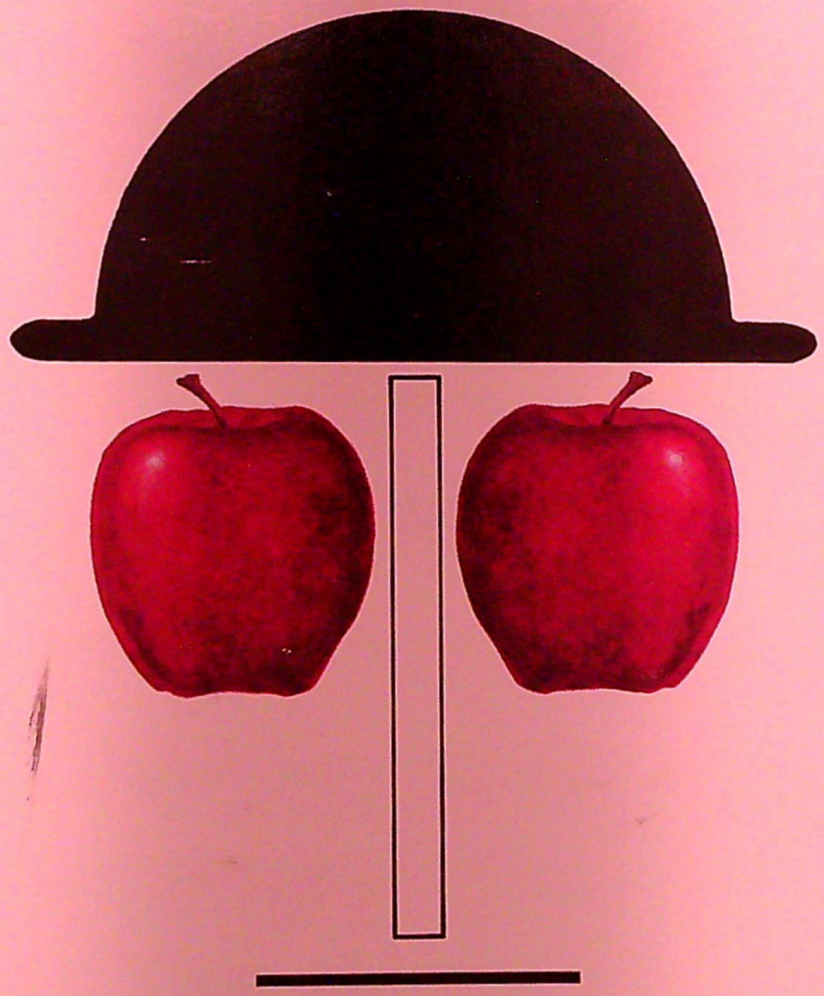
2775

SMART APPLES

31 ARTIFICIAL INTELLIGENCE EXPERIMENTS WITH THE APPLE II®, II+®, *II*e®, *II*c® and IIGS®



Delton T. Horn

SMART APPLES

31 ARTIFICIAL INTELLIGENCE EXPERIMENTS WITH THE APPLE II,® II+,® *IIe*,® *IIc*® and IIGS®

Delton T. Horn

TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

FIRST EDITION
FIRST PRINTING

Copyright © 1987 by TAB BOOKS Inc.
Printed in the United States of America

oduction or publication of the content in any manner, without express
ission of the publisher, is prohibited. No liability is assumed with respect to
the use of the information herein.

Library of Congress Cataloging in Publication Data

Horn, Delton T.
Smart Apples.

Includes index.
1. Apple II (Computer)—Programming. 2. Artificial
Intelligence—Computer programs. I. Title.
QA76.8.A662.H67 1987 005.265 87-14007
ISBN 0-8306-7755-0
ISBN 0-8306-2775-8 (pbk.)

Questions regarding the content of this book
should be addressed to:

Reader Inquiry Branch
Editorial Department
TAB BOOKS Inc.
P.O. Box 40
Blue Ridge Summit, PA 17214

Contents

List of Programs iv

Introduction v

1 What Is Artificial Intelligence? 1

Some Historical Background—Computer Thought—Computer Error—Applications for Artificial Intelligence

2 Decision Making 12

Simple Decisions—Boolean Tests—Option Weighting—Defining Problems—State Graphs—Tree Searches—Depth-First Search—Breadth-First Search—Making the Choice—Chapter Summary

3 Introduction to Game Theory 49

Kala—Frustration—Tic-Tac-Toe

4 Advanced Games 83

Gomoku—The Minimax Method—Heuristics

5 The Turing Test 104

The Turing Test Procedure—Practical Limitations—Expert Systems—A MI Program

6 Conversation 1 111

Automated Translation—Semantics and Syntax—Conversation 1—Conversation 2

7 More On Conversation Programs 135

ELIZA—Parry—The Mad Doctor

8 Computer Creativity 149

Who Is the Artist?—Poetry Generators—Music Composition

9 Story-Writing Programs 173

Story-Writer 1—Story-Writer 2—Going Further

10 Programming Languages 187

BASIC—Pascal—Assembly Language—LISP

Index 191

List of Programs

Listing #	Title
2-1	OPTION WEIGHER 37
2-2	1ST MAZE 39
2-3	2ND MAZE 40
2-4	SIMPLE TREE SEARCH 41
2-5	LARGE TREE SEARCH 43
2-6	DEPTH-FIRST SEARCH 45
2-7	BREADTH-FIRST SEARCH 47
3-1	MANUAL KALA 61
3-2	AI-KALA 64
3-3	SOLO-FRUSTRATION 67
3-4	DUO-FRUSTRATION 69
3-5	DUMB-FRUSTRATION 72
3-6	SMART-FRUSTRATION 75
3-7	MANUAL TIC-TAC-TOE 78
3-8	AI TIC-TAC-TOE 80
4-1	MANUAL GOMOKU 95
4-2	AI GOMOKU 100
6-1	SIMPLE CONVERSATION 127
6-2	ADVANCED CONVERSATION 130
7-1	THE MAD DOCTOR 138
8-1	SIMPLE POETRY 158
8-2	ADVANCED POETRY 160
8-3	RANDOM COMPOSER 166
8-4	WEIGHTED COMPOSER 167
8-5	WEIGHTED W/BACK CHECK 168
8-6	TUNE IMITATOR 169
8-7	TUNE SHAPE IMITATOR 171
9-1	STORY WRITER-1 180
9-2	STORY WRITER-2 184

Introduction

Artificial intelligence, or AI, is a fascinating field of computer applications. In a way, AI represents an idealized goal that can never be completely reached, but as technology and research progress, we can come closer and closer to the theoretical ideal.

In the last few years, the general public has begun to hear about artificial intelligence. Articles on AI have appeared in many general magazines. To most of us, the whole idea of artificial intelligence is something right out of science fiction, and extremely futuristic. To many, it is even rather frightening.

But it isn't really such a new idea. The dream of man-made intelligence has been around for centuries. Many old myths feature the concept in one form or another. But it is only in recent years that technology has reached a point that makes the dream seem feasible, at least to some degree.

When computers were first invented, they were called "thinking machines," or "electronic brains," even though they were incapable of thought in any meaningful sense. More properly, they were glorified adding machines.

In the last few decades, a number of laboratories began serious research into artificial intelligence. Initially, all such experiments were performed on large mainframe computers. At first, that was the only kind of computer in existence. Later, when smaller minicomputers and microcomputers were developed, AI research remained confined to mainframe computers, because of the enormous memory and speed requirements.

Microcomputers have improved enormously in the last few years, in terms of computing power, speed, and memory storage capabilities. They are still no match for the room-filling mainframes at major universities. Serious AI research will probably be limited to large laboratories indefinitely—certainly for the foreseeable future.

However, personal computers are advanced enough now that we can perform a number of crude, but fascinating AI experiments. The experiments we will be working on here would be overly simplistic and "old hat" to a professional researcher in the field, but this book is intended for hobbyists.

