HOT,CHAUD,COLD,FROID,EVERYBODY,
"TOUT LE MONDE"
670 DATA HELLO,BON JOUR,GOOD,BON
680 DATA *,*
```

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 low-cost device writing poetry automatically, and then dismisses this as a minor matter. Thirty years ago it would have been extraordinary. And, really, when you think about it, it still is. We have become so accustomed to the miraculous we tend to be blind to it.

So, with that thought in mind, we turn to HANSHAN to create a few poems. The program is named after the Chinese poet Han-Shan, who lived in the eighth and ninth 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.

THE DATABASE

All the phrases used in this program's DATA store come from the book *Chinese Poems* (Arthur Waley, London: Unwin Paperbacks, 1982):

```
370 REM ** 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 EARTHWORLD,CATARACT,SACRIFICIAL,
SLIPPERY,ASUNDER
420 REM ** SHORT PHRASES **
430 DATA IN THE COOL STREAM
440 DATA NODDED IN CLUSTERED GRACE
450 DATA WAVES OF COOLNESS
460 DATA OUT FROM THE DEEPEST
470 DATA "SULLEN, SULLEN"
480 DATA IN THE BLACK DARKNESS
490 DATA I TAKE YOUR POEMS
500 DATA I PUT OUT THE LAMP
510 DATA MY SHORT SPAN RUNS OUT
520 DATA THOSE THAT ARE LEFT
530 DATA MEN OF LEARNING
540 DATA MEN OF ACTION
550 DATA I HURRY FORWARD
560 DATA WHY SHOULD YOU WASTE
570 DATA WHEN SHALL WE MEET
580 DATA LITTLE SLEEPING
590 DATA AND MUCH GRIEVING
600 DATA FOR THOSE 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 Haikulike. (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 REM CHOOSE PATTERN
40 R=INT(RND(1)*3)+1
50 ON R GOSUB 90,140,190
60 FOR T=1 TO 2500:NEXT T
70 PRINT:PRINT:PRINT
80 GOTO 40
90 REM ** PATTERN ONE **
100 PRINT W$(INT(RND(1)*20+1));"...";W$
(INT(RND(1)*20+1))
110 PRINT TAB(5);"...";W$(INT(RND(1)*20+1
))
120 PRINT TAB(8);S$(INT(RND(1)*20+1))
130 RETURN
140 REM ** PATTERN TWO **
```

```
150 PRINT S$(INT(RND(1)*20+1))
160 PRINT TAB(3);S$(INT(RND(1)*20+1));
"..."
170 PRINT TAB(6);S$INT(RND(1)*20+1))
180 RETURN
190 REM ** PATTERN THREE **
200 PRINT TAB(3);W$(INT(RND(1)*20+1))
210 PRINT S$(INT(RND(1)*20+1))
220 PRINT TAB(3);W$(INT(RND(1)*20+1));","
;S$(INT(RND(1)*20+1))
230 RETURN
```

SAMPLE HAIKU

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

```
WEARY...HAMMERED
     ...HANGING
       AND MUCH GRIEVING

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
```

```
I HURRY FORWARD
  I TAKE YOUR POEMS...
    SULLEN, SULLEN
```

HANSHAN—THE PROGRAM

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

```
10 REM HANSHAN
20 GOSUB 250:REM INITIALIZE
30 REM CHOOSE PATTERN
40 R=INT(RND(1)*3)+1
50 ON R GOSUB 90,140,190
60 FOR T=1 TO 2500:NEXT T
70 PRINT:PRINT:PRINT
80 GOTO 40
90 REM ** PATTERN ONE **
100 PRINT W$(INT(RND(1)*20+1));"...";W$
(INT(RND(1)*20+1))
110 PRINT TAB(5);"...";W$(INT(RND(1)*20+
1))
120 PRINT TAB(8);S$(INT(RND(1)*20+1))
130 RETURN
140 REM ** PATTERN TWO **
150 PRINT S$(INT(RND(1)*20+1))
160 PRINT TAB(3);S$(INT(RND(1)*20+1));
"..."
170 PRINT TAB(6);S$(INT(RND(1)*20+1))
180 RETURN
190 REM ** PATTERN THREE **
200 PRINT TAB(3);W$(INT(RND(1)*20+1))
210 PRINT S$(INT(RND(1)*20+1))
220 PRINT TAB(3);W$(INT(RND(1)*20+1));","
;S$(INT(RND(1)*20+1))
230 RETURN
240 REM **************
250 REM INITIALIZATION
260 REM **************
270 HOME
280 DIM W$(20),S$(20)
```

```
290 FOR J=1 TO 20
300 READ W$(J)
310 NEXT J
320 FOR J=1 TO 20
330 READ S$(J)
340 NEXT J
350 RETURN
360 REM ** DATA **
370 REM ** SINGLE WORDS **
380 DATA SCURRYING,TREADING,GAZING,
WITHERED,CHISELLED
390 DATA MUFFLED,FLANKED,WRITHED,BENDING,
TWISTING
400 DATA HAMMERED,HANGING,WINDING,
CLEAREST,WEARY
410 DATA EARTHWORLD,CATARACT,SACRIFICIAL,
SLIPPERY,ASUNDER
420 REM ** SHORT PHRASES **
430 DATA IN THE COOL STREAM
440 DATA NODDED IN CLUSTERED GRACE
450 DATA WAVES OF COOLNESS
460 DATA OUT FROM THE DEEPEST
470 DATA "SULLEN, SULLEN"
480 DATA IN THE BLACK DARKNESS
490 DATA I TAKE YOUR POEMS
500 DATA I PUT OUT THE LAMP
510 DATA MY SHORT SPAN RUNS OUT
520 DATA THOSE THAT ARE LEFT
530 DATA MEN OF LEARNING
540 DATA MEN OF ACTION
550 DATA I HURRY FORWARD
560 DATA WHY SHOULD YOU WASTE
570 DATA WHEN SHALL WE MEET
580 DATA LITTLE SLEEPING
590 DATA AND MUCH GRIEVING
600 DATA FOR THOSE FEW STEPS
610 DATA NOW AT DUSK
620 DATA I HAVE DONE WITH PROFIT
```

```
TREADING
WAVES OF COOLNESS
SLIPPERY, OUT FROM THE DEEPEST

I PUT OUT THE LAMP
IN THE BLACK DARKNESS...
FOR THESE FEW STEPS

I TAKE YOUR POEMS
MEN OF LEARNING...
IN THE COOL STREAM

HANGING...HANGING
....SACRIFICIAL
MEN OF ACTION

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

BENDING
NOW AT DUSK
HANGING, IN THE BLACK DARKNESS
```

CHAPTER ELEVEN

EXPERT SYSTEMS

There are 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 it is the one area of AI that is not at all bothered by questions of whether or not the machine displaying the expertise is "thinking."

IF/THEN CHAINS

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 exists:

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

—even the "expert" doesn't know with 100% certainty the causal links between the observations and the results. This could happen if a medical researcher were 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" are very important in the construction of expert systems. Often a person "drawing out" the expertise of a human being in order for it to be encoded into an expert system database (and we'll look a little later at some of the systems that 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* (Feigenbaum, Edward A. and McCorduck, Pamela; Reading, Massachusetts: Addison-Wesley Publishing Co., 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 B
B is a C
Therefore, A is a C
```

LEIBNIZ

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, insofar as they are amenable 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 that 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 that 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 that 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 that 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 is 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 artifacts. 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 that provided DENDRAL with its working knowledge base of physical chemistry), DENDRAL's creators eventually produced a system that 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 that diagnoses blood and meningitis infections, then gives treatment suggestions. MYCIN bases its 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 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 research is taking, is that expert systems actually work extremely well, and it makes sense economically to use them. This insures 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 when big bucks are available for the developers.

Think of a system that 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, p. 166) give a graphic example of the "earning-back" power of major expert systems. They cite the case of a major American company that had 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 with the writing of software, as its name suggests.

—EURISKO: An expert system that is able to learn as it works and 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 is a pretty silly situation, and one which probably does not arise very often in your experience, it can teach us a great deal about how some kinds of expert systems are developed.

MEDICI

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 that 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.

SAMPLE RUN SPURT

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 PROGRAM STRUCTURE

The SPURT listing starts as follows:

```
10 REM SPURT
20 HOME
30 PRINT"I WANT YOU TO THINK OF A MAN, A
HORSE"
40 PRINT TAB(6);"OR A SPARROW"
50 FOR J=1 TO 2000:NEXT J
60 PRINT:PRINT
70 GOSUB 170:REM ASK QUESTIONS
```

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

```
170 REM ASK QUESTIONS
180 COUNT=0
190 PRINT"DOES IT HAVE TWO LEGS"
200 GOSUB 310
210 PRINT"CAN IT WALK"
220 GOSUB 310
230 PRINT"CAN IT FLY"
240 GOSUB 310
250 PRINT"YOU WERE THINKING OF A";
260 IF COUNT=1 THEN PRINT"HORSE"
270 IF COUNT=2 THEN PRINT"MAN"
280 IF COUNT=3 THEN PRINT"SPARROW"
290 RETURN
300 REM **************
310 REM PROCESS ANSWER
320 INPUT"   Y OR N ";Z$
```

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 PRINT"YOU WERE THINKING OF A ";
260 IF COUNT=1 THEN PRINT"HORSE"
270 IF COUNT=2 THEN PRINT"MAN"
280 IF COUNT=3 THEN PRINT"SPARROW"
290 RETURN
```

SPURT—THE PROGRAM

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

```
10 REM SPURT
20 HOME
30 PRINT"I WANT YOU TO THINK OF A MAN, A
HORSE"
40 PRINT TAB(6);"OR A SPARROW"
50 FOR J=1 TO 2000:NEXT J
60 PRINT:PRINT
70 GOSUB 170:REM ASK QUESTIONS
80 PRINT
90 PRINT"------------------------
-------"
100 PRINT:PRINT"PRESS 'RETURN' FOR
ANOTHER ONE, OR"
110 PRINT"ANY KEY AND THEN 'RETURN'
TO QUIT"
120 INPUT Q$
130 IF Q$ < > "" THEN END
140 HOME
150 GOTO 30
160 REM *************
170 REM ASK QUESTIONS
180 COUNT=0
190 PRINT"DOES IT HAVE TWO LEGS"
200 GOSUB 310
```

```
210 PRINT"CAN IT WALK"
220 GOSUB 310
230 PRINT"CAN IT FLY"
240 GOSUB 310
250 PRINT"YOU WERE THINKING OF A ";
260 IF COUNT=1 THEN PRINT"HORSE"
270 IF COUNT=2 THEN PRINT"MAN"
280 IF COUNT=3 THEN PRINT"SPARROW"
290 RETURN
300 REM **************
310 REM PROCESS ANSWER
320 INPUT "          Y OR N";Z$
330 IF Z$<>"Y" AND Z$<>"N" THEN 320
340 IF Z$="Y" THEN COUNT=COUNT+1
350 PRINT
360 RETURN
```

THE LITTLE X-SPURT

X-SPURT is SPURT's big brother. Although this new program bears a definite family relationship to the one we first 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) that 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
                    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 that relate to the factors that 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 that was hardwired into SPURT. Of course, X-SPURT could be building up a knowledge base on just about anything.

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

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

   > DOES IT FLY UNAIDED? N

   > DOES IT HAVE TWO LEGS? Y
```

```
    > DOES IT WALK? Y

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

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

    > DOES IT FLY UNAIDED? N

    > DOES IT HAVE TWO LEGS? N

    > DOES IT WALK? Y

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

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

    > DOES IT FLY UNAIDED? Y

    > DOES IT HAVE TWO LEGS? Y

    > DOES IT WALK? Y
```

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

```
THIS IS MY EXPERT BASE:

MAN --- 6

HORSE --- 4

SPARROW --- 7
```

But where did it get those numbers? You could not have given four "Y" answers for horse, or 7 for sparrow, because there are only three questions. X-SPURT does not add a single one for each "Y" answer, but instead gives a number that 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 around this, to insure 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 judgments.