Besides being fun, working with these AI experiments will improve your programming techniques, and you will learn more about how the mind works.

The projects in this book are written for the Apple II. This is one of the more popular hobbyist computers around today. If you happen to own an IBM, or Tandy, or

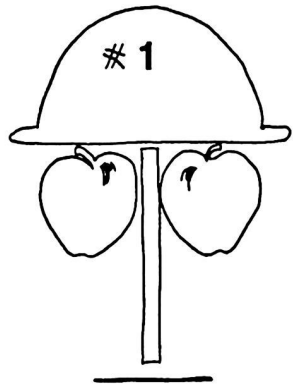
Commodore (or almost any other popular micro), don't despair. These programs don't involve any exotic commands or machine language routines. They can easily be adapted to run on almost any microcomputer that uses any version of Microsoft BASIC, with just a few minor changes. Some of the programs can be run directly on other computers with no changes at all.

Adapting the programs should be no trouble, since the emphasis in this book will be on using the various AI programming techniques for customized applications. You are strongly encouraged to experiment and make changes in all the programs. While many complete, functional programs are included in this volume, they are intended primarily as examples. Use them as starting points for your own experimentation.

All of the programs in this book are written entirely in BASIC. Admittedly, this is not the best programming language for serious AI work. Other programming languages will be discussed in Chapter 10.

BASIC is perfectly adequate for getting started. The choice was made for several reasons. BASIC is included on almost every modern personal computer. It is reasonable to assume that every reader with access to a computer has access to and some familiarity with BASIC. We won't have to waste space here covering the basics of programming. If you do not know how to program in BASIC, you should read a good text on programming fundamentals before trying to understand the techniques described in this volume.

I feel BASIC is also a good choice for our purposes here, because BASIC programs are relatively clear and easy to understand. English-like commands are used in an English-like syntax. I feel the techniques we will be exploring in this book are easiest to understand with sample programs in BASIC. Remember, BASIC was originally devised as a beginner's programming language. It certainly has its flaws and limitations. But it can do the job just fine.



What Is Artificial Intelligence?

Before we can get into AI programs, we need to define our terms. Just what do we mean by artificial intelligence anyway?

Usually the definition is given something like this; "*Artificial intelligence* is any process that permits mechanical (constructed, non-organic) device to perform tasks which would require intelligence to be performed by a person."

That definition certainly sounds reasonable enough, but it is not entirely satisfactory. Perhaps it is a bit too simplistic. I think most of us would agree that performing mathematical operations would require some intelligence for a person to do. A pocket calculator or even an old fashioned mechanical adding machine cannot only perform mathematical operations, it can do so more accurately and faster than a human being. Yet, I seriously doubt that many of us would be willing to claim to admit these devices are intelligent, at least, not in any meaningful way.

Clearly, defining artificial intelligence is no easy task. A reasonable first step would be to define natural intelligence. What do we mean when we say a person has intelligence? Unfortunately no one has managed to come up with a definitive definition.

The dictionary in my office offers the following definitions for the word "intelligence;"

- ☆ the faculty of understanding: capacity to know or apprehend
- ☆ an intelligent being: a person of some intellectual capacity
- ☆ the act of understanding: comprehension: knowledge

(Webster's Third New International Dictionary—G. & C. Merriam Co. 1976)

These definitions tell us what the word means, but don't really explain the concept of intelligence very well. A big difficulty in defining artificial intelligence is that the public's concept of intelligence keeps shifting. If task A is assumed to require intelligence, by the time someone has programmed a computer to perform this task, people start saying that task A didn't really require true intelligence in the first place. It's a form of Catch 22. If it can be accomplished with AI techniques, then it can't really require intelligence.

The reason for this is that we can see the steps a computer takes in "thinking," while human thought appears to be an almost mystical and mysterious process. Modern computers, at least, certainly don't "think" in the same manner as humans. AI supporters reply, "So what?" Ultimately, it is the results that count, not the steps taken to achieve said results. AI programs attempt to simulate (not necessarily duplicate) intelligent behavior. It is the behavior we are interested in, not in how it comes about.

While some AI research has the examination of human thought processes as its fundamental goal, most practical AI programs written today make no attempt to duplicate the step by step processes of human thought, as long as the task goals are met.

A human brain has millions of synapses that work together in various patterns to create thought. Even the largest computer in existence today can't come remotely close to the memory capabilities of the human brain.

For the micro-computers we will be using in this book, the inherent inefficiency of directly imitating human thought processes is clearly out of the question. Far too much memory and computing time is required. While computers perform mathematical operations with incredible speeds, the multitude of instructions involved in AI tasks can make the program run ridiculously slow.

Even in AI laboratories there is considerable debate over whether or not a computer can ever truly duplicate human thought processes. Personally, I suspect that true duplication is theoretically impossible, at least for digital computers. While I am no expert, what I have read in this area has convinced me that human thought is an analog, rather than digital process.

Even so, it is probably possible for a computer to perform most, if not all human tasks, at least in theory.

In some cases, we are still too hazy on what the human tasks involve, so we can't come remotely close to simulating them on a computer. For example, there is the thorny question of creativity. No one really knows just what it is. If the definition of "intelligence" is a tangle, the definition of "creativity" is a hopeless quagmire. It's often virtually impossible to get two experts to agree on whether or not a given task involves creativity.

Computer creativity will be covered in Chapters 8 and 9. Two major stumbling blocks are the computer's lack of emotional response, and judgement. Still, some rather amazing things can be achieved in this area.

In working with AI on your personal computer, don't form any unreasonable expectations. You certainly aren't going to create a new HAL9000 (from "2001") or C3P0 (from "Star Wars") on your micro-computer. That level of generalized intelligence has never yet been achieved, even on the largest mainframe computers in major AI labs. Autonomous, generalized artificial intelligence is still entirely in the realm of science fiction.

However, it is possible today to create a reasonable degree of specialized artificial intelligence. Modern AI programs are "idiot savants." They are "smart," or "knowledgeable," only in one narrow area. In all other areas, they are completely ignorant, and even incapable of learning. Some AI programs can do some independent learning within their narrow subject confines.

SOME HISTORICAL BACKGROUND

Despite popular opinion, the computer is not really a new invention. It's roots go way back to the ancient abacus, which is a manually operated calculating device. It is basically made up of a wooden frame, and a number of beads on strings. The beads are moved about on the strings to represent various mathematical operations. Each bead is a rough equivalent of a bit in a modern digital computer.

The abacus certainly was not a true computer (in the modern sense), but it is an unquestionable ancestor.

Most people think the term "computer" can only refer to a specific class of electronic devices. Such a definition is a bit too restrictive. A more general definition might be "a computer is a programmable automated calculating device." This calls for a couple of secondary definitions. "Programmable" means that the device can be given a number of operating instructions in advance to perform any of several possible tasks. "Automated" means that once programmed, the computer can execute its stored instructions in sequence to complete the task without direct human supervision or interaction.

A Frenchman named Blaise Pascal invented a mechanical adding machine. This was not a true computer, according to our definition. It is not programmable, but it was semi-automated, at least, compared to the manually operated abacus.

This might not seem to be a very exciting, or significant step in the development of the modern computer. But Pascal's contributions are recognized in the field. A popular programming language was named after him.

A German by the name of Gottfried Wilhelm von Leibniz improved Pascal's adding machine concept. In 1671 he designed a machine that could both add and multiply. Unfortunately, it took him awhile to build a working prototype. The first practical model was built in 1694.

The next important name in computer history is Charles Babbage (1792 - 1871). This man was serious about developing a computing device. He spent fifty years (and a lot of government money) designing two complicated contraptions, which he called a difference engine, and an analytical engine.

These devices were genuine mechanical (as opposed to electronic) computers. They were limited by today's standards, but so were the first electronic computers in the 1940s and 1950s.

Unfortunately, Babbage's years of work didn't bear much fruit. The technology of the day was not up to Babbage's imagination. A complete working prototype was never completed for either of Babbage's remarkable inventions. They would have worked, but there was no way at the time to achieve the required level of precision in building the mechanical components.

It is fascinating to contemplate how history might have been changed if Babbage's

mechanical computers had been practical. Computers in the nineteenth century!

Towards the end of the nineteenth century, the U.S. Census Bureau held a contest for the design of a machine that could help them cope with their ever increasing piles of data to be correlated. The winner was Dr. Herman Hollerith, he won this contest with an electric tabulator which recorded data on punched cards. It wasn't a true computer, but it furthered the technology needed to make the computer a practical reality.

This electric tabulator, and related devices which evolved from it, were found to have a number of practical applications in business. Naturally, Hollerith went into business manufacturing such equipment. As time went on, several mergers took place. Eventually, in 1924 to be exact, the name of the corporation was changed to International Business Machines, or IBM.

I'm sure you recognize that name, and you probably suspect that we are finally getting close to the appearance of genuine computers.

Several near-computers were developed over the next few years, by IBM, and other companies. It is hard to pinpoint just where the computer was born. But it is generally agreed that the first fully automatic digital computer was the Automatic Sequence Controlled Calculator—Mark I. This huge machine was completed by IBM in 1944 for the U.S. Navy. When I say this gadget was big, I mean it. It was eight feet high and fifty-one feet long. It was considered an amazingly intelligent machine at the time. Actually, its overall capabilities were less than the average personal computer of today. Its functions were closer to those of a programmable scientific calculator.

The Mark I was an electromechanical rather than electronic device. Its programs were externally stored on a roll of punched paper tape, rather like the scroll found in a player piano.

Next came the thirty ton Electronic Numerical Integrator and Computer (or ENIAC) in 1945. The ENIAC offered several advantages over the Mark I. The external paper tape programs were eliminated. Programs were physically wired in the machine by hooking up various circuits in a manner resembling a telephone switchboard.

From then on, new developments started appearing at an increasingly rapid rate. The history of the computer in this period is a barrage of acronyms; ENIAC, EDVAV, AVIDAC, ILLIAC, SILLIAC, and so on.

Once computers were in existence, attempts to program them for artificial intelligence were probably inevitable. It is impossible to say just where and when AI programming originated. The line between AI programming and regular programming is incredibly thin and nebulous. Many important early experimenters never used the term "artificial intelligence."

Most of the early AI programs were developed more or less independently. It is only recently that AI has been considered a distinct field worthy of serious attention (i.e., financing by major corporations, and the government). Probably the majority of early AI experimenters didn't know what the others were doing. This probably was beneficial because each programmer had to develop his own approach, so the field of AI as a whole did not fall into any dead-ends of premature standardization.

This is not the place to try to sort out the tangle of early AI programming history. I will mention a few important examples here to give you an idea of the scope of the

work done in this field. They will be presented here in conceptual, rather than chronological order because they represent several divergent paths. Strict chronological sequencing would not only be meaningless, it would be confusing.

Game playing was often the goal of AI programmers. This is because the tasks to be performed by the computer are clearcut and easily defined. The relative success (intelligence) can also be easily measured for a game playing AI program. We only have to consider how well does it play the game? Can it win against a good human player?

Many different games have been AI programmed, ranging from simple games like Tic-Tac-Toe, to complex games like Chess. Arthur L. Samuels's checkers playing program is particularly fascinating. This program is capable of learning (although it can only learn how to play a better game of checkers. It can never learn to play Backgammon, for example.) Whenever this program loses a game, it remembers what moves it made, and how its opponent responded. Once it identifies a "bad move", it never makes the same mistake again. It has learned to avoid any move that got it into trouble.

Allen Newell and Herbert Simon developed a General Problem Solver program. This program relies on "rules of thumb." Instead of just blindly searching through piles of data, it first sorts through general categories to determine the most likely location of the solution to the problem it is currently considering.

Another common branch of AI programming is in the area of natural language processing. Computer programming languages are artificial languages created specifically for the purpose. Such languages are precise and unambiguous. Each word has one (and only one) clear-cut and unchanging meaning. This works well with the computer's strict black and white view of the world.

Natural languages, such as English, or French, aren't nearly so cooperative. Natural languages are full of ambiguities of context, syntax, and even definition. What does the word "lead" mean, for example. Is it a verb meaning "to guide," or is it a noun meaning "a type of metal"? Natural languages are filled with such inconsistencies and ambiguities.

This was soon discovered by AI researchers. In the mid-fifties considerable effort was placed on creating a program to translate between Russian and English. The results were disappointing, at best. What should have been translated as "The spirit is willing, but the flesh is weak," ended up as the Russian equivalent of "The wine is agreeable, but the meat has spoiled." Obviously, this was a less than satisfactory translation. Yet, it was typical of the output from the translation programs devised by the best researchers in the area. The effort was eventually given up as a lost cause.

In more recent years, considerable progress has been made in natural language processing, but the translation project is still a bit over ambitious.

Probably the most famous AI program is Eliza, which was created by Dr. Joseph Weizenbaum. This program plays the part of a Rogerian psychiatrist, with a human operator taking the role of a patient. The computer carries on a conversation with the human operator. Sometimes, the results are perfectly uncanny. (The computer's dialogue is in ALL CAPITALS.):

WHAT IS BOTHERING YOU?

I hate my brother George.
TELL ME MORE ABOUT YOUR FAMILY
They never believed me when I told them about George.
PLEASE GO ON
George used to beat me up.
GEORGE USED TO BEAT YOU UP?
Yes. It was terrible.
YOU SOUND QUITE POSITIVE

The computer certainly isn't contributing very much to this conversation. But Rogerian psychotherapy is supposed to be "nondirective." The computer certainly seems to understand what the "patient" says, and it responds accordingly.

In truth, Eliza doesn't "understand" any of the conversation. It simply plays semantic tricks according to simple rules. If the user input doesn't fit into the patterns, the program will cheerfully spew out utter gibberish, like:

EARLIER YOU SAID YOUR NEW CAR.
What do you mean by that?
WHY DO YOU ASK?

This kind of thing can be amusing, but it hardly qualifies as intelligence. In fact, Weizenbaum has become an opponent of the concept of artificial intelligence.

A related program is PARRY, which plays the part of a paranoid patient. ELIZA and PARRY will be discussed in more detail, and simplified versions of this type of program will be presented in Chapters 6 and 7.

More recent programs do seem to have some understanding of a conversation that is kept within limited bounds. In 1971 Terry Winograd completed a program called SHRDLU. This program appears fairly intelligent about what is going on within a miniature, imaginary world comprised of several blocks on a tabletop. A typical exchange with SHRDLU might be something like this:

Pick up the large green block.
OK.
Find a block which is longer than the block you are holding and put it behind the box.
OK.
Take the blue triangle out of the box.
OK.
What is in the box?
A SMALL GREEN BLOCK AND A RED TRIANGLE.
What are you holding?
THE LARGE GREEN BLOCK AND THE BLUE TRIANGLE.
Where is the biggest red block?
BEHIND THE BLUE BOX.

SHRDLU is not just playing semantic tricks like ELIZA. It appears to understand what is going on.

However, SHRDLU's intelligence is extremely limited. It can never expand its horizons beyond its handful of blocks. It has no idea of what a tennis shoe is, and it can't learn about tennis shoes. It only knows about its blocks.

Other programs have been written to understand stories. Information (stories) is entered into the computer, then the computer is asked questions about items which are implied, but never explicitly stated in the story. If the computer can deal successfully with such questions, it doesn't seem too unreasonable to say that it "understands" the story on some level.