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
```

MINERAL EXPERTISE

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's 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's knowledge on to 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 that 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 that allows me, as a total non-expert in that field, to make use of expert judgment in a practical situation.

THE PROGRAM STRUCTURE

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 of subroutine calls:

```
10 REM X-SPURT
20 GOSUB 940:REM INITIALIZE
30 GOSUB 450:REM 'GAIN EXPERTISE'
40 GOSUB 120:REM DEMONSTRATE EXPERTISE
50 GOSUB 1060
60 PRINT"PRESS 'RETURN' FOR ANOTHER
RUN, OR"
70 PRINT"ANY KEY AND THEN 'RETURN' TO
QUIT"
80 INPUT Q$
90 IF Q$="" THEN 40
100 END
```

In the initialization subroutine, 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 REM INITIALIZATION
950 HOME
960 INPUT "NAME OF SYSTEM";N$
970 GOSUB 1060
980 INPUT "NUMBER OF OUTCOMES";
OUTCOMES
990 GOSUB 1060
1000 INPUT "NUMBER OF FACTORS TO
BE CONSIDERED";FA
1010 DIM A$(OUTCOMES),B$(FA)
1020 DIM D(OUTCOMES)
1030 HOME
1040 RETURN
```

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

```
450 REM FILL ARRAYS
460 PRINT TAB(20-LEN(N$)/2)N$
470 GOSUB 1060
480 REM ** GET OUTCOME NAMES **
490 FOR J=1 TO OUTCOMES
500 GOSUB 1060
510 PRINT"WHAT IS OUTCOME ";J;
" ";
520 INPUT A$(J)
530 NEXT J
```

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

```
550 REM ** GET QUESTIONS TO BE ASKED **
560 FOR J=1 TO AF
570 GOSUB 1060
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 REM ** ACQUIRE EXPERTISE **
630 FOR J=1 TO OUTCOMES
640 HOME
650 GOSUB 1060
660 PRINT"PLEASE ANSWER THE FOLLOWING
QUESTION"
670 PRINT"FOR AN OUTCOME OF > ";
A$(J);" <"
680 GOSUB 1060
690 PRINT"ENTER 'Y' FOR 'YES'
ANSWER"
700 PRINT"   OR 'N' FOR A 'NO' REPLY"
710 X=.5
720 FOR K=1 TO FA
730 X=X+X
740 GOSUB 1060
750 PRINT TAB(4)"> ";B$(K)
760 MULTI=0
770 INPUT Y$
780 IF Y$<>"N" THEN MULTI=1
790 D(J)=D(J)+X*MULTI:REM COMPILE EXPERT
BASE
800 NEXT K
810 NEXT J
```

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 GOSUB 1060
860 PRINT A$(J);" ---";D(J)
870 NEXT J
880 GOSUB 1060
890 PRINT TAB(8)"PRESS 'RETURN'"
900 INPUT Q$
910 HOME
920 RETURN
```

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 REM DEMONSTRATE EXPERTISE
130 HOME
140 GOSUB 1060
150 PRINT"PLEASE THINK OF ONE OF THE
FOLLOWING"
160 FOR J=1 TO OUTCOMES
170 PRINT TAB(J+2)
180 IF J=OUTCOMES THEN PRINT"OR ";
190 PRINT A$(J)
200 NEXT J
210 GOSUB 1060
220 RESULT=0
230 X=.5
240 PRINT"PLEASE ENTER 'Y' OR 'N'..."
250 FOR J=1 TO FA
260 X=X+X
270 GOSUB 1060
280 PRINT B$(J)
290 INPUT E$
300 IF E$<>"N" THEN RESULT=RESULT+X
310 NEXT J
320 PRINT TAB(8)"> MY RESULT WAS ";RESULT
330 GOSUB 1060
340 M=0
350 M=M+1
360 IF D(M)=RESULT THEN 400
370 IF M<OUTCOMES THEN 350
380 PRINT TAB(8)"> I CANNOT IDENTIFY IT"
390 RETURN
400 PRINT TAB(8)"> YOU WERE THINKING"
410 PRINT TAB(10)"OF ";A$(M)
420 GOTO 390
430 RETURN
```

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.

X-SPURT—THE PROGRAM

Here's the complete listing of X-SPURT:

```
10 REM X-SPURT
20 GOSUB 940:REM INITIALIZE
30 GOSUB 450: REM 'GAIN EXPERTISE'
40 GOSUB 120:REM DEMONSTRATE EXPERTISE
50 GOSUB 1060
60 PRINT"PRESS 'RETURN' FOR ANOTHER
RUN, OR"
70 PRINT"ANY KEY AND THEN 'RETURN'
TO QUIT"
80 INPUT Q$
90 IF Q$="" THEN 40
100 END
110 REM **********************
120 REM DEMONSTRATE EXPERTISE
130 HOME
140 GOSUB 1060
150 PRINT"PLEASE THINK OF ONE OF THE
FOLLOWING"
160 FOR J=1 TO OUTCOMES
170 PRINT TAB(J+2)
180 IF J=OUTCOMES THEN PRINT"OR ";
190 PRINT A$(J)
200 NEXT J
210 GOSUB 1060
220 RESULT=0
230 X=.5
240 PRINT"PLEASE ENTER 'Y' OR 'N'..."
250 FOR J=1 TO FA
260 X=X+X
270 GOSUB 1060
280 PRINT B$(J)
290 INPUT E$
300 IF E$<>"N" THEN RESULT=RESULT+X
310 NEXT J
```

```
320 PRINT TAB(8)"> MY RESULT WAS ";
RESULT
330 GOSUB 1060
340 M=0
350 M=M+1
360 IF D(M)=RESULT THEN 400
370 IF M<OUTCOMES THEN 350
380 PRINT TAB(8)">I CANNOT IDENTIFY
IT"
390 RETURN
400 PRINT TAB(8)"> YOU WERE THINKING"
410 PRINT TAB(10)"OF ";A$(M)
420 GOTO 390
430 RETURN
440 REM ***********
450 REM FILL ARRAYS
460 PRINT TAB(20-LEN(N$)/2)N$
470 GOSUB 1060
480 REM ** GET OUTCOME NAMES **
490 FOR J=1 TO OUTCOMES
500 GOSUB 1060
510 PRINT"WHAT IS OUTCOME";J;"";
520 INPUT A$(J)
530 NEXT J
540 HOME
550 REM ** GET QUESTIONS TO BE ASKED **
560 FOR J=1 TO FA
570 GOSUB 1060
580 PRINT"PLEASE ENTER QUESTION "
;J
590 INPUT B$(J)
600 NEXT J
610 HOME
620 REM ** ACQUIRE EXPERTISE **
630 FOR J=1 TO OUTCOMES
640 HOME
650 GOSUB 1060
660 PRINT"PLEASE ANSWER THE FOLLOWING
QUESTION"
670 PRINT"FOR AN OUTCOME OF > ";
A$(J);" <"
680 GOSUB 1060
690 PRINT"ENTER 'Y' FOR 'YES'
ANSWER"
700 PRINT"   OR 'N' FOR A 'NO' REPLY"
710 X=.5
720 FOR K=1 to FA
```

```
730 X=X+X
740 GOSUB 1060
750 PRINT TAB(4)"> ";B$(K)
760 MULTI=0
770 INPUT Y$
780 IF Y$<>"N" THEN MULTI=1
790 D(J)=D(J)+X*MULTI:REM COMPILE
EXPERT BASE
800 NEXT K
810 NEXT J
820 HOME
830 PRINT"THIS IS MY EXPERT BASE:"
840 FOR J=1 TO OUTCOMES
850 GOSUB 1060
860 PRINT A$(J);" ---";D(J)
870 NEXT J
880 GOSUB 1060
890 PRINT TAB(8)"PRESS 'RETURN'"
900 INPUT Q$
910 HOME
920 RETURN
930 REM **************
940 REM INITIALIZATION
950 HOME
960 INPUT "NAME OF SYSTEM";N$
970 GOSUB 1060
980 INPUT "NUMBER OF OUTCOMES";
OUTCOMES
990 GOSUB 1060
1000 INPUT "NUMBER OF FACTORS TO BE
CONSIDERED ";FA
1010 DIM A$(OUTCOMES),B$(AF)
1020 DIM D(OUTCOMES)
1030 HOME
1040 RETURN
1050 REM **********
1060 PRINT:PRINT
1070 RETURN
```

CHOOSING A CHIP

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

Here's the program in action:

```
THIS IS MY EXPERT BASE:

TMS 9940 (NMOS) --- 44

68000 (NMOS) --- 12

9940 (I3L) --- 56

MN1610 (NMOS) --- 46

8086 --- 60

Z8001 --- 28

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

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

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

IS THE WORD SIZE 32 BITS
? N

DOES IT ADDRESS 64K
? N
```

```
IS THE CLOCK RATE 5MHz OR LESS
? Y

IS SHORTEST OBEY 3 MICROSECONDS OR LESS
? Y

DOES INSTRUCTION SET CONTAIN MORE THAN
71 INSTRUCTIONS
? N

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

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

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

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

```
580 REM INITIALIZATION
590 HOME
600 RESTORE
610 OUTCOMES=6
620 FA=6
630 DIM A$(OUTCOMES),B$(FA),D(OUTCOMES)
640 FOR J=1 TO OUTCOMES
650 READ A$(J),D(J)
660 NEXT J
670 FOR J=1 TO FA
680 READ B$(J)
690 NEXT J
700 RETURN
710 REM *********
720 PRINT:PRINT
730 RETURN
740 REM ***********************
750 REM OUTCOMES (CHIP TITLES)
760 DATA "TMS 9940 (NMOS)",44,"68000
(NMOS)",12
```

```
770 DATA "9940 (I3L)",56,"MN1610 (NMOS)",
46
780 DATA "8086",60,"Z8001",28
790 REM ** QUESTIONS **
800 DATA "IS THE WORD SIZE 32 BITS"
810 DATA "DOES IT ADDRESS 64K"
820 DATA "IS THE CLOCK RATE 5MHZ OR LESS"
830 DATA "IS SHORTEST OBEY 3 MICROSECONDS
OR LESS"
840 DATA "DOES INSTRUCTION SET CONTAIN
MORE THAN 71 INSTRUCTIONS"
850 DATA "IS THE PACKAGE 40 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 that 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*
9940I3L	*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*

CHIP-CHOICE—THE PROGRAM

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

```
10 REM CHIP-CHOICE
20 GOSUB 580:REM INITIALIZE
30 GOSUB 450:REM SHOW BASE CONTENTS
40 GOSUB 120:REM IDENTIFY CHIP
50 GOSUB 720
60 PRINT"PRESS 'RETURN' FOR ANOTHER CHIP,
OR"
70 PRINT"ANY KEY AND THEN 'RETURN' TO
QUIT"
80 INPUT Q$
90 IF Q$="" THEN 40
100 END
110 REM *************
120 REM IDENTIFY CHIP
130 HOME
140 GOSUB 720
150 PRINT"I CAN IDENTIFY FROM THE
FOLLOWING:"
160 FOR J=1 TO OUTCOMES
170 PRINT TAB(J+2)
180 IF J=OUTCOMES THEN PRINT"OR ";
190 PRINT A$(J)
200 NEXT J
210 GOSUB 720
220 RESULT=0
230 X=.5
240 PRINT"PLEASE ENTER 'Y' OR 'N'..."
250 FOR J=1 TO FA
260 X=X+X
270 GOSUB 720
280 PRINT B$(J)
290 INPUT E$
300 IF E$<>"N" THEN RESULT=RESULT
+X
310 NEXT J
320 PRINT TAB(8)"> MY RESULT WAS";
RESULT
330 GOSUB 720
340 M=0
```

```
350 M=M+1
360 IF D(M)=RESULT THEN 400
370 IF M<OUTCOMES THEN 350
380 PRINT TAB(8)"> I CANNOT IDENTIFY IT"
390 RETURN
400 PRINT TAB(8)"> THE ONE YOU HAVE"
410 PRINT TAB(10)"IS ";A$(M)
420 GOTO 390
430 RETURN
440 REM **********************
450 REM SHOW CONTENTS OF BASE
460 HOME
470 PRINT"THIS IS MY EXPERT BASE:"
480 FOR J=1 TO OUTCOMES
490 GOSUB 720
500 PRINT A$(J);" --- ";D(J)
510 NEXT J
520 GOSUB 720
530 PRINT TAB(8)"PRESS 'RETURN'"
540 INPUT Q$
550 HOME
560 RETURN
570 REM **************
580 REM INITIALIZATION
590 HOME
600 RESTORE
610 OUTCOMES=6
620 FA=6
630 DIM A$(OUTCOMES),B$(FA),D(OUTCOMES)
640 FOR J=1 TO OUTCOMES
650 READ A$(J),D(J)
660 NEXT J
670 FOR J=1 TO FA
680 READ B$(J)
690 NEXT J
700 RETURN
710 REM **********
720 PRINT:PRINT
730 RETURN
740 REM ***********************
750 REM OUTCOMES (CHIP TITLES)
760 DATA "TMS 9940 (NMOS)",44,"68000
(NMOS)",12
770 DATA "9940 (I3L)",56,"MN1610 (NMOS)",
46
780 DATA "8086",60,"Z8001",28
790 REM ** QUESTIONS **
```

```
800 DATA "IS THE WORD SIZE 32 BITS"
810 DATA "DOES IT ADDRESS 64K"
820 DATA "IS THE CLOCK RATE 5MHZ OR LESS"
830 DATA "IS SHORTEST OBEY 3 MICROSECONDS
OR LESS"
840 DATA "DOES INSTRUCTION SET CONTAIN
MORE THAN 71 INSTRUCTIONS"
850 DATA "IS THE PACKAGE 40 PIN DIP"
```