For example:

> Bob went into a restaurant and ordered a hamburger.
> I UNDERSTAND
> Why did Bob go into the restaurant?
> HE WAS HUNGRY AND WANTED SOMETHING TO EAT.
> What was he wearing?
> I DON'T KNOW.
> Did he get waited on?
> YES. HE ORDERED A HAMBURGER.

These programs can be astonishing in what they appear to understand. However, any "common sense" they may have must be programmed into it.

I once read about a similar program which had to make decisions within a story. It was told it wanted to get the newspaper from the front lawn, but it was raining out, and the program was told it didn't want to get wet.

The program knew that if it was raining out and it didn't want to get wet, it should put on a raincoat. Unfortunately, the programmer neglected to tell the program that wearing a raincoat would prevent it from getting wet. The computer's output was along these lines:

> I LOOK OUT THE WINDOW
> I SEE IT IS RAINING
> I PUT ON A RAINCOAT
> I LOOK OUT THE WINDOW
> I SEE IT IS RAINING
> I PUT ON A RAINCOAT
> I LOOK OUT THE WINDOW
> I SEE IT IS RAINING
> I PUT ON A RAINCOAT
> I LOOK OUT THE WINDOW...

The stupid computer insisted on standing in the imaginary front hall, looking out the window and putting on raincoat after raincoat.

Do these programs really understand anything in a meaningful way? This is a difficult question that will be alluded to throughout this book. But don't expect a definitive answer. The experts are hotly debating this issue, and will probably continue to do so for some time to come.

COMPUTER THOUGHT

Computers do not think like people. This is not to imply that one is necessarily better than the other. The processes used in each case are inherently different.

In some AI research considerable effort is placed on getting the program to mirror human thought processes as much as possible. In these programs the goal is not to accomplish specific tasks, but to study natural intelligence.

Most AI programs are more goal oriented. The idea is to accomplish a specific task, such as playing a game, or to translate some text. Attempting to duplicate human thought processes in such programs would be extremely inefficient, and probably impossible. But such duplication isn't really needed. Ultimately, it's the results that count. The computer and the human brain may take two entirely different paths, but they reach the same destination (at least, that's the goal). To work with artificial intelligence, we need to understand the computer's thought processes.

On their most fundamental level, computers are incredibly stupid gadgets. The largest, most expensive, sophisticated computer in the world can do one thing, and one thing only. It can distinguish between a high voltage signal and a low voltage signal. And that is *all* it can do. Each such signal is called a bit.

A bit can exist in one of two conditions or states. Either it is a high voltage or a low voltage. It can not ever take on an intermediate value.

The two bit states may be identified in any of a number of ways. The most commonly used names are:

HIGH	LOW
1	0
ON	OFF
+	-
YES	NO

Each of these terms pairs mean exactly the same thing. Which one we should use in a given situation will depend on which is the most convenient for the specific application. The name used doesn't change anything, except how we talk about the state in question.

In our discussion it will probably be most convenient to call the high voltage state "1", and the low voltage state "0". Computer signals are treated like binary numbers. The binary system is made up of just two digits, 1 and 0. We are more familiar with the decimal system, which has ten digits (1, 2, 3, 4, 5, 6, 7, 8, and 0). These are just two different ways of counting. You do not have to know how to perform binary math to work with AI, so we won't go into the details here.

Using our new terminology, we can say that the computer can tell the difference between a 0 and a 1. This certainly doesn't sound very impressive, or useful. But it is all any computer really does. But the computer can do this task so efficiently, that literally millions of bits can be checked in less than a second.

Bits are grouped into chunked patterns. These patterns are variously called "nibbles" (four bits), "bytes" (eight bits), and "words" (sixteen or thirty-two bits).

Since each bit can take on either of two exclusive states, there are sixteen possible nibbles:

1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	0000

Similarly, there are 256 possible bytes, 65,536 possible sixteen bit words, and 4,294,967,296 possible thirty-two bit words. Obviously, the greater the number of bits, the greater the variety of possible combinations.

Moreover, nibbles, bytes, and words can be combined to create even larger patterns.

The computer can examine a bit pattern at its input, and then generate a different bit pattern at its output in response. The specific relationship between the input and output patterns is determined by the computer's program.

That is all that any computer does. It looks at incoming signals (bits) and generates outgoing signals (bits) in response according to a previously defined set of rules (the program).

So what? Well, the various bit patterns can mean anything we want. They can be used to represent encoded data. For example:

0100	0001	A
0100	0010	B
0100	0011	C
0100	0100	D
0100	0101	E
0100	0110	F

and so forth. Any data can be encoded in this manner.

We won't go into the details of how data is encoded here. Since we will be using standard personal computes with ASCII keyboards, we don't have to worry about the encoding. We just type the data directly into the keyboard, and the computer system's hardware takes care of the encoding for us. (ASCII, incidentally, is a standardized encoding system that is used in virtually all modern computer equipment.) You just

need to remember that encoding does take place. The computer does not deal with any data directly, only an encoded representation.

If you want to know more about these processes, many fine books on the inner workings of computers are available.

In AI work, we are more concerned with higher level computer activity (input and output patterns). When we think, we aren't concerned with the synapses firing in our brains. Similarly, why should we bother with bits when working with AI systems?

But we can save ourselves a lot of confusion and frustration if we are aware that the computer's "thought" processes are fundamentally from our own.

COMPUTER ERROR

Probably every one of us has had some personal experience with "computer error." It can be difficult to fight a mistake in a computerized bill, for example. We start cursing, "Those damned computers!"

In over 99 percent of the cases of "computer error," the computer is taking a bum rap. True computer error is very rare. Usually "computer error" is "programmer error," or "operator error."

People will often put blind trust in the computer. "That's what our computer says. Computers don't make mistakes, you know." It is vital to remember that computers have absolutely no common sense. The nice thing about computers is that they do exactly what they're told. The bad thing about computers is that they do exactly what they're told.

Remember the case of the AI program that knew it was supposed to put on a raincoat before going out in the rain, but didn't know (and couldn't figure out) that the rain coat would keep it from getting wet.

Some years ago, a utility company on the east coast converted to a new computerized billing system. The programmer forgot to tell the computer what to do if the balance owned by a customer was zero. Many customers received dunning letters from the computer demanding immediate payment of $0.00.

Everything must be explicitly spelled out in a computer program, especially for AI work. Nothing can be taken for granted. A computer can only follow directions, it can make no assumptions on its own.

APPLICATIONS FOR ARTIFICIAL INTELLIGENCE

The field of artificial intelligence covers quite a bit of ground. There are a great many applications for AI research. One that has already been mentioned in this chapter, is to study natural thought processes via duplication.

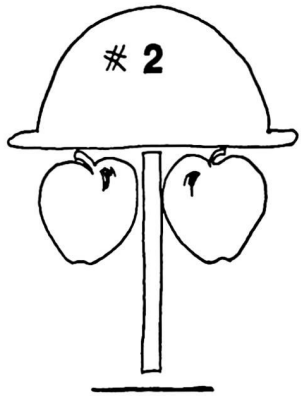
Other AI applications include solving specific problems that would require too many man-hours otherwise, or that might have too many variables to be conveniently handled by a human.

AI is used in robotics and automated manufacturing. Related areas include image/scene analysis, speech recognition, and Computer Aided Design (CAD).

Games can be used in commercial entertainment systems. A number of artists are using computers to spur their creativity, and to explore new art forms. Some AI programs have been created which can write new programs based on input parameters.

Natural language processing is another important class of applications for artificial intelligence. Much has been learned about human communication and assumption forming through these programs. Related techniques can be used in creating software that can serve as educational aids.

We will touch upon many of these AI applications throughout this book. I feel I should end this chapter by mentioning what is probably the most important reason for experimenting with AI on a hobbyist level—it can be very exciting, and a lot of fun.



Decision Making

It would not be unreasonable to call a computer a decision making machine. It "decides" whether each bit is a 1 or a 0. It "decides" what action should follow the current input pattern. It "decides" where it will next look in its memory bank.