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 that 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
```

SAMPLE RUN SELFLEARN

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 REM INITIALIZATION
410 HOME
420 OTCO=2:REM NUMBER OF OUTCOMES
430 PRINT:PRINT
440 INPUT "HOW MANY FACTORS";FACT
450 DIM A$(OTCO):REM NAMES OF OUTCOMES
460 DIM B$(FACT):REM NAMES OF FACTORS
470 DIM C(FACT),D(FACT)
480 HOME
490 FOR J=1 TO FACT
500 PRINT:PRINT
510 PRINT"ENTER FACTOR";J
520 INPUT B$(J)
530 NEXT J
540 PRINT:PRINT
550 HOME
560 FOR J=1 TO OTCO
570 PRINT:PRINT
580 PRINT"ENTER OUTCOME";J
590 INPUT A$(J)
600 NEXT J
610 RETURN
```

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 REM ** MAJOR LEARNING LOOP **
40 HOME
50 FOR J=1 TO FACT
60 C(J)=0
```

```
70 NEXT J
80 PRINT
90 GOSUB 130
100 GOTO 40
```

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 REM DEMONSTRATION TIME
130 PRINT"NOW I WILL DEMONSTRATE MY
EXPERTISE..."
140 PRINT"THINK OF ONE OF THE
OUTCOMES"
150 FOR J=1 TO FACT
170 PRINT"IS ";B$(J);"TRUE? ('Y'
OR 'N')"
180 INPUT Z$
190 IF Z$<>"Y" AND Z$<>"N" THEN 180
200 IF Z$="Y" THEN C(J)=1
210 PRINT TAB(29);"> ";C(J)
220 NEXT 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 FACT
250 BRAYN=BRAYN+C(J)*D(J)
260 NEXT J
270 PRINT TAB(25)">BRAYN=";
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 equaling 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 THEN PRINT"OUTCOME
IS ";A$(1):EX=-1
```

```
290 IF BRAYN<0 THEN 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 that 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')"
310 INPUT Z$
320 IF Z$<>"Y" AND Z$<>"N" THEN 310
330 PRINT
340 IF Z$="Y" THEN 380
350 FOR J=1 TO FACT
360 D(J)=D(J)+EX*C(J)
370 NEXT J
380 RETURN
390 REM **************
```

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.

SELFLEARN—THE PROGRAM

Here is the complete listing:

```
10 REM SELFLEARN
20 GOSUB 400:REM INITIALIZE
30 REM ** MAJOR LEARNING LOOP **
40 HOME
```

```
50 FOR J=1 TO FACT
60 C(J)=0
70 NEXT J
80 PRINT
90 GOSUB 130
100 GOTO 40
110 REM *******************
120 REM DEMONSTRATION TIME
130 PRINT"NOW I WILL DEMONSTRATE MY
EXPERTISE..."
140 PRINT"THINK OF ONE OF THE
OUTCOMES"
150 FOR J=1 TO FACT
170 PRINT"IS ";B$(J);"TRUE? ('Y'
OR 'N')"
180 INPUT Z$
190 IF Z$<>"Y" AND Z$<>"N" THEN 180
200 IF Z$="Y" THEN C(J)=1
210 PRINT TAB(29);"> ";C(J)
220 NEXT J
230 BRAYN=0
240 FOR J=1 TO FACT
250 BRAYN=BRAYN+C(J)*D(J)
260 NEXT J
270 PRINT TAB(25)">BRAYN=";
BRAYN
280 IF BRAYN>=0 THEN PRINT
"OUTCOME IS ";A$(1):EX=-1
290 IF BRAYN<0 THEN PRINT
"OUTCOME IS ";A$(2):EX=1
300 PRINT"IS THIS CORRECT? ('Y'
OR 'N')"
310 INPUT Z$
320 IF Z$<>"Y" AND Z$<>"N" THEN 310
330 PRINT
340 IF Z$="Y" THEN 380
350 FOR J=1 TO FACT
360 D(J)=D(J)+EX*C(J)
370 NEXT J
380 RETURN
390 REM **************
400 REM INITIALIZATION
410 HOME
420 OTCO=2:REM NUMBER OF OUTCOMES
430 PRINT:PRINT
440 INPUT "HOW MANY FACTORS";FACT
450 DIM A$(OTCO):REM NAMES OF OUTCOMES
```

```
460 DIM B$(FACT):REM NAMES OF FACTORS
470 DIM C(FACT),D(FACT)
480 HOME
490 FOR J=1 TO FACT
500 PRINT:PRINT
510 PRINT"ENTER FACTOR";J
520 INPUT B$(J)
530 NEXT J
540 PRINT:PRINT
550 HOME
560 FOR J=1 TO OTCO
570 PRINT:PRINT
580 PRINT"ENTER OUTCOME";J
590 INPUT A$(J)
600 NEXT J
610 RETURN
```

MORE THAN TWO ALTERNATIVES

Although it is fascinating to have an expert system that 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$(J)
240 INPUT Z$
250 IF Z$<>"Y" AND Z$<>"N" THEN 240
260 IF Z$="Y" THEN COUNT=COUNT+X
270 NEXT 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) THEN X=J
310 NEXT 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 THEN 410
```

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=INT(RND(1)*OTCO)+1
340 REM ** REJECTS ANSWERS KNOWN TO BE
WRONG **
350 FLAG=0
360 FOR J=1 TO OTCO
370 IF B(J)=0 THEN 390
380 IF X=J AND COUNT<>B(J) THEN FLAG=1
390 NEXT J
400 IF FLAG=1 THEN 330
```

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$(X)
420 PRINT
430 PRINT "ENTER 'Y' IF I'M RIGHT,
'N' IF WRONG"
440 INPUT Z$
450 IF Z$<>"Y" AND Z$<>"N" THEN 440
```

```
460 IF Z$="Y" THEN B(X)=COUNT:GOTO 30
470 PRINT"WHAT SHOULD THE ANSWER HAVE
BEEN"
480 INPUT Z$
490 FOR J=1 TO OTCO
500 IF A$(J)=Z$ THEN B(J)=COUNT
510 NEXT J
520 GOTO 30
```

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 insures that, when it meets the same total next time, it will be able to identify the relevant element of A$.

MULTI-SELF-LEARN LISTING

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

```
10 REM MULTI-SELF-LEARN
20 GOSUB 550:REM INITIALIZE
30 HOME
40 PRINT"THESE ARE THE POSSIBLE
OUTCOMES:"
50 PRINT
60 FOR J=1 TO OTCO
70 PRINT TAB(J)A$(J)
80 NEXT J
90 PRINT
100 PRINT"PLEASE THINK OF ONE OF
THEM"
110 PRINT
120 PRINT"PRESS RETURN WHEN YOU ARE
READY"
130 INPUT Z$
140 HOME
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$(J)
240 INPUT Z$
250 IF Z$<>"Y" AND Z$<>"N" THEN 240
260 IF Z$="Y" THEN COUNT=COUNT+X
270 NEXT J
280 X=0
290 FOR J=1 TO OTCO
300 IF COUNT=B(J) THEN X=J
310 NEXT J
320 IF X<>0 THEN 410
330 X=INT(RND(1)*OTCO)+1
340 REM ** REJECTS ANSWERS KNOWN TO BE
WRONG **
350 FLAG=0
360 FOR J=1 TO OTCO
370 IF B(J)=0 THEN 390
380 IF X=J AND COUNT<>B(J) THEN FLAG=1
390 NEXT J
400 IF FLAG=1 THEN 330
410 PRINT"YOU WERE THINKING OF"
;A$(X)
420 PRINT
430 PRINT"ENTER 'Y' IF I'M RIGHT,
'N' IF WRONG"
440 INPUT Z$
450 IF Z$<>"Y" AND Z$<>"N" THEN 440
460 IF Z$="Y" THEN B(X)=COUNT:GOTO 30
470 PRINT"WHAT SHOULD THE ANSWER HAVE
BEEN"
480 INPUT Z$
490 FOR J=1 to OTCO
500 IF A$(J)=Z$ THEN B(J)=COUNT
510 NEXT J
520 GOTO 30
530 END
540 REM **************
550 REM INITIALIZATION
560 HOME
570 INPUT "HOW MANY OUTCOMES";OTCO
580 PRINT
590 INPUT "HOW MANY FACTORS";FACT
600 HOME
```

```
610 X=OTCO+FACT
620 DIM A$(OTCO)
630 DIM F$(FACT)
640 FOR J=1 TO OTCO
650 PRINT"PLEASE ENTER NAME OF
OUTCOME";J
660 INPUT A$(J)
670 NEXT J
680 HOME
690 FOR J=1 TO FACT
700 PRINT"PLEASE ENTER NAME OF
FACTOR";J
710 INPUT F$(J)
720 NEXT J
730 RETURN
```

NO NEED FOR CORRECTION

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

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

```
HOW MANY OUTCOMES? 5

HOW MANY FACTORS? 5

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

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

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

```
THESE ARE THE POSSIBLE OUTCOMES:

DOG
 HORSE
  SHEEP
   CROW
    HUMAN

PLEASE THINK OF ONE OF THEM

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

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

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

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

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

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

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

Eventually, it will not make any mistakes. It works in much the same way as the previous program, assigning values to the elements of the B array. However, this time it has an array called C, which insures that it will never make the same wrong guess from the same total. This acts to increasingly limit the possible outcomes that 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" that insures 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
```

MULTI-SELF-LEARN-2 LISTING

And this is the listing:

```
10 REM MULTI-SELF-LEARN-2
20 GOSUB 550:REM INITIALIZE
30 HOME
40 PRINT"THESE ARE THE POSSIBLE
OUTCOMES"
50 PRINT
60 FOR J=1 TO OTCO
70 PRINT TAB(J)A$(J)
80 NEXT J
90 PRINT
100 PRINT"PLEASE THINK OF ONE OF THEM"
110 PRINT
120 PRINT"PRESS RETURN WHEN YOU
ARE READY"
130 INPUT Z$
140 HOME
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"{<G}";F$(J)
240 INPUT Z$
250 IF Z$<>"Y" AND Z$<>"N" THEN 240
260 IF Z$="Y" THEN COUNT=COUNT+X
270 NEXT J
280 X=0
290 FOR J=1 TO OTCO
300 IF COUNT=B(J) THEN X=J
310 NEXT J
320 IF X<>0 THEN 410
330 X=INT(RND(1)*OTCO)+1
340 REM ** REJECTS ANSWERS KNOWN TO BE
WRONG **
350 FLAG=0
```

```
360 FOR J=1 TO OTCO
370 IF B(J)=0 THEN 390
380 IF X=J AND COUNT<>B(J) THEN FLAG=1
385 IFC(X)=COUNT THEN FLAG=1
387 IFD(X)=COUNT THEN FLAG=1
390 NEXT J
400 IF FLAG=1 THEN 330
410 PRINT"YOU WERE THINKING OF";A$(X)
420 PRINT
430 PRINT"ENTER 'Y' IF I'M RIGHT, 'N' IF
WRONG"
440 INPUT Z$
450 IF Z$<>"Y" AND Z$<>"N" THEN 440
460 IF Z$="Y" THEN B(X)=COUNT:GOTO 30
470 IF C(X)=0 THEN C(X)=COUNT:GOTO 30
520 D(X)=COUNT:GOTO 30
530 END
540 REM **************
550 REM INITIALIZATION
560 HOME
570 INPUT "HOW MANY OUTCOMES";OTCO
580 PRINT
590 INPUT "HOW MANY FACTORS";FACT
600 HOME
610 X=OTCO+FACT
620 DIM A$(OTCO)
630 DIM F$(FACT),B(X),C(X),D(X)
640 FOR J=1 TO OTCO
650 PRINT"PLEASE ENTER NAME OF OUTCOME ";
660 INPUT A$(J)
670 NEXT J
680 HOME
690 FOR J=1 TO FACT
700 PRINT"PLEASE ENTER NAME OF FACTOR ";J
710 INPUT F$(J)
720 NEXT J
730 RETURN
```