Of course we are using the word "decide" very loosely here. All of the computer's decisions are the result of hard and fast rules, with absolutely no room for variation. If all of the input conditions are duplicated exactly, the output of any computer program will be exactly the same.

BASIC (and other programming languages) includes a RND command, which allows the computer to select a random number. (We will be using this command a lot throughout this book.) But the numbers generated by such commands are not truly random. They are more properly known as pseudo-random numbers. The randomness is essentially an illusion, although the apparent randomness will be good enough for most processes.

Usually a pseudo-random number is generated by performing a number of complex mathematical operations on a "seed" value. If the same seed value is used, the same number will be returned by the pseudo-random generator. After a pseudo-random number has been generated, the seed value is updated by the generated number. Consequently, the next time a pseudo-random number is generated, it will (probably) be different, because a new seed value has been used.

Generally the best source of an initial seed value is an internal timer/counter within the computer. The timer/counter value is continuously being updated as long as power is applied to the computer. The current value at the instant the RND function is called is used as the initial seed value.

Some versions of BASIC (and certain other programming languages) allow the operator to enter a seed value manually. If you enter the same seed value for two different runs of a program, the same series of pseudo-random numbers will be generated.

Any pseudo-random number generator will eventually start to repeat itself. The same sequence pattern of numbers will be generated over and over. A good pseudo-random number generator will produce a very long sequence before repeating. This will effectively disguise the redundant nature of the sequence. The user will not be able to predict what the next generated value will be. This is usually all that is necessary for a practical pseudo-random number generator.

The important thing to remember is that the computer does *not* make up random numbers out of the clear blue sky. It is only rigidly following its programmed rules.

While we can say that a computer makes decisions, there is no trace of free will in such decision making. If everything is repeated exactly, the computer will make precisely the same decisions, 100 percent of the time.

SIMPLE DECISIONS

The simplest decisions are clear-cut yes or no choices. Such decisions are easy to program. BASIC offers a single program command for this purpose. It is the IF . . . THEN . . . command. It generally takes this form:

IF (condition to be examined) THEN (perform appropriate action)

For example:

```
IF X = O THEN GOTO 100
```

When the computer executes this command, it looks at variable X. If this variable holds a value of 0, the program jumps to line 100. If X has any other value, the "GOTO 100" is simply ignored, and the computer proceeds to execute the next command in sequence.

IF . . . THEN . . . decisions do not have to be based on simple equality. Comparatives can also be used. For instance:

```
IF X < 4 GOTO 100
```

The symbol < means less than. With this command, the program will jump to line 100 if X = 0, X = −1, X = 2, X = 3.4, etc. On the other hand, if X = 4.5, X = 5, X = 97, or X = 32098, the "GOTO 100" will be ignored.

Alternatively, this symbol can be used, >. It means greater than. It works in just the opposite way as <.

The two symbols can be used together (< >) to test for inequality. For example, when this command is:

```
IF X < > 0 GOTO 100
```

the "GOTO 100" will be ignored only if X has a value other than 0.

In some cases we will need one action to be performed if the tested condition is true, and a second action to be performed if the tested condition is false (not true). This can usually be accomplished by adding an ELSE . . . clause to the IF . . . THEN . . statement. To demonstrate this, here is a simple demonstration program for the IF . . . THEN . . . ELSE . . . command:

```
10 PRINT "ENTER A VALUE FOR X"
20 INPUT X
30 PRINT "ENTER A VALUE FOR Y"
40 INPUT Y
50 IF X> Y THEN PRINT "X IS LARGER THAN Y" ELSE PRINT
   "Y IS LARGER THAN X"
60 GOTO 10
```

Enter and run this program. Try various values for X and Y. The computer will respond with the appropriate statement. Either:

```
X IS LARGER THAN Y
```

or:

```
Y IS LARGER THAN X
```

This program will always print the correct information with one single exception. Try entering identical values for both X, and Y. Let's say, 5.

In line 50, the computer will compare the values of X and Y:

```
IS X > Y?
```

That is:

```
IS 5 > 5?
```

No, 5 is not greater than 5, so the computer dutifully prints out:

```
Y IS LARGER THAN X
```

Of course that is not true. Once again, the computer is only doing exactly what it was told to do. It was never told to check for equality, or to make sure that Y > X before printing out the second message.

Clearly, some additional programming will be required to get this program to function properly.

Some versions of BASIC will allow you to chain several IF . . . THEN . . . tests together in a single command.

```
IF X > Y THEN PRINT "X IS LARGER THAN Y"
```

```
ELSE IF X = Y THEN PRINT "X = Y"
  ELSE PRINT "Y IS LARGER THAN X"
```

If we replace line 50 of our sample program with this expanded command, the program will give the correct results for all values of X and Y.

Let's try a few examples and see just how the computer deals with this command.

```
X = 8
Y = 3
```

The computer determines that X is greater than Y, and prints out the message:

```
X IS GREATER THAN Y
```

It then ignores the rest of line 50 from the first ELSE on. The program goes ahead to the next line (60).

Now let's see what happens when X and Y are equal:

```
X = 7
Y = 7
```

The first test (X > Y) fails (is false), so the THEN command is ignored. The computer skips through the line until it finds the code word ELSE. Now it finds a command to perform a new test (X = Y). This test is true, so the next THEN command is performed, printing out the message:

```
X=Y
```

Again, everything from ELSE on is ignored, and the program proceeds to the next line.

The final possibility is for X to be smaller than Y. Let's assume we enter the following values:

```
X = 2
Y = 6
```

Now, when the computer gets to line 50, the first test (X > Y) turns out false, so the computer skips ahead until it finds the ELSE command. This is the second test (X = Y) which is also false. The computer again skips ahead through the line until it finds the next ELSE. This tells it to display the message:

```
Y IS LARGER THAN X
```

Whenever you program a computer to make a decision, you *must* account for all possibilities, or the program may not give the expected (or desired) results. This is what happened with the utilities company computer that didn't know it was supposed to ignore a balance due of $0.00.

BOOLEAN TESTS

Don't let the term "Boolean" scare you off. It is really fairly simple. The mathematician that this type of operation was named after just happened to have a rather intimidating name. In a Boolean system, test data can be combined in various ways. There are three basic Boolean operators.

The simplest Boolean operator is NOT. This just reverses the test. For example:

```
IF NOT X=0
```

will register true if X does NOT equal 0. In other words, it is exactly the same as:

```
IF X < > 0
```

In many programs it may be necessary to test multiple pieces of data together. For example, let's say we want the program to jump to line 100 only of both X and Y have values greater than 4, otherwise, the program should continue with the next line.

We could use two IF . . . THEN . . . tests:

```
IF X > 4 THEN
  IF Y > 4 THEN GOTO 100
```

A more elegant solution would be to use the Boolean operator AND:

```
IF X > 4 AND Y > 4 THEN GOTO 100
```

The two command forms will have exactly the same result. The GOTO 100 will be executed if and only if both test conditions are true. If either (or both) of the test conditions are false, the "GOTO 100" will be ignored.

The third Boolean operator is OR. This one too is pretty much self-explanatory. For example:

```
IF X > 4 OR Y > 4 THEN GOTO 100
```

The GOTO 100 will be executed if either (or both) of the test conditions are true. That is, all of the following combinations will result in a program jump to line 100:

```
X=5      Y=7
X=9      Y=1
X=2      Y=6
```

Boolean tests are made up of these three operators, NOT, AND, and OR, in various combinations. Boolean operations are summarized in truth tables. A true condition is represented by a 1, while a false condition is represented by a 0. The truth table identifies the output conditions (test results) for all possible input combinations.