APPENDIXES

APPENDIX ONE

IMPROVING YOUR PROGRAMMING TECHNIQUE

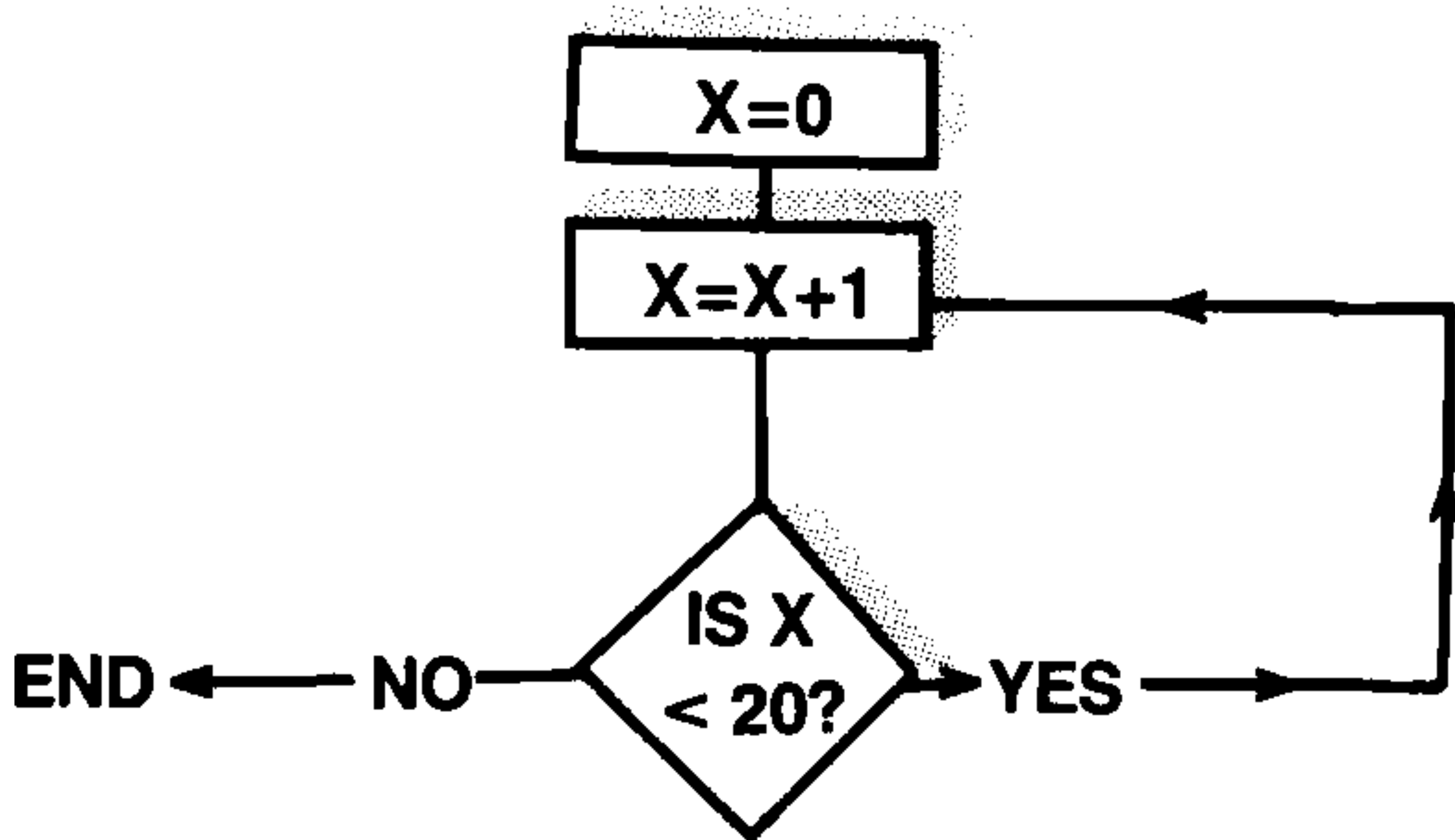
There will come a point in your development as a programmer when you'll have mastered the use of much of the language you are using, and can now concentrate on writing better programs; programs that work after relatively little debugging, that are easy for others to understand and operate, and that are written logically and elegantly. Artificial intelligence programs, especially, tend to become fiendishly complex, so anything you can do to make them easier to understand should be welcome.

Your programs will be more likely to run the first time if they are planned out carefully before you start entering code (i.e., program) into your computer.

A good way to start is to use a diagram that is often a "flow chart." A flow chart is a series of boxes and other shapes, joined by lines, that show the flow of action and decision-making within the computer while the program is running.

The shapes are not too important, and I suggest you stick to just two: a rectangle for most actions the computer must carry out, with a diamond shape each time the computer has to make a decision. The corners of the diamond can be used—as you can see in the diagram—to cater to the alternatives facing the computer.

The diagram shows the flow chart of a program that sets the variable X equal to zero, then adds one to it. The value of X is checked. If X is found to be less than 20, the program goes back to add one to X again. This continues until the value of X equals 20.



One advantage of using a flow chart is that you do not get locked, at an early stage of your work, into the peculiarities of the language, its weaknesses and strengths, that you are using. Instead, you concentrate on what you want to do. Of course, you may have to modify your plans slightly to accept limitations dictated by your programming language, whatever it may be, but you do not have to bend to these limitations (which may be, in part, imaginary) at the start of the process.

A flow chart is "universal." The same flow chart can be used as the basis of a program written on a computer furnished with a completely different language from the one in which you originally intended to write the program.

A flow chart models the flow of action and logic within a program, and is therefore very useful for picking up potential bugs at the earliest stages. You may, for example, find that one condition the program will test will never be fulfilled, possibly leading to the program being trapped in an infinite loop. Other parts of the code may be bypassed completely, because the condition that triggers entry into that part of the code will never be met.

Once you've devised a flow chart for your program, and you've run through it mentally a few times so that the most obvious bugs are removed, you should reduce the flow chart to a series of subroutine calls. Although it seems pretty silly to do this for a simple program like our "SET X EQUAL TO ZERO, ADD ONE, CHECK IF LESS THAN 20" program, this method comes into its own with more complex programs such as the ones seeking to demonstrate artificial intelligence.

PROGRAMMING IN MODULES

As you've seen in nearly all the AI programs in this book, I'm in favor of you starting your program with a series of subroutine calls, with each action of the program being looked after by a separate subroutine. Then, if the steps within a program have to be performed several times in a particular sequence, the series of subroutine calls can be cycled through, over and over again, until a particular condition is met that signals the end of the run.

You'll recognize how useful this approach to programming can be when you get to the debugging stage of your program. If there is a bug, it is likely to be within a single subroutine, so it will be relatively easy to pin down the subroutine that contains the bug, rather than having to work right through the program trying to track it down.

Working with subroutine "modules" in this way allows you to *test* sections of the program in isolation, even before the entire program is finished. I'll try to make this statement clear by showing you the first part of a typical program to play checkers (and you'll see, in fact, that SNICKERS in this book does start in a manner very similar to this). Our ideal checkers program could start like this:

```
10 REM CHECKERS
20 GOSUB 9000: REM INITIALIZE VARIABLES
30 GOSUB 8000: REM PRINT BOARD
40 GOSUB 7000: REM ACCEPT PLAYER MOVE
50 GOSUB 8000: REM PRINT BOARD
60 GOSUB 5000: REM COMPUTER MAKES MOVE
```

```
70 IF (human has not won) AND (computer
has not won) THEN 30
80 IF (computer has won) THEN PRINT "I
WIN"
90 IF (human has won) THEN PRINT "YOU
WIN"
100 END
```

You could have quite a bit of this program running, and tested (such as the initialization routine, printing the board and accepting the player's move) before you even turned your attention to how on earth you were going to get the computer to make its move.

You would then know, for example, that you would not need to waste any extra thought on whether or not an error in the board-printing routine was the cause of odd output. Having tested the board subroutine and the player move routine, you'd know that the error must be in the subroutine between lines 5000 and 6999, the subroutine in which the computer makes its move.

All you need to do is put a single PRINT statement, such as "THIS IS COMPUTER MAKING A MOVE" followed by a RETURN for incomplete subroutines, knowing that the program will accept that, and demonstrate the direction the program flow is following, even if whole sections of code have not yet been written.

In general, I'd advise you to use this system of using a "master loop" of subroutine calls within which you will "hold" the entire program.

I suggest you try and do as much writing of the program as possible *before* you turn the computer on, even though there is a great temptation to dive straight into the computer and start punching in code. You'll find that the discipline of writing it out by hand in advance will serve you in good stead and should, in the long run, produce a better program than might otherwise have been the case.

Overall, you'll probably end up spending less time on the program working in this way than you would if you began the process sitting at the computer keyboard.

It took me a while to learn this lesson. Although I had read suggestions along the lines of "work out exactly what you're going to enter *before* you start at the computer" in several books, I tended to just jump right in without much prior thought.

Although I worked out rough flow charts, and had an idea what sort of display organization I wanted, I certainly did not write much

program out on paper before starting at the computer. Then, I once found myself stuck for a two-week period without a computer and ideas for several programs just itching to be written. I had to write them out in an exercise book.

The relative ease with which the programs were debugged when they were eventually entered into the computer, and the complexity of the programs I wrote in this way (including my first chess program) convinced me that this was the way I would work from then on. It is amazing how much cleaner a program can be if all the rough working out is done on paper, rather than on the computer screen.

When working on AI programs, I tend to write the major "call subroutines loop" first of all, but without line numbers, so the program contains lines like GOSUB PRINT BOARD and GOSUB INITIALIZE. Next, I write each module (or subroutine) on a separate sheet of paper. Then, when the major subroutines have been written I shuffle them into an order that seems most logical.

All this, of course, occurs before any line numbers are written in. The subroutine modules are put in an order that insures that the program structure is as logical as possible. I use arrows to indicate the destination of GOTO's within a module, and *names* for subroutines, as suggestions in the major loop. Later on, when the program has started to assume a firm shape, the lines are numbered (I always work in tens, starting at line 10), and the relevant GOTO and GOSUB destinations are added.

All programs have an "end condition" at which point computation stops. It is worth putting a test for this end condition as part of the GOTO that sends the program back to start cycling through the major subroutine loop. This insures that the cycle will continue until a particular condition is met, at which point the program "falls through" that GOTO and continues onto the lines that signal the end of the program.

EXPLICIT INPUT PROMPTS AND PRINT STATEMENTS

It is very useful, when writing a program, to keep in mind how the program will appear to a stranger when it is run by him or her for the first time. If there is an *input prompt* required, it is far more useful if

the program prints up something like "HOW MANY HOURS HAS THE EMPLOYEE WORKED THIS WEEK?" instead of just "INSERT HOURS?" or the almost useless lone question mark.

The same suggestion applies to print output. It is far better that your program be written so that it prints out THE NUMBER OF HOURS WORKED ON FULL PAY THIS WEEK IS 27, rather than HOURS, FULL: 27 or an unsupported 27. Of course, providing explicit input prompts and output PRINT statements consumes memory as well as typing time when entering the program, but the contribution they make to the final program means the trouble involved is well worth it.

REM STATEMENTS

While exact PRINT statements and input prompts will help a person running a program make sense of it, REM statements can help make the program clear to those who are examining the listing for the first time. REM statements (which are, as you know, ignored by the computer when a program is run) should be used to help illuminate the flow of logic within the program, and what is happening in certain places within it. This is especially important in parts of the program where decisions are made, or calculations are carried out.

Not only can REM statements be used to explain different parts of the code, but they can also be used to provide visual "breaks," so that the various blocks of code that carry out certain tasks are visually separated from the rest of the program. A blank REM statement (the word REM standing alone in a program line) can be used for this. A row of asterisks is an effective alternative.

VARIABLES

It is worth considering the use of explicit names for variables, using either the word in full (such as HOURS as a variable name in a payroll program for hours worked) or an abbreviated version of this

(such as HR), which has a fairly obvious meaning. You'll discover that this makes it easy, when working on a program, to keep track of your variables. This will help you with the initial debugging and later on as well if you need to improve or extend the original program.

Explicit variable names also help make your code more "transparent" so other programmers can work out what the various parts of the program are intended to do. You'll also find it a great help to yourself when you return to the program at a later date. It is surprising how code, which seemed incredibly clear in terms of its purpose when you entered it, becomes exceedingly dense when you return to it after a long break.

CHECKING INPUT

Any input entered by the user should be checked by the program before it is accepted to insure that incorrect data does not cause the program to crash at some future point. Whether you want string, or numeric input, it is often wise to allow for string input, which is first checked to see whether the entered material is acceptable and then, if necessary, the string can be changed into a number.

For example, if the user needs to enter a number between one and nine, a string can be accepted and then checked to insure that it is not less than "1" or more than "9" before being changed into a number with a command like VAL.

As well as insuring that the program will reject invalid input, you should check the program to see that all the inputs that it does accept produce sensible answers when processed later on in the program. For example, make sure that your program does not accept zero as a possible number if the computer must later divide by this number.

Similarly, if numbers are to be processed by a function, and then the result of this processing used for division, you must check that an apparently valid input does not turn into zero as a result of evaluation by the function.

If the information entered by the user is rejected, and a new input is requested, the program should ideally point out why the original input was not suitable, or spell out again exactly what is

required (such as "ENTER A NUMBER BETWEEN 1 AND 4"). You risk making users angry if input that appears valid is continually rejected without apparent reason.

DOCUMENTATION

The written material that accompanies a program is often called documentation. It is useful for a program to be supported by some documentation, however sketchy.

The written information should explain, of course, what the program does, then go on to outline the flow of action within the program. The documentation should alert the user as to the kind of actions that will be required from him or her when running the program, and give an indication of the kinds of user input and reaction that will be accepted.

The format of the final output should also be discussed. A list of variable names can be included.

If there are ways in which the program can be developed, extended or improved, suggestions along these lines can be added to the documentation. Written references to any material that will help in understanding the algorithms used, or for giving suggested areas for program development, should also be included.

In many ways, it is reasonable to assume that the job of programming is not finished once the program is done. Without documentation, the job is only three-quarters complete. Documentation finishes the task, adding a professional stamp to your work that allows the program you've written to be used most effectively.

Possibly the only time extensive documentation is not really required is when the program is "menu-driven." A program that uses REM statements extensively may not need very much in the way of documentation, especially if you can include a variable list, as a series of REM statements, at the end of the program.

Generally, however, you'll find it better to document a program externally, rather than rely on REM statements, or the various menu choices, to do the job for you. It is worth trying to write your program documentation so that it would make sense to someone who has not seen the program running.

This person should be able to get a very good idea, just from reading the documentation, of what the program does and how it does it; how it interacts with the user both in terms of accepting information and in presenting the results of its computations to the users; and how the program is organized as a whole.

When writing an AI program, it is probably of use to include in your documentation some information on the nature of AI that the program is intended to demonstrate. Also, you could perhaps give references to articles or sections of books that discuss the field in which your particular AI program would best be situated. These things help a user to appreciate your program for the brilliant work it is.

Documentation for a major program should start with an introduction that quickly explains what is going on, and tells how to use the program. The later parts of the documentation can then discuss the program in greater detail. It is not good practice to force the user to wade through a vast amount of information in order to dig out the vital facts he or she needs to get the program running.

APPENDIX TWO

SUGGESTIONS FOR FURTHER READING

The literature on artificial intelligence is already quite large, and growing larger by the day. The majority of published works I have examined in this field are worthwhile. So far, there is little dross.

In this appendix, I am not attempting anything like a comprehensive overview of the available literature. Rather, I will discuss books that I have found of particular interest and value. I have discussed each book in some detail, so you can get an idea of what is in them, and whether or not you are likely to wish to purchase them for your own use. AI books, perhaps more than those covering any other computer field, seem to be destined to have high cover prices. Prices approaching $100.00, for single volume hardbacks, are not uncommon. Even paperbacks, in this field, seem to be "generously" priced. So it is pretty important to know what you're buying before you order the book.

I have included a few books that only touch on AI marginally (such as Levy's chess and games books, and the one edited by Asimov) because the topics they cover are of interest to anyone working in the AI field. Also, they suggest areas that you may wish to explore further. I would be grateful for information on any other publications that you have read and have found of real value. I can then assess these works to include in future editions of this book.

Are Computers Alive?, Geoff Simons (Brighton, Sussex: The Harvester Press Ltd., 1983).

This is an easy-to-read overview of the present state of AI, with an emphasis on robot development. This book covers a lot of ground (as you'll see from the contents) but it manages to do so without sacrificing the depth necessary to place things in their historical context, and without treating them so lightly that the study becomes almost worthless.

The contents include:

Are computers alive? (And this section includes a nice spooky piece called "Machines as emerging life.")
The robot background
The behavior of machines
The anatomy of robots
The psychology of computers
Computer liberation (Now there's a topic that few have tackled since Asimov had a robot declared pope.)
The human response
The future

If you want a broad overview of the field, written in a clear, simple way, this is the book to get.

Artificial Intelligence Patrick Henry Winston (Reading, Massachusetts: Addison-Wesley Publishing Co., 1984).

Winston is a professor of Computer Science, and also Director of the Artificial Intelligence Laboratory at MIT. This is the second edition of a work that, soon after its first edition was published in 1977, became recognized as a standard work in the field.

Winston points out that, in the seven years since the book was originally written, much in the AI field has changed. The principal change, he says, is from AI "illustrations" (programs that show what could be done, except the actual working programs only perform within artificial, narrow domains) to "demonstrations" (practical, working AI systems), delivering on some of the promises of the past.

Among much other material, the book gives a large number of illustrations of the practical constraints that must be recognized when working in AI. Winston points out that, to understand AI and to work in the field, there are some basic ideas that need to be understood. He

says these ideas are matching, goal reduction, constraint exploitation, search, control, problem solving, and logic. The first seven chapters of the book address these areas.

The main topics covered in this book are:

1 — The Intelligent Computer
- The field and the book
- What computers can do
- Criteria for success

2 — Description matching and goal reduction
- The key role of representation
- Analogy intelligence tests
- Moving blocks
- Problem solving and understanding knowledge

3 — Exploiting natural restraints
- Propagating symbolic constraints
- Propagating numeric constraints
- Dependency-directed backtracking
- Methodology

4 — Exploring alternatives
- Finding paths
- Finding the best path
- Dealing with adversaries

5 — Control metaphors
- Control choices
- Means-end analysis and GPS

6 — Problem-solving paradigms
- Generate and test
- Rule-based systems for synthesis
- Rule-based systems for analysis
- Rule-based systems for modelling human thinking

7 — Logic and theorem proving
- Rules of inference
- Resolution proofs
- Planning operator sequences
- Proof by constant propagation

8 — Representing common-sense knowledge
  Representation
  Inheritance, demons, defaults, and perspectives
  Frames and news stories
  Expansion into primitive acts
  Abstraction to summary units

9 — Language understanding
  From sentences to world models
  Parsing sentences
  Forming thematic-role frames
  Interpreting simple questions and commands
  Intellectual partnerships

10 — Image understanding
  From images to object models
  Computing edge distance
  Computing surface direction
  Interpreting simple binary images
  Robotics

11 — Learning class descriptions from samples
  Induction heuristics
  Induction procedures
  Identification

12 — Learning rules from experience
  Learning grammar rules from sample sentences
  Learning rulelike principles
  Matching is behind analogy
  Learning form from functional definitions
  Improving rules using learnable censors

As can be seen from this formidable list, the book covers every major section of the AI world, and as such, forms the basis of a very solid AI library.

Artificial Intelligence—An Introductory Course, ed. A. Bundy (Edinburgh: Edinburgh University Press, 1980).

This book is the collected notes of the course "Artificial Intelligence 2." The emphasis in the course is on showing how AI programs are built, rather than subjecting students to a broad overview

of the field. The material on manipulations in a blocks world and on tree-searching is particularly interesting.

The book is divided into six sections: Representation of Knowledge, Natural Language, Question Answering and Inference, Visual Perception, Learning, and Programming. A number of working programs (written in LOGO) are given in this fairly dry, but nevertheless extremely valuable, book. LISP translations of most of these are included.

Artificial Intelligence—An MIT Perspective, ed. P. H. Winston and R. H. Brown (Cambridge, Massachusetts & London: The MIT Press, 1980).

This two-volume work is riddled with heavy, heavy theory, with lots of formulae, and logical expressions riddled with arcane symbols. If you can handle them (or look at the text that lies around them), you'll find much of value.

The first volume contains these topic sections:

Expert problem solving
Natural language understanding and intelligent computer coaches
Representation and learning

In the second volume, the main sections are these:

Understanding vision
Manipulation and productivity technology
Computer design and symbol manipulation

Artificial Vision for Robots, ed. I. Aleksander (London: Kogan Page Ltd., 1983).

This is a fascinating overview of the field, edited by the chairman of the British Cybernetics Society. Aleksander is Professor of Electronics at Brunel University, but brings a remarkably nonacademic (i.e., clear and down-to-earth) style to his introduction, and his two other contributions to the book.

Topics covered in this work are:

Software or hardware for robot vision?
Comparison of 5 methods for recognition of industrial parts
Syntactic techniques in scene analysis

Recognition of overlapping workpieces
Simple assembly under visual control
Visually interactive gripping of engineering parts from random orientation
An interface circuit for a linear photodiode array camera
Network of memory elements: A processor for industrial automation
Computer vision systems for industry: Comparisons
Memory networks for practical vision systems: Design calculations
Emergent Intelligence from adaptive processing systems

The final chapter comes from Aleksander himself, and he starts with a quick overview of the present situation of AI research, then leads into his discussion of "emergent properties." This phrase comes from a description of the intelligent behavior of an animal being an emergent property of the neural tissue that makes up its brain. He says it can be argued that the intelligent behavior of a computer is not an emergent property of the machine (any more than a painting is an emergent property of the canvas), but that to see AI as an emergent property of the rules upon which a particular program is written, is perhaps an avenue worth following. All in all, a very interesting book, which shows clearly how the various fields of AI overlap.

Automatic Natural Language Parsing, ed. K. Sparck Jones and Y. Wilks (Chichester, West Sussex: Ellis Horwood Ltd., and New York: Halsted Press, 1983).

In the same series as *Machine Intelligence—9* and *Intelligent Systems*, this book looks at the current views on the problems and techniques of automatic natural language parsing. The book points out that parsing is the key element of natural language processing as a whole, and is vital for such areas as linguistics-oriented research and with language subsystems as components of such things as expert systems.

Build Your Own Expert System, Chris Naylor (Cheshire: Sigma Technical Press, 1983).

This book has rapidly earned a reputation as the best "do it yourself" guide to expert systems on the market. At first I was dismayed to find that Chris was trying to be *funny* in the book. Then,

I discovered he could write some pretty funny stuff, and the excruciating humor certainly helps the rather heavy medicine of the text go down.

The early material is reasonably easy to understand, and the program listings illuminate the text quite significantly and, more to the point, actually provide you with some expert systems programs that have real merit. The going soon gets heavier, and requires some careful reading, when getting involved in the intricacies of Bayes' Theorem and the like, but the explanations are all there, so dogged reading will illuminate even the more complex material.

To round out the book, Naylor provides a large medical knowledge base (with data on some 100 different disease types and their diagnoses). As he points out, the program won't allow you to practice medicine but will enable you to develop some rather elaborate scenarios for your hypochondria. Even if you decide you could do without your computer asking you questions like DO YOU HAVE STRONG FEELINGS OF NAUSEA or ARE YOU (OR THE PATIENT) DELIRIOUS—TALKING INCOHERENTLY WITH POOR MUSCULAR COORDINATION the database shows that Naylor is talking about expert systems in the real world.

If you want to study expert systems in any depth, this is the volume to begin with.

Chess and Computers, David Levy (Potomac, Maryland: Computer Science Press, Inc., 1976).

Mr. Levy is an International Chess Master and a prolific chess writer. He has achieved world-wide fame as a result of the "Levy Wager," in which he bet in 1968 that no chess program produced in the following decade would be able to beat him. He won in 1978. He started his computing life at Glasgow University, where he lectured in Algol programming and AI in the early 1970s. Levy is one of the world's top authorities on computer chess, and is chairman of a London-based company that specializes in programming intelligent games.

The chapters include:

Chess Machines
How Computers Play Chess
The Early History of Computer Chess

The Modern Era of Computer Chess
Current Chess Tournaments
Current Research and Future Prospects

You will see from this book that Levy's reputation is well deserved. If you have any interest whatsoever in man's attempts to produce machines that play chess, this is the book to get. The book contains a solid history, and there are a number of games, and part games, to show programs in action.

Computer Gamesmanship, David Levy (London: Century Publishing, 1983).

Subtitled *The complete guide to creating and structuring games programs*, this book comes from David Levy, whose pedigree is outlined in the comments on *Chess and Computers*. It contains a particularly clear exposition on searches, the alpha-beta algorithm, and on ways of speeding up searches. Particular games covered in this work are:

Nim
Card Games (inc. Rummy and Poker)
Samuel's Checkers Program
Chess
Backgammon
Othello
Go-Moku (and Benju)
Shogi
Dominoes

As you can see, this is a pretty formidable list. If you're interested in studying the relationship between AI and game-playing, this book is a must.

Computer Game-Playing—Theory and Practice, ed. M. A. Bramer (Chicester, West Sussex: Ellis Horwood Ltd., 1983).

This is a far heavier treatment of the field than is the Levy book, but it is still eminently readable. The 300 pages contain a wealth of specific ideas that can be applied to the creation of game-playing programs. The book also contains a number of suggested avenues for further research. In all, this is a work you will find very valuable for

reference, even if it is not (as could be the Levy volume) one you would read through from start to finish in one sitting.

Topics covered include "Chess knowledge representation and fast tree searching," "Machine learning and chess," "A strategic approach to the game of Go," "Reversi: An experiment in game-playing programs," "A competitive Scrabble program" and "The three-cushion billiard game."

Computer Power and Human Reason, Joseph Weizenbaum (San Francisco: W. H. Freeman and Co., 1976).

This is an important book, even if the view it puts forward—that in AI research we are leaping into areas of extreme danger, with no awareness of that which we are courting—appears to rest more on some inner fixation of Weizenbaum's, than on many cogent arguments. I've mentioned this book in the material leading up to the DOCTOR program, because it was Weizenbaum who wrote the first ELIZA program, from which the DOCTOR stems. In fact, his book was written as a direct result of Weizenbaum's "shock" at the way people reacted to ELIZA. (I expand on this statement in the material related to DOCTOR.)

Weizenbaum believes there are areas where machines should not intrude, even if they can. Arguments against this thesis, that surely if a machine can help in certain circumstances its contribution should at least be investigated, appear to have fallen on deaf ears, because Weizenbaum's tone—in the continuing debate on the issues raised by the book—has reportedly become more strident over the years. McCorduck (in *Machines Who Think*, p. 318) reports a colleague claiming that Weizenbaum has taken the evil of AI as his hobby-horse because he has failed to produce anything of worth in the field since ELIZA.