There are two possibilities for a NOT operation. If the test condition is true (1),

the output condition will be false (0), or vice versa. The truth table for a NOT operation is as follows:

INPUT	OUTPUT
0	1
1	0

AND and OR operations have at least two input tests. The AND truth table for two inputs, is as follows:

INPUTS	OUTPUT
A B	
0 0	0
0 1	0
1 0	0
1 1	1

The AND operation can be expanded for three or more inputs (simultaneous tests). Just remember the output will be true (1) if and only if all input tests are true (1).

The two input OR truth table looks like this:

INPUTS	OUTPUT
A B	
0 0	0
0 1	1
1 0	1
1 1	1

The OR operation can be expanded for three or more simultaneous input tests. The operation will register as true (1) if at least one of the test conditions is true (1).

Boolean operations can be combined. For example,

IF NOT (A AND B) THEN . . .

INPUTS	OUTPUT
A B	
0 0	1
0 1	1
1 0	1
1 1	0

The NOT operator reverses the AND operation.

Another combined Boolean operation can take this format:

IF NOT A AND B THEN . . .

INPUTS	OUTPUT
A B	
0 0	0
0 1	1
1 0	0
1 1	0

Only the A test is NOTted.

More complex combinations are also possible. Such as:

IF NOT A AND (B OR C) THEN . . .

INPUTS	OUTPUT
A B C	
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	0
1 1 1	0

The operation(s) in parentheses are performed first.

An additional variation on the basic OR operation is the Exclusive-OR, or XOR operation. The output is true if one, and only one, of the input tests are true. The truth table looks like this:

INPUTS	OUTPUT
A B	
0 0	0
0 1	1
1 0	1
1 1	0

The operation can be simulated using the standard operators:

IF (A OR B) AND (NOT(A AND B)) THEN . . .

Obviously, the XOR function simplifies matters:

```
IF A XOR B THEN . . .
```

Boolean operations may look a little confusing at first, but after you play with them for awhile, they should become like second nature. Using Boolean operators wisely can greatly simplify computer programming.

OPTION WEIGHTING

In AI programs more complex decisions will often have to be made. The real world often refuses to cooperate with the computer's simplistic black/white approach. The real world often is in shades of gray. A number of different factors have to be considered, and they may not all be of equal importance.

The solution is option weighting. Each relevant factor is assigned a weighting value. Then each option is rated for each individual factor. The option with the highest total is selected as the best choice.

A simple program using this approach is given in Listing 2-1. This program was written to be as non-specific as possible.

The user first determines the number of options to be considered, and assigns a name to each in lines 50 through 150.

The next step is to determine the number of factors, and their labels (lines 160 through 270). The computer next clears an array (CH (x)) that will be used for each option's weighting totals (lines 280 through 310). The weighting for each factor is entered in lines 320 through 400. Weighting values from 1 to 10 (inclusive) are accepted. Fractional values (such as 4.5) are fine. Now the program goes into a pair of nested loops (lines 410 through 520) that permits the user to rate each option for each individual factor.

The rating is simply multiplied by the appropriate weighting factor and added to that options total. Next comes a short sorting routine (lines 530 through 610) to arrange the options in descending order, based on the option totals stored in array CH(x).

Finally, the options are displayed in descending order. The best option is redisplayed again at the end for convenience. If many options are used, the best option might scroll off the top of the screen, and this is presumedly the option we are most interested in.

The variables used in this program are summarized in Table 2-1. The results of a sample run are given in Table 2-2. A career change is being considered, but the program indicates that this user should probably stick with his present job.

You certainly shouldn't take the results from this (and any other similar program) as gospel. It is simply intended as a guide to make decision-making a little more convenient.

DEFINING PROBLEMS

Any problem can be broken up into three components:

STATE(s)—the initial or current conditions of the factors involved in the problem

A	Number of options
AA	A-1
B	Number of factors
C	User inputs
X	Counting loops
Z	Sorting marker
A$	Temporary string for sorting sequence
B$	Temporary string for sorting sequence
	ARRAYS
CH(A)	Calculated choice values
FR(B)	Factor ratings
FC$(B)	Factor names
OP$(A)	Option names

Table 2-1. Variables Used in the OPTION WEIGHER Program of Listing 2-1.

Table 2-2. A Sample Run of the OPTION WEIGHER Program.

```
How many options?< 5
Option    #  1   ?>Continue present job
Option    #  2   ?>Join Schwartz & Co.
Option    #  3   ?>Join NOIP productions
Option    #  4   ?>Send out resumes
Option    #  5   ?>Start own business

How many factors to be weighed?> 8
Factor    #  1   ?>Chance of success
Factor    #  2   ?>Relocation?
Factor    #  3   ?>Income
Factor    #  4   ?>Short-term security
Factor    #  5   ?>Long-term security
Factor    #  6   ?>Independence
Factor    #  7   ?>Fringe benefits
Factor    #  8   ?>Social contacts

Rank each factor from 1 to 10
Chance of success?> 8
Relocation??> 6
Income?> 7.5
Short-term security?> 6
Long-term security?> 9
Independence?> 7
Fringe benefits?> 7
Social contacts?> 5

Rate each factor from 1 to 10 for Option # 1
Continue present job
Chance of success?> 8
Relocation??> 10
Income?> 5
Short-term security?> 9
Long-term security?> 4
Independence?> 4
Fringe benefits?> 7
Social contacts?> 9

Rate each factor from 1 to 10 for Option # 2
Join Schwartz & Co.
Chance of success?> 6
Relocation??> 7
Income?> 7
Short-term security?> 7
Long-term security?> 6
Independence?> 6
Fringe benefits?> 7.5
Social contacts?> 7

Rate each factor from 1 to 10 for Option # 3
Join NOIP productions
Chance of success?> 3
Relocation??> 5
Income?> 8.5
Short-term security?> 7
Long-term security?> 7
Independence?> 5
Fringe benefits?> 7.5
Social contacts?> 5

Rate each factor from 1 to 10 for Option # 4
Send out resumes
Chance of success?> 5
Relocation??> 6
Income?> 7
Short-term security?> 3
Long-term security?> 7
Independence?> 5
Fringe benefits?> 5
Social contacts?> 5

Rate each factor from 1 to 10 for Option # 5
Start own business
Chance of success?> 6.5
Relocation??> 1
Income?> 5
Short-term security?> 3
Long-term security?> 8
Independence?> 9
Fringe benefits?> 3
Social contacts?> 7

1—Continue present job      373.5
2—Join Schwartz & Co.       368
3—Join NOIP productions     335.25
4—Send out resumes          304.5
5—Start own business        304.5

Best choice—Continue present job 373.5
```

OPERATION(s)—the action or actions that might be taken to change the STATE(s).
GOAL—the desired end result or final STATE

The initial state and the goal are given. The problem solver must decide which operations must be performed to reach the goal from the given state.

Let's consider a simple example. Imagine a small room, as shown in Fig. 2-1. In this room we have two tables (A and B), a blue box, a red box, and a robot. The robot's programming allows it to do four things (operations):

pick up a box
put down a box
move from table A to table B
move from table B to table A

Our initial state has the robot beside table B. The blue box is on table A. The red box is on top of the blue box. Our desired goal is to have the robot beside table A, the red box on table B, and the blue box on table A. What operations need to be performed here?

Several solutions are possible for this problem (this is true of most problems). Probably the most efficient solution would be to have the robot perform the following sequence of operations:

move from table B to table A
pick up red box
put red box down on floor
pick up blue box
move from table A to table B
put blue box down on table B
move from table B to table A

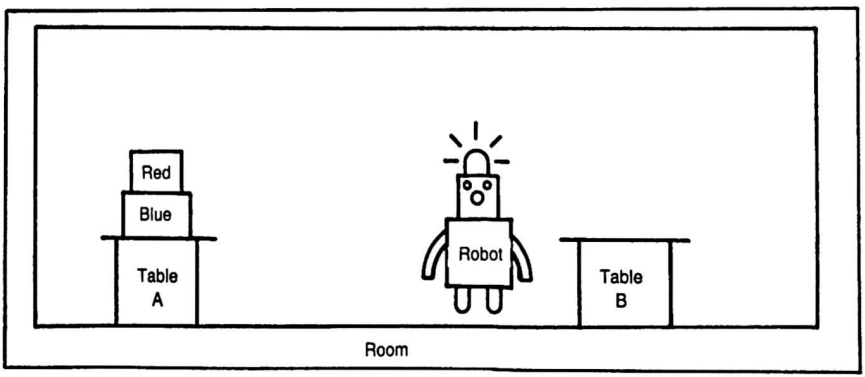


Fig. 2-1. Manipulating the objects demonstrates the steps involved in solving a problem.

pick up red box
put red box down on table A

We have now reached our goal state and the problem has been solved.