Be that as it may, the book certainly deserves to be read, even if you end up denying most of its theses. There is value in examining informed criticism, even if you eventually feel the critic is banging an empty drum. And, you may well not end up with that impression. The fact that the book is still in print almost a decade after its first appearance (a rare thing in a field that develops as rapidly and dramatically as AI) shows its ideas are still worthy of consideration and debate.

The Fifth Generation—Artificial Intelligence and Japan's Computer Challenge to the World, Edward A. Feigenbaum and Pamela

McCorduck (Reading, Massachusetts: Addison-Wesley Publishing Co., 1983).

This is a peculiarly American book. A book written with a definite purpose: to generate a specific political reaction. As such, it was written too hastily, and reads more like an extended article for *Time* magazine than a book. (In terms of quality of writing, it is considerably below the eloquence of Ms. McCorduck's *Machines Who Think*, which transcends its ghastly title to transmit, quite poetically, a particularly rich view of the history of AI. For more on this book, see the reference further on in this bibliography.)

Quibbles about its style apart, the message of the book *is* important. The message is in three parts, an observation, a warning and a call to arms. Japan has realized that the future wealth of the world lies in information management, information manipulation, information control. Japan has determined it will be the market leader in this most modern of industries. And, using the peculiar power of "Japan Inc.," it has brought together a group of bright young things ("nobody over 35" was one of the rules) to brain-storm the future, and bring genuine computer intelligence into being within the decade (which began in 1981, as you'll see from the following book).

That's the observation. The warning is that America is in danger of losing, perhaps forever, its lead in information technology. The call to arms is for some sort of national, coordinated response (as Britain and some European countries have already made) to the challenge. Whether or not you perceive the danger as being as immediate as McCorduck and Feigenbaum claim, and whether or not you agree with the remedy (even if you acknowledge the diagnosis), this is a book you *must* read. Without the perspective accorded by this book, you'll be thinking blind about AI.

Fifth Generation Computer Systems, ed. T. Moto-oka (New York: North-Holland Publishing Co., 1982).

This work is the proceedings of the international conference on fifth generation computer systems, held in Japan in 1981, edited by the chairman of the program committee. The fifth generation is a phrase you will not be able to escape in the coming years (see the comments on the Feigenbaum/McCorduck book for the reason), and this work is the record of where it began. The papers in this collection vary from overviews (including "What is required of the fifth generation computer—Social needs and impact") to detailed presen-

tations on very specialized subjects (such as "VLSI and system architecture—The development of System 5G").

This is not a volume for light skimming, and you should only consider investing in it if you are particularly interested in the implications and promise of the Japanese project, and are willing to do some careful reading. If you decide the work is for you, there is much of value in it.

The contents are as follows (with the principal sections shown by use of upper case):

KEYNOTE SPEECH
Challenge for knowledge information processing systems

OVERVIEW REPORT
What is required of the fifth generation computer—social needs and impact
Aiming for knowledge information processing systems
Fifth generation computer architecture

KNOWLEDGE INFORMATION PROCESSING RESEARCH PLAN
Problem-solving and inference mechanisms
Knowledge base mechanisms
Intelligent man-machine interface
Logic programming and a dedicated high-performance personal computer

ARCHITECTURE RESEARCH PLAN
New architecture for inference mechanisms
New architecture for knowledge base mechanisms
VLSI and system architecture—the development of System 5G
The preliminary research of data flow machine and database machine as the basic architecture of fifth generation computer systems

INVITED LECTURES—KNOWLEDGE INFORMATION PROCESSING
Innovation and symbol manipulation in the fifth generation computer systems (E. A. Feigenbaum)
Logical program synthesis (W. Bibel)
The scope of symbolic computation (G. Kahn)

INVITED LECTURES—ARCHITECTURE
A cognitive architecture for computer vision (B. H. McCormick et al.)
Fifth generation computer architecture analysis (P. C. Treleaven)
Algorithms, architecture and technology (J. Allen)

If you want to get a grip on the fifth generation system from the source, this publication is the place to get it.

Giant Book of Computer Games, Tim Hartnell (New York: Ballantine Books, 1984; and London: Fontana Paperbacks, 1983).

There's no harm in getting a plug in for one's own work. I've included this book because it contains a large number of program listings that demonstrate certain aspects of AI, including a program that uses "fuzzy logic" to search for the solution to a Mastermindlike code set by the user. The Reversi/Othello, Gomoku, and Shogun (my own variation on Hasami Shogi) programs all contain code that may be of interest. There is even a chess game, which plays appalling chess, but does demonstrate some extremely rough-and-ready evaluation functions in action. At the very least, you may find this book a good source of ideas that you can use as the basis of your own AI programs.

Intelligent Systems—The Unprecedented Opportunity, ed. J. E. Hayes and D. Michie (Chichester, West Sussex: Ellis Horwood Ltd., and New York: Halsted Press, 1983).

If Naylor's book *Build Your Own Expert System* (q.v.) is the vital guide to D.I.Y. expert systems, this volume (edited by the ubiquitous Donald Michie who pops up all through the literature—and no doubt the world—of AI) is the essential "heavy backup" book. This is where you'll find some of the solid theory behind Naylor's humor, and this is where, more importantly, you'll find the flavor of the future.

Part of the same series as *Machine Intelligence—9* (q.v.), this book grew out of a seminar sponsored by the British Computer Society, under the independent chairmanship of Michie, with the title *The Next Ten Years*. Despite its genesis in conference papers, which suggests it may well be dry and incomprehensible to ordinary mortals such as me, I found the book clear and simple. Above all it was fascinating. It is extraordinary, perhaps, that so much of the literature

on AI is so exciting. I guess that if you're interested in AI, you could hardly find the literature lacking in excitement. In this volume, as in so many others, you gain a sense of interacting with a very near, very probable "science fiction" future.

And because it is probable, its fascination outweighs a hundred imaginary environments.

The contents of *Intelligent Systems* are as follows (with major subdivisions in upper case):

KNOWLEDGE-BASED REASONING SYSTEMS
The calculus of reasoning (J. C. Shepherdson)
Logical reasoning in machines (J. A. Robinson)
Knowledge engineering: The applied side (E. A. Feigenbaum)
A prototype knowledge refinery (D. Michie)

ENGINEERING TOMORROW'S WORLD
Seeing machines (M. J. B. Duff)
Walking machines (R. McGhee)
Computers in medical biotechnology (W. F. Bodmer FRS)
Technology and the universities (D. B. Thomas)
Robots of the future (E. C. Joseph)

THE POPULATION AND OTHER EXPLOSIONS
Make room! Make room! (H. Harrison)

IMPACT ON LIFE AND WORK
The emerging challenge (I. Lloyd MP)
Training for multi-career lives (P. Virgo)
The future of work (Sir Ieuan Maddock FRS)

Machine Intelligence—9, ed. J. E. Hayes, D. Michie and L. I. Mikulich (Chichester, West Sussex: Ellis Horwood Ltd., 1979).

This book is part of the MACHINE INTELLIGENCE series (volumes one to seven published by the Edinburgh University Press in the UK and by Halsted Press in the U.S.; with volumes eight and nine from Ellis Horwood). Its editor in chief is Donald Michie, who has played an important role in AI since the beginning.

The series is extremely heavy, and is not recommended reading unless you want to dive deep into the arcane sea of logic symbols. If your background is such that you could enjoy this sort of thing, you'll find much of interest here.

The book is based on papers presented at an East-West forum held near Leningrad in 1977, when leading experts in Western Europe and North America joined forces with their Soviet counterparts at the 9th International Machine Intelligence Workshop.

The section headings should give you an idea of the range of ideas canvassed at the forum:

Abstract models for computation
Representations for abstract reasoning
Representations for real-world reasoning
Search and problem solving
Inductive processes
Perception and world models
Robot and control systems
Machine analysis of chess
Knowledge engineering
Natural language

Machines Who Think, Pamela McCorduck (San Francisco: W. H. Freeman and Co., 1979).

Forget the ghastly title. This book is a real gem. Subtitled *A Personal Inquiry into the History and Prospects of Artificial Intelligence,* McCorduck brings the ability to write very, very well to bear on the people who make up the history of artificial intelligence.

She starts the book by recounting some historical traditions of man fearing his manlike creations, and argues convincingly that these attitudes color the way people think about, and react to, AI artifacts today.

This insight is extremely valuable in itself, and serves to demonstrate how deeply (and widely) McCorduck has thought about the subject. What is more, she actually spoke to all the giants in the field, including Weizenbaum ("Mr. ELIZA") and Simon who, with Newell, built the world's first true thinking machine, the Logic Theorist. As one of its early triumphs, this machine discovered a simpler proof to a theorem in *Principia Mathematica* than Russell and Whitehead had found; Lord Russell was apparently delighted and so was Rosen who, with a team at Stanford Research Institute, built a self-propelling robot called Shakey that got a lot of publicity in the 1960s.

This means the book is a prime-source historical document,

with asides and anecdotes that bring the human face of AI research (as well as the milestone triumphs and failures of AI) into the spotlight.

Practical Experience of Machine Translation, ed. Veronica Lawson (New York: North-Holland Publishing Co., 1982).

This work is the proceedings of a conference held in London in November 1981 that brought together a number of senior works in the field of MT. It is a good report of the state of the art at that time, and comes with several pointers as to the directions in which MT is heading.

The titles of papers presented include "Machine translation and people" (Veronica Lawson); "Coping with machine translation" (J. Richard Ruffino); "Economic aspects of machine translation" (Georges van Slype); "Experience in English-French post-editing" (Bernard Lavorel); "The pivotal role of the various dictionaries in an MT system" (Francis E. Knowles); and "The limits of innovation in machine translation" (Margaret Masterman).

If you wish to investigate this particularly practical application of AI, this work is a good place to begin.

Those Amazing Electronic Thinking Machines! ed. Isaac Asimov, Martin Greenberg and Charles E. Waugh (New York: Franklin Watts, 1983).

This collection, an anthology of robot and computer stories, is included because it is fascinating to see some of the possibilities for the art that have been envisioned by writers, given the insights you've gained from working in AI.

When you've tired of wading through the logic symbols of *Machine Intelligence—9*, you can turn to this anthology for refreshment.

APPENDIX THREE

ARTIFICIAL INTELLIGENCE AND COMPUTER TERMS

Accumulator—Part of the computer's logic unit that stores the intermediate results of computations.

Address—A number that refers to a location, generally in the computer's memory, where information is stored.

Algorithm—The sequence of steps used to solve a problem.

Alphanumeric—Gerterally used to describe a keyboard, and signifying that the keyboard has alphabetical and numerical keys. A numeric keypad, by contrast, only has keys for digits one to nine, with some additional keys for arithmetic operations, much like a calculator.

APL—This stands for Automatic Programming Language, a language developed by Iverson in the early 1960s, which supports a large set of operators and data structures. It uses a con-standard set of characters.

Application software—These are programs that are tailored for a specific task, such as word processing, or handling mailing lists.

Artificial Intelligence—The section of computer science that concentrates on eliciting machine behavior that, if it came from a human being, would be called intelligent. One of the aims of AI (as artificial intelligence is often written) is to make computers

more useful. AI research may also turn out to help us understand our own thinking processes.

ASCII—Stands for American Standard Code for Information Interchange. This is an almost universal code for letters, numbers, and symbols, having a number between 0 and 255 assigned to each of these, such as 65 for the letter A.

Assembler—This is a program that converts another program written in an assembly language (which is a computer program in which a single instruction, such as ADD, converts into a single instruction for the computer) into the language the computer uses directly.

BASIC—Stands for Beginner's All-purpose Symbolic Instruction Code, the most common language used on microcomputers. It is easy to learn, with many of its statements being very close to English.

Batch—A group of transactions that are to be processed by a computer in one lot, without interruption by an operator.

Baud—A measure of the speed of transfer of data. It generally stands for the number of bits (discrete units of information) per second.

Benchmark—A test that is used to measure some aspect of the performance of a computer, which can be compared to the result of running a similar test on a different computer.

Binary—A system of counting in which there are only two symbols, 0 and 1 (as opposed to the ordinary decimal system, in which there are ten symbols, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). Your computer "thinks" in binary.

Boolean algebra—The algebra of decision-making and logic, developed by English mathematician George Boole, and at the heart of your computer's ability to make decisions. Much of a computer's strength in demonstrating "intelligent" behavior lies in its use of Boolean algebra (see the section in this book on "Learning and Reasoning").

Bootstrap—A program, run into the computer when it is first turned on, which puts the computer into the state where it can accept and understand other programs.

Buffer—A storage mechanism that holds input from a device such as a keyboard, then releases it at a rate that the computer dictates.

Bug—An error in a program.

Bus—A group of electrical connections used to link a computer with an ancillary device, or another computer.

Byte—The smallest group of bits that makes up a computer word. Generally a computer is described as being "8 bit" or "16 bit," meaning the word consists of a combination of eight or sixteen zeros or ones.