You can see that even this simple task can be broken down into a series of very simple steps. For a complex task, the number of simple steps can become extremely large. Working your way through them can be tedious at best. It would be very easy to make a mistake. Computers work very well in tedious, repetitious tasks, if we can figure out a way to tell them what we want done.

For each operation performed in solving a problem, a number of decisions have to be made:

What is the current state?
What operations are possible?
What is the probable result of each potential operation?
Will the operation bring us closer to the goal?

The last question can be especially tricky. Sometimes we need to move away from the goal to reach it. In our simple box moving example, the red box starts out on table A, which is where we want it to end up. But it must be moved in order to get at the blue box. To determine the results of an operation, we need to think about future operations and their probable results too.

In some cases, it may be convenient to break the problem up into sub-goals that have to be reached on the way to the main goal. Our robot first has to solve the problem of how to get the blue box (sub-goal) before it can achieve its main goal which involves moving the blue box to the other table.

There is another limitation on operations—will the result be legal, or will it negate the goal. For an example of this, let's consider the old logic puzzle of the farmer who needs to transport a bushel of corn, a chicken and a fox across a river. (I never have figured out what he was doing with that fox.) His boat is small so he can only take one item across at a time.

If he leaves the chicken alone with the corn, the chicken will eat the corn. Similarly, he can't leave the chicken unattended with the fox, or when he gets back he'll find a fat fox, and perhaps a few feathers. These are illegal states. From the initial state there are four possible operations:

- ☆ Farmer crosses river with an empty boat—this would bring us no closer to the goal, and would result in an illegal state—the chicken will eat the corn if the fox doesn't eat the chicken first
- ☆ Farmer crosses river with the corn—this would result in an illegal state—the fox will eat the chicken
- ☆ Farmer crosses the river with the chicken—this leaves the fox alone with the corn, which is perfectly acceptable
- ☆ Farmer crosses the river with the fox—this would result in an illegal state—the chicken will eat the corn

There are four possible operations, but three of them would result in the desired goal (all three items on the other side of the river) becoming impossible. So, in this instance, there is only one acceptable operation—the farmer must take the chicken across the river on the first trip.

How could a computer deal with this problem. Actually, this particular problem lends itself very well to the yes/no type of decisions a computer is good at. We will just set up variables to represent the four relevant items:

F = farmer
B = bushel of corn
C = chicken
X = fox

There are two acceptable values for each variable. We will arbitrarily declare that a 0 means the item is on the original bank of the river, and a 1 means that object is on the destination bank. Initially, all four variables have a value of 0.

Each operation requires the farmer to move from one bank to the other, so the variable F will change value each time. When he carries an item with him, the appropriate variable will change values.

The computer must test for illegal states each time it considers an operation. How can this be done? Think about the illegal states. If the corn (B) and the chicken (C) are on the same bank (B = C) and the farmer is on the opposite bank (C = NOT F), we have an illegal state. The same is true if the chicken (C) and the fox (X) are on the opposite bank as the farmer F. We can sum the illegal states up as follows:

```
If ((B=C) OR (C=X)) AND (C= NOT F)
  Then it is an illegal state
```

NOT F, of course, has the opposite value of F.

The computer could try every possible operation on each move and simply check to see if it would result in an illegal state. It could simply bumble through acceptable operations until it reaches the goal. This inelegant, brute-force approach would probably work eventually for this simple problem, but for more complex problems it would be too inefficient. Suppose there are 100 possible operations on each move, and the solution would take at least 500 moves, with 25 variables to check for 125 illegal states. Even at computer speeds, the wasted time of the brute force method could add up.

Even for our simple problem of the farmer, the brute force method leaves something to be desired. With the constraints we've given the computer so far, it might come up with this solution:

```
FARMER takes CHICKEN to new bank
FARMER takes CHICKEN to old bank
FARMER takes CHICKEN to new bank
FARMER goes to old bank
FARMER takes CORN to new bank
```

FARMER takes CHICKEN to old bank
FARMER takes CHICKEN to new bank
FARMER takes CORN to old bank
FARMER goes to new bank
FARMER takes CHICKEN to old bank
FARMER takes CHICKEN to new bank
FARMER goes to old bank
FARMER takes FOX to new bank
FARMER takes CHICKEN to old bank
FARMER takes CORN to new bank
FARMER takes FOX to old bank
FARMER takes FOX to new bank
FARMER goes to old bank
FARMER goes to new bank
FARMER goes to old bank
FARMER takes CHICKEN to new bank

Yes, we finally stumbled onto the desired goal, but there was a lot of effort involved. This is hardly an intelligent solution by anyone's definition.

The computer must also check each operation to make sure that it isn't a step backwards away from the goal. If nothing else, an operation should not simply undo the results of the immediately preceding operation. That is, don't take the CHICKEN across the river, only to immediately bring it right back. That doesn't accomplish everything.

To solve this problem, it is necessary to do some back-tracking, and bring something from the new bank back to the old bank. The most efficient (correct) solution is:

FARMER takes CHICKEN to new bank
FARMER goes to old bank
FARMER takes CORN to new bank
FARMER takes CHICKEN back to old bank
FARMER takes FOX to new bank
FARMER goes to old bank
FARMER takes CHICKEN to new bank

This time the problem is solved in a minimum of steps. There are often multiple possible solutions to a problem, but we want the BEST one. Generally, this means the most efficient solution, or the solution with the lowest cost. We can consider that each operation costs something, if only the effort of performing the operation.

In many practical solutions, such as those involving decisions in building materials, different operations will have different costs. To find the best path to the goal, the computer must keep track of the cost of each solution. In some cases, the solution with the fewest operations may cost more than a longer solution with less expensive operations.

STATE GRAPHS

So far the problems we have discussed have been trivial. It would probably be

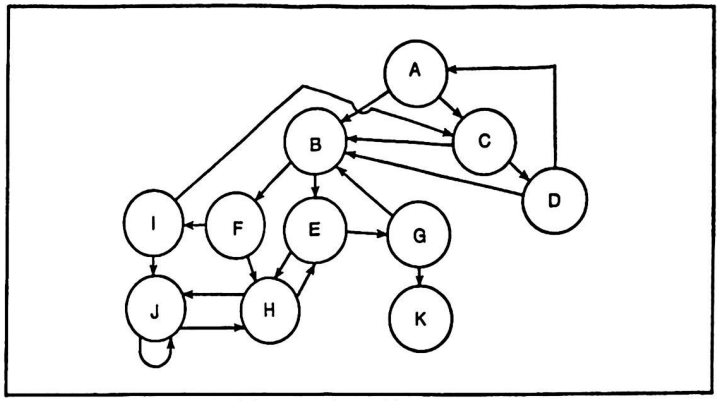


Fig. 2-2. A state graph can help define a complex problem.

wasteful to turn to a computer for a solution to simple problems like this. It might be easier to solve the problem yourself than to program a computer to solve it for you.

To help define more complex problems, it is often helpful to draw state graphs. This is simply an illustration that allows us to see the relationships between the states and operators, without concerning ourselves with their specific details.

A state graph is made up of two basic elements. These are circles called nodes (corresponding to states) and connecting lines called arcs (corresponding to operations). Small arrows are placed on the arcs to indicate the initial and resulting states for each operation.