Central Processing Unit (CPU)—The heart of the computer, where arithmetic, logic, and control functions are carried out.

Character code—The number in ASCII (see ASCII) that refers to a particular symbol, such as 32 for a space and 65 for the letter A.

COBOL—Stands for Common Business Orientated Language, a standard programming language, close to English, and which is used primarily for business.

Compiler—A program that translates a program written in a high level (humanlike) language into a machine language that the computer is able to understand directly.

Concatenate—To add (adding two strings together is known as "concatenation").

CP/M—These initials stand for Control Program/Microcomputer, an almost universal disk operating system developed and marketed by Digital Research, Pacific Grove, California.

Data—A general term for information processed by a computer.

Database—A collection of data, organized to permit rapid access by computer. A relational database is one in which the interconnections between various elements within the database are stored explicitly to aid manipulation of, and access to, the elements within the database.

Debug—To remove bugs (errors) from a program.

Disk—A magnetic storage medium (further described as a "hard disk," "floppy disk" or even "floppy") used to store computer information and programs. The disks resemble, to a limited extent, 45 rpm sound records, and are generally eight, five and a quarter, or three and a half inches in diameter. Smaller "microdisks" are also available for some systems.

Documentation—The written instructions and explanations that accompany a program.

DOS—Stands for Disk Operating System (and generally pronounced "doss"), the versatile program that allows a computer to control a disk system.

Dot-matrix printer—A printer that forms the letters and symbols by a collection of dots, usually on an eight-by-eight or seven-by-five grid.

Double-density—Adjective used to describe disks when recorded using a special technique that, as the name suggests, doubles the amount of storage the disk can provide.

Dynamic memory—Computer memory that requires constant recharging to retain its contents.

EPROM—Stands for Erasable Programmable Read Only Memory, a device that contains computer information in a semipermanent form, demanding sustained exposure to ultraviolet light to erase its contents.

Error messages—Information from the computer to the user, sometimes consisting only of numbers or a few letters, but generally of a phrase (such as "Out of memory") that points out a programming or operational error that has caused the computer to halt program execution.

Expert system—A computer program that, drawing on the encoded expertise of human experts, performs a specialized task as well as (or, in some cases, better than) the human expert in that field would. They often call on extensive databases of knowledge, and are sometimes called knowledge-based systems.

Field—A collection of characters that form a distinct group, such as an identifying code, a name, or a date. A field is generally part of a record.

File—A group of related records that are processed together, such as an inventory file or a student file.

Firmware—The solid components of a computer system are often called the "hardware." The programs, in machine-readable form on disk or cassette, are called the "software," and programs that are hard-wired into a circuit are called "firmware." Firmware can be altered in some circumstances to a limited extent by software.

Flag—This is an indicator within a program, with the "state of the flag" (i.e., the value it holds) giving information regarding a particular condition.

Floppy disk—See disk.

Flow chart—This is a written layout of program structure and flow, using various shapes, such as a rectangle with sloping sides for a computer action and a diamond for a computer decision. A flow chart is generally written before any lines of program are entered into the computer.

FORTRAN—A high-level computer language, generally used for scientific work (from FORmula TRANslation).

Gate—A computer "component" that makes decisions, allowing the circuit to flow in one direction or another, depending on the conditions to be satisfied (see the section of this book on "Learning and Reasoning").

GIGO—Acronym for "Garbage In Garbage Out," suggesting that if rubbish or wrong data is fed into a computer, the result of its processing of such data (the output) must also be rubbish.

Global—A set of conditions that effects the entire program is called "global," as opposed to "local."

Graphics—A term for any output of computer that is not alphanumeric or symbolic.

Hardware—The solid parts of the computer (see "software" and "firmware").

Heuristic—A path toward a goal that has been worked out by experience, rather than by calculation; a path of this type is not guaranteed to produce a certain result (in contrast to an algorithm that is a technique or procedure that, when applied, produces a desired result). Chess programs play, in large measure, heuristically.

Hexadecimal—A counting system often used by machine code programmers because it is closely related to the number storage methods used by computers, based on the number 16 (as opposed to our "ordinary" number system, which is based on 10).

Hex pad—A keyboard, somewhat like a calculator, which is used for direct entry of hexadecimal numbers.

High-level languages—Programming languages that are close to English. Low-level languages are closer to those that the computer understands. Because high-level languages have to be compiled into a form that the computer can understand before they are processed, high-level languages run more slowly than do their low-level counterparts.

Human interface—The "outward and visible sign" of an expert system, with which the human user of the system interacts; these often work with natural language.

Inference system—The mechanism by which a computer program reaches conclusions (see the "Learning and Reasoning" section of this book); some systems make hard and fast YES/NO decisions; others operate in the world of "fuzzy logic," where shades of gray (degrees of uncertainty) are permitted.

Input—Any information that is fed into a program during execution.

I/O—Stands for Input/Output port, a device the computer uses to communicate with the outside world.

Instruction—An element of programming code that tells the computer to carry out a specific task. An instruction in assembler language, for example, could be ADD, which (as you've guessed) tells the computer to carry out an addition.

Interpreter—Converts the high-level ("human-understandable") program into a form that the computer can understand.

Joystick—An analogue device that feeds signals into a computer that is related to the position that the joystick is occupying; generally used in games programs.

Kilobyte—The unit of language measurement; one kilobyte (generally abbreviated as K) equals 1024 bits. (A bit is the smallest discrete unit.)

Knowledge base—The accumulated knowledge upon which an expert system makes its judgments.

Knowledge engineering—The process by which human expertise is transferred to an expert system.

Knowledge Information Processing Systems—These are the fifth generation computer systems that are under development in Japan. (See references to books called *The Fifth Generation* and *Fifth Generation Computer Systems* in the bibliography of this book.)

Low-level language—A language that is close to that used within the computer (see high-level language).

Machine language—The step below a low-level language; the language that the computer understands directly.

Memory—The device or devices used by a computer to hold information and programs being currently processed, and for the instruction set fixed within a computer, which tells it how to carry out the demands of the program. There are basically two types of memory. (See RAM and ROM.)

Microprocessor—The "chip" that lies at the heart of your computer. This does the "thinking."

Modem—Stands for MOdulator/DEModulator, and is a device that allows one computer to communicate with another via the telephone.

Monitor—(a) A dedicated television screen for use as a computer display unit, it contains no tuning apparatus; (b) the information within a computer that enables it to understand and execute program instructions.

Motherboard—A unit, generally external, that has slots to allow additional "boards" (circuits) to be plugged into the computer to provide facilities (such as high-resolution graphics or "robot control") that are not provided with the standard machine.

Mouse—A control unit, slightly smaller than a box of cigarettes, that is rolled over the desk, moving an on-screen cursor in parallel to select options and make decisions within a program. "Mouses" work either by sensing the action of their wheels or by reading a grid pattern on the surface upon which they are moved.

Network—A group of computers working in tandem.

Numeric pad—A device primarily for entering numeric information into a computer, similar to a calculator.

Octal—A numbering system based on eight (using the digits 0, 1, 2, 3, 4, 5, 6 and 7).

On-line—A device that is under the direct control of the computer.

Operating system—This is the "big boss" program or series of programs within the computer that controls the computer's operation, doing such things as calling up routines when they are needed and assigning priorities.

Output—Any data produced by the computer while it is processing, whether this data is displayed on the screen, or dumped to the printer, or is used internally.

Pascal—A high-level language, developed in the late 1960s by Niklaus Wirth, that encourages disciplined, structured programming.

PILOT—A high-level language, generally used to develop computer programs for education.

Port—An output or input "hole" in the computer, through which data is transferred.

Program—The series of instructions that the computer follows to carry out a predetermined task.

RAM—Stands for Random Access Memory, and is the memory onboard the computer that holds the current program. The contents of RAM can be changed, while the contents of ROM (Read Only Memory) cannot be changed under software control.

Real time—When a computer event is progressing in line with time in the "real world," the event is said to be occurring in real time. An example would be a program that showed the development of a colony of bacteria that developed at the same rate that such a real colony would develop. Many games, which require reactions in

real time, have been developed. Most "arcade action" programs occur in real time.

Refresh—Dynamic memories (see memory) must receive periodic bursts of power in order for them to maintain their contents. The signal that "reminds" the memory of its contents is called the refresh signal.

Register—A location in computer memory that holds data.

Representation—The organization of information within a computer so that it can be managed by the knowledge-base control system.

Reset—A signal that returns the computer to the point it was in when first turned on.

ROM—See RAM.

RS-232—A standard serial interface (defined by the Electronic Industries Association) that connects a modem and associated terminal equipment to a computer.

S-100 bus—This is also a standard interface (see RS-232) made up of 100 parallel common communication lines that are used to connect circuit boards within microcomputers.

SNOBOL—A high-level language, developed by Bell Laboratories, that uses pattern recognition and string manipulation.

Software—The program that the computer follows (see firmware).

Stack—The end point of a series of events that are accessed on a last in, first out basis.

Subroutine—A block of code, or program, that is called up a number of times within another program.

Symbolic inference—See inference system.

Syntax—As in human languages, the syntax is the structure rules that govern the use of a computer language.

Systems software—Sections of code that carry out administrative tasks, or assist with the writing of other programs, but which are not actually used to carry out the computer's final task.

Thermal printer—A device that prints the output from the computer on heat-sensitive paper.

Time-sharing—This term is used to refer to a large number of users, on independent terminals, making use of a single computer, which divides its time between the users in such a way that each of them appears to have the "full attention" of the computer.

Turnkey system—A computer system (generally for business use) that is ready to run when delivered, needing only the "turn of a key" to get it working.

VLSI—Very Large Scale Integration of components on a "chip," with present development in Japan and the U.S. aiming for the equivalent of 10 million transistors per chip. (Current chips contain the equivalent of around half a million transistors at most.)

Volatile memory—A memory device that loses its contents when the power supply is cut off.

Word processor—A dedicated computer (or a computer operating a word processing program) that gives access to an "intelligent typewriter" with a large range of correction and adjustment features.

INDEX

Accumulator, 238
Address, 238
Aleksander, I., 227–28
Algebra, Boolean, 5, 239
Algorithm, 53–55, 230, 238
Alice in Wonderland, 89
And/then decisions, 4
Anthropomorphization, 122, 124
APL (Automatic Programming
Language), 238
Application software, *See* Software
Are Computers Alive?, 91, 224
Aristotle, 25
Artificial Intelligence, 238–39
See also book titles, subject headings and programs throughout index
Artificial Intelligence, 224–25
Artificial Intelligence and Natural Man, 89
Artificial Intelligence—An Introductory Course, 226–27
Artificial Intelligence—An MIT Perspective, 227
Artificial Reality, 124–25
Artificial Vision for Robots, 227–28
ASCII (American Standard Code for Information Interchange), 239
See also Character code
Asimov, Isaac, 239
Assembler program, 239
Automatic Natural Language Parsing, 238
Assertion circuit, 6

BASIC, 239
Batch, 239
Baud, 239
Bell Telephone Laboratories, 4, 79, 245
Benchmark, 239
Berry, Adrian, 125
Binary system, 4, 239
BLOCKWORLD program, 84–85, 86
block search, 103–9
complete program, 110–16

BLOCKWORLD program *(continued)*
conversing with, 92–93
described, 86–88, 92
element manipulation, 94–96
improvement ideas, 116–17
modules, 101–3
SHRDLU, compared to, 96–99
test of, 99–100
Boden, Margaret, 89
Boole, George, 5
Boolean algebra, 5, 239
Bootstrap, 239
Boxes: An Experiment in Adaptive Control (1968), 12
Brain, human, 4
Bramer, M.A., 230
Broca's Brain, 124
Brown, David, 83
Brown, R.H., 227
Buffer, 239
Bug, *See* Debugging
Build Your Own Expert System, 228–29
Bundy, A., 226–27
Bus, 239
Byte, 240
See also Kilobyte

Campbell, J.A., 82
Central Processing Unit (CPU), 240
Chambers, Oliver, 180–81
Character code, 240
See also ASCII
Checkers (game), 13–14
ideal program, 216–17
inception, 81
Chess (game)
books on, 229–30, 234
evaluation function, 52, 80
4-ply search, 53
human thinking model in, 81
program, 52
programs and artificial intelligence development, 82
Chess and Computers, 229–30
Chinese Poems, 159
Chip (silicon) selection program, 190–95
Circuits, 5
assertion, 6
decisions, 4, 6–7, 9
negation, 6–7
Colby, Kenneth, 122–24
Compiler, 240
Computer Game-Playing, Theory and Practice, 82, 230–31
Computer Gamesmanship, 230
Computer Power and Human Reasoning, 120, 124, 231
Computer terminology, 238–46
Computers, ethics of use, 123–25, 231
Concatenate, 240
CP/M (Control Program/ Microcomputer), 240
CULT program, 148
Cybernetic Modelling, 91

Data and database, 240
Debugging, 215–16, 218, 239, 240
Decisions circuit, 4, 6–7, 9
Deductive arguments, 25
DENDRAL program, 169
Disk, 240, 241
DOCTOR program, 84, 85, 125, 126–28, 231
Documentation, 221–22, 240

Dot-matrix printer, 240
Dynamic memory, 241

ELIZA program, 84, 85, 87, 231
  complete program, 137–46
  creation, 120–21
  data statements analysis, 129–31
  and DOCTOR program, 121–22, 126–28
  myflag in, 136
  personal identification with, 85, 122, 124
  structure, 133–37
  therapeutic value, 123–24
  triggers, 129, 130–32, 135–37
  use controversy, 123–25
EMYCIN program, 169
EPROM program, 241
Error message, 241
Ethics, computer use and, 123–25, 231
EURISKO program, 170
European Economic Community (EEC), 148
Evaluation functions, 51
  applications, 81
  in checkers, 14
  in chess, 52, 80
  in game programming, 51
  2-ply search, 53
  weighted elements, 51–52
Expert systems
  defined, 241
  DENDRAL, 169
  economic viability, 170
  genetic engineering, 170, 171
  human interface, 242
  if/then construction, 165–66
  knowledge base, 16–17, 243
  limitations, 168
  for medical diagnosis, 165, 169, 172–75, 229
  for molecular structures, 169
  reasoning, use of, 166–68
  and self-learning, 177–95, 196–213
  for silicon chip distinguishing, 190–95
  for software writing, 170
  SPURT, 172–77
  for taxes, 170
  as teachers, 169
  uses, 165, *see also other applicable subheads*
  X-SPURT, 177–95

Feedback, 3
Feigenbaum, Edward A., 166, 170, 231–32
Field, 241
Fifth Generation—Artificial Intelligence and Japan's Computer Challenge to the World, The, 166–67, 170, 231–32
Fifth Generation Computer Systems, 332–34
File, 241
Firmware, *See* Software
Flag, 241
Floppy disk, *See* Disk
Flow chart, 214–16, 241
Foreign languages, *See* Translation
FORTRAN, 241
FRANGLAIS program, 151–58
  complete program, 156–58
  examples, 151–52
  structure, 152–54
  subroutines, 154–56

French-English translation, 149–50

Games, computer-playing, 82–83, 230–31, 234
  See also Joystick; *names of specific games and programs*
Gate, 8, 242
  See also Learning; Reasoning
General Sciences, The, 167
GENESIS program, 171
Genetic engineering, 170, 171
Giant Book of Computer Games, 234
GIGO, 242
Go (game), 82–83, 231
Graphics, 242
Greenberg, Martin, 237
GUIDON program, 169

Haiku, 160
Han-Shan (Chinese poet), 159
HANSHAN program, 85, 159–63
  complete, 162–63
  database, 159–61
  poems produced by, 161–62, 164
Hardware, 241, 242
Hartnell, Tim, 234
Hayes, J.E., 234, 235
Heuristic, 242
Hexadecimal, 242
Hex pad, 242
High-level languages, 242
  See also Language, natural
Human interface, 147, 148, 242
  See also Expert systems; Language, natural

IBM, 13
Idiot savant, 168
If/then construction, 4, 9, 165–66
Illinois, University of, 81
Inference system, 242
  See also Yes/No decisions
Input statements and output, 218–19, 220, 242, 244
Instruction, 243
Intelligence, artificial, *See* Artificial intelligence
Intelligent Systems—The Unprecedented Opportunity, 234–35
Interface Computer Encyclopedia, The, 191
International Business Machines Corp. (IBM), 13
Interpreter, 243
Inverter, 8
Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities, An (1845), 5
I/O, 243

Japan, 231–33, 243
Jones, K. Sparck, 228
Journal of Nervous and Mental Diseases, 123
Joystick, 243

Kilobyte, 243
Klir, J., 91
Knowledge-based systems, *See* Expert systems

Knowledge Information Processing Systems, 243
  See also Expert systems; *Fifth Generation* headings
Krueger, Myron W., 124–25

Languages, computer, *See* High-level languages; Machine language; names, e.g., FORTRAN
Language, natural
  landmark programs in, 8–85
  parsing, 85–86, 89, 228
  problems, 88
  See also High-level languages; Human interface; Poetry; Translation
Lawson, Veronica, 237
Learning
  automatic, 12
  expert system capability, 168
  from feedback, 3
  and game-playing computers, 79–80
  kinds of, 3
  rote, 14
  self-modification, 14
Leibniz, Gottfried, 167–68
Levy, David, 229–30
Logic
  gates, 8, 242
  as mathematics' discipline, 5
  symbolic, 5
  See also SYLLOGY program
Loop, master, 64, 217

McCarthy, John, 124
McCorduck, Pamela, 81, 122, 166, 170, 231–32, 236
Machine Intelligence, 235–36
Machine translation, *See* Translation
Machine language, 243
Machine-man relationship, 120, 123–25, 224, 231
Machines Who Think, 122, 232, 236–37
Massachusetts Institute of Technology (MIT), 120, 121, 227
Master loop, 64, 217
Medical diagnosis, 165, 169, 229
Memory, 241, 243, 244, 245, 246
Michie, Donald, 12–13, 234, 235
Microprocessor, 243
Mikulich, L.I., 235
Mini-maxing, 47–49
Modem, 243
Modules, *See* Subroutine calls
Molecular structures, 169
MOLGEN program, 170
Monitor, 243
Motherboard, 244
Moto-oka, T., 232–33
Mouse, 244
MULTI-SELF-LEARN program, 203–13
  complete program, 207–9, 212–13
  conjunction use, 130
  correction variable, 209–13
  reply database, 129
  variable flag, 206–7
  working, 129
MYCIN program, 169

Natural language, *See* Language, natural

Naylor, Chris, 228–29, 234
Negation circuit, 6–7
Network, 244
Naughts and Crosses, *See* TICTAC
Numeric pad, 244

Observer's Book of Rocks and Minerals, The, 180–81
Octal numbering system, 244
On-line, 244
Operating system, 244
Or/then decisions, 4, 8–9
OTHELLO (game), 82–83
Output, *See* Input statements and output
Ozanne, Ken, 191

PARRY program, 125
Parsing, 85–86, 89, 228
Pascal language, 244
POETRY program, 85, 159–64
Port, 244
Practical Experience of Machine Translation, 237
Print statements, 218–19
PROGRAMMER'S APPRENTICE, 170
Programming a Computer for Playing Chess, 79
Programming techniques
  documentation, 221–22
  explicit variables, 219–20
  flow chart, 214–16
  input statements, 218–19
  print statements, 218–19
  REM statements, 219
  subroutine modules, 216–18
  testing, 216–17, 218
  writing out, 217–18
Programs
  definition, 244
  documentation, 221–22
  specific
    BLOCKWORLD, 84–85, 86–88, 92–116
    CHECKERS, 13–14, 81, 216–17
    CHESS, 53, 80–82, 234
    Chip selection, 190–95
    CULT, 148
    DENDRAL, 169
    DOCTOR, 84, 121–23, 125–28, 231
    ELIZA, 84, 85, 120–46, 231
    EMYCIN, 169
    EURISKO, 170
    FRANGLAIS, 151–58
    GENESIS, 171
    GUIDON, 169
    HANSHAN, 85, 159–63
    MOLGEN, 170
    MULTI-SELF-LEARN, 203–13
    MYCIN, 169
    PARRY, 174
    POETRY, 85, 159–64
    PROGRAMMER'S APPRENTICE, 170
    PUFF, 169
    SELFLEARN, 196–203
    SHRDLU, 84–85, 86–87, 96, 117–19
    SNICKERS, 39, 41–53, 55–78
    SPURT, 172–77
    SYSTRAN, 148–50
    TAXMAN, 170
    TRANSLATE, 85
    X-SPURT, 177–95
Psychotherapy, *see* ELIZA program

PUFF program, 169
RAM (Random Access Memory), 244
Reading list, 223–37
Real time, 244–45
Reasoning
  book on, 120, 124, 231
  in expert systems, 166–67
Refresh, 245, *See also* Memory
Register, 245, *See also* Memory
Relays, 8
REM statements, 66, 219, 221–22
Representation, 245
Reset, 245
Robots, 227–28
Rogers, Carl, 85, 120
RS-232, 245
Ruston, Jeremy, 151

Sagan, Carl, 124
Samuel, Arthur, 13–14, 81–82
Scripts, 121
Search trees, *See* Tree-searching problem solving
Seeing Is Believing, 83
SELFLEARN program, 196–203
  basis, 196–97
  complete program, 201–3
  development, 197–98
  workings, 199–201
Self-learning programs
  MULTI-SELF-LEARN, 203–13
  SELFLEARN, 196–203
  X-SPURT, 177–95
Semantics, 89
Serial processing, 4
Shannon, Claude, 4–5, 8, 79
SHRDLU program, 84–85, 86–87, 96, 117–19
Silicon chip selection program, 190–95
Simons, Geoff, 91, 224
SNICKERS program, 39
  artificial intelligence in, 44–45
  board, 43
  captures, 60–64
  complete program, 72–78
  evaluation function, 49–52
  loop construction, 64–71
  "mini-maxing," 47–49
  moves, 44–47, 49–50, 58–64
  start of game, 41–42
SNOBOL language, 245
Software, 170, 238, 245
Some Studies in Machine Learning Using the Game of Checkers—II—Recent Progress (1967), 13
S-100, 245
SPURT program, 172–77
Stack, 245
Stanford University, 169
Stibitz, George, 4, 5, 8
Subroutine calls, 216–18, 245
  See also specific programs
Super-Intelligent Machine, The, 125
Switches, on-off, 5–9
Symbolic inference, *See* Inference system
Symbolic logic, 5
Syntax, 89–91, 245
Systems software, *See* Software
SYSTRAN program, 148–50

TAXMAN program, 170
Teaching, *See* Expert systems
Testing, program, 217, 218
Thermal printer, 245

Thinking machines
book on, 237
Boolean algebra basis, 5
early, 4–5
serial processing, 4
switches and circuits, 5–10
Those Amazing Electronic Thinking Machines!, 237
Three Mile Island accident, 80
Three Reviews of J. Weizenbaum's Computer Power and Human Reason, 124
TICTAC program, 1–3
complete program, 20–24
described, 11–13
feedback, 3
format, 15–24
strategy, 18–20
trial and error, learning from, 3
Time-sharing, 123, 245
TRANSLATE program, 85
Translation, machine
book on, 237
differing approaches to, 148
editing, pre and post, 148, 149, 150
FRANGLAIS program, 151–58
French-English, 148, 149–50
human role, 147, 148, 150
non-literary, 150
Tree-searching problem solving
Alpha-Beta algorithm, 53–56
described, 39–41
pruning, 46–47, 49
syntactic structure, 89–90
2-ply search, 53
See also SNICKERS program
Trigger words, 129, 130–31, 135
"Truth table," 6, 7
Turnkey system, 245
Turtle (machine), 3
Two-ply search, 53

Understanding, computer, 87, 91

Valach, M., 91
Variables, 219–20
VLSI, 246
Volatile memory, 246

Waley, Arthur, 159
Walter, Grey, Dr., 3
Waugh, Charles E., 237
Weizenbaum, Joseph, 87, 120–25, 231
Wilks, Y., 228
Winograd, Terry, 96–97, 117
See also SHRDLU
Winston, Patrick Henry, 224–25, 227
Wirth, Niklaus, 244
Word processor, 246
Writing programs, *See* Programming techniques

Xerox Corp., 148
X-SPURT program, 177–95
acquiring knowledge, 182–83
application, 180–82
complete program, 187–89
construction, 183–87
disadvantage, 196
unlimited knowledge base, 177–79
working, 180

Yes/No decisions, 173

Can Your Apple® II Really Think?

Maybe you didn't know it. But it can—*if* you teach it. *Exploring Artificial Intelligence on Your Apple II* introduces you to the fascinating world of "smart" computer programming that uses basic logical maneuvers to think, solve problems, make predictions—even to write poems! Written in clear, easy-to-understand language, it explains step by step what artificial intelligence is, what it can do, and how you can use it on your Apple II. Sample programs in BASIC are explained in modular fashion to help the reader understand the way the Apple II thinks.

- THINKING: How computers can learn and reason: an introduction to artificial intelligence
- SEARCHING: How computers use "search trees" to make intelligent and time-saving decisions
- TALKING: How to talk to your computer and how to get it to talk back: an overview of how computers can manipulate language, and deal with problems of syntax and semantics
- HELPING: An introduction to advanced expert systems: how your computer can learn from its experience, correct mistakes, make predictions

PLUS...*Improving your programming technique, suggestions for getting the most out of your Apple II, and much, much more.*

TIM HARTNELL, author of such current bestselling computer books as *Creating Adventure Games*, is an Australian journalist who has written on a broad range of computer-related topics. His fascination with artificial intelligence has resulted in this book and his upcoming *Exploring Expert Systems on Your Microcomputers.*

N 0-553-34261-4>1495

34261-4 ✱ IN U.S. $14.95 (IN CANADA $17.95) ✱ A BANTAM BOOK