A simple state graph is shown in Fig. 2-2. It is rather like a maze. We need to move from the initial state (A) to the goal (K). Notice that there is more than one arrow leading in and out of most nodes. Some nodes (C and J) have useless operations that only lead back to themselves.

The multiple arcs leading into each node mean there is more than one possible solution. Looking at the state graph, we can easily see that the most efficient path is:

A – B – E – G – K

In this particular example, two operations (arcs) are possible from each state (node), except the goal state (K), of course.

A simple program for letting a computer tackle this maze problem is given in Listing 2-2. The state graph is held in a two-dimensional array (A(x,y)). The first element (x) identifies the current state:

1 = A
2 = B
3 = C
4 = D
5 = E

6 = F
7 = G
8 = H
9 = I
10 = J
11 = K

The second element (y) identifies which of the two operations will be taken. The values stored in the array represent the results of the indicated operation. For example, from node A, we can move either to node B or node C, so:

A(1,1) = 2
A(1,2) = 3

The array is filled by READing values from the DATA statement.

A second array (A$(x)) simply holds the names of each node for convenience in printing out the results. This string array is loaded from the second data statement.

The variable N is used to keep track of the current node. At each node, the computer randomly chooses a value (Z) of either 1 or 2. (Note that the BASIC statement for selecting a random number varies for different microcomputers. Check your owner's manual.) The value of Z determines which of the two possible operations will be taken from each node.

A typical result from this program is shown in Table 2-3. The computer eventually stumbled upon the goal by pure chance. Choices were made randomly, with absolutely no intelligence involved. Notice how the computer often makes the same mistake over and over again. It never learns.

An improved version of this program is given in Listing 2-3. Here, the computer remembers what it has already done. Since we won't ever want to take the same operation from a given node more than once, the computer will replace the appropriate array value with 0. A new array (B(v)) is added to keep track of the steps made by the computer. If it runs into a node with both operation values set to 0, it backs up to the previous node until it finds a move it can make. The variable V keeps track of the appropriate location in array B(x).

A typical result from this program is shown in Table 2-4. It is still not perfect, but it does a lot better than the original blind luck version. This version of the program shows some slight (admittedly, very slight) intelligence. It at least has the "common sense" not to make the same mistakes over and over.

TREE SEARCHES

Another way to look at this problem is consider the original state node as the root

Table 2-3. A Sample Run of the MAZE Program of Listing 2-2.

A-C-C-C-D-B-F-I-J-H-J-H-J-H-E-H-J-H-J-H-J-H-E-G-B-F-I-C-D-A- C-C-D-B-E-H-E-G-B-E-H-J-H-J-H-J-H-J-J-J-H-J-J-J-H-J-J-J-H-J-H-J- H-E-H-J-H-E-G-B-F-I-C-D-A-C-C-C-D-B-E-G-K
DONE!

Table 2-4. Greater Efficiency can be Achieved with a Program that "learns" and Avoids Bad Moves.

A-B-E-H-J-J-H-E-G-B-F-I-J-C-D-B-A-C-C-H-K
,DONE!

of an upside-down tree. Each node branches off in two or more directions, as illustrated in Fig. 2-3. We can call this a repeated branching problem.

The tracing of the nodes and operations from the root in an effort to find the goal is called a tree search. A tree is a directed graph in which each node has only one predecessor, except the root, which has no predecessors. Note that a given state may be reached by more than one path, but it is shown as separate nodes in the tree.

The operations (connecting lines) between the nodes are called branches. Any node which has no successor (a dead-end, or the goal) is called a leaf.

A tree's nodes are divided into levels. The root is alone on level 0. Its immediate successors (or children) are on level 1. The successors of the level 1 nodes are on level 2, and so forth. The predecessor (or parent) of each node is on the preceding level.

A search tree is a graph of all paths that have been explored. It is rarely represented directly in the computer's memory, but it is a convenient method of graphing what the computer "knows" about solving the problem so far.

The simplest method of using a search tree is to test out each and every branch. That is, try all possible solutions until we find the one we want.

A simple tree search program is given in Listing 2-4. The tree is made up of four levels. Each node (except for the leaves in level 3) has two successors. Each node is assigned a random value.

In this program, the tree data is stored in a two-dimensional array—A(15,3). The second dimension is used to define the meaning of the stored value:

A(x,1) = value of node x

Table 2-5. Typical Results from the TREE SEARCH Program of Listing 2-4.

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

        1 - 2 - 4 -  8  =  27       ###  0
        1 - 2 - 4 -  9  =  25       ###  27
        1 - 2 - 5 - 10  =  19       ###  27
        1 - 2 - 5 - 11  =  24       ###  27
        1 - 3 - 6 - 12  =  29       ###  27
        1 - 3 - 6 - 13  =  26       ###  29
        1 - 3 - 7 - 14  =  26       ###  29
        1 - 3 - 7 - 15  =  23       ###  29

THE BEST TOTAL IS 29
PATH = = = 1 - 3 - 6 - 12
```

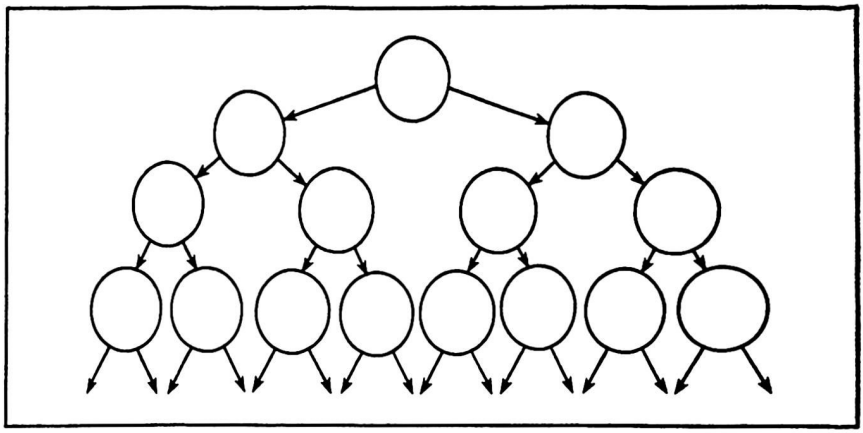


Fig. 2-3. The original state node can be considered the root of an upside down tree.

A(x,2) = child #1 of node x
A(x,3) = child #2 of node x

The nodes are numbered in straight numerical order from left to right:

1	LEVEL 0
2 3	LEVEL 1
4 5 6 7	LEVEL 2
8 9 A B C D E F	LEVEL 3

(A = 10, B = 11, C = 12, D = 13, E = 14, F = 15)

The goal of this program is to find the path down to level 3 with the lowest total value of nodes. (Random node values are assigned in lines 140 through 170.)

A couple of sample runs for this program are illustrated in Table 2-5. (The program was modified by changing all PRINT statements to LPRINT statements, and adding a line to LPRINT each path as it is tested.)

Since there are only eight possible paths, the computer can whip right through all the possible solutions using the brute force method in less than a second.

For more complex searches, the brute force method is too inefficient and awkward. To demonstrate this, an expanded version of this program is given in Listing 2-5. This time the tree goes down 9 levels (from 0 to 8) with 256 possible paths to be checked out. Try running this program, and notice how much longer it takes than the one in Listing 2-4. As the sample run shows in Table 2-6, the computer has to do a lot of useless number crunching.

Practical problems can be far worse. There may be more than 9 levels. (In some cases, there may be dozens, or even hundreds.) There may be more than two children for each node too.

Table 2-6. A "brute-force" tree Search can be very inefficient.

(1)	1 - 2 - 4 - 8 - 16 - 32 - 64 - 128 - 256	= = = 53	< 0 >
(2)	1 - 2 - 4 - 8 - 16 - 32 - 64 - 128 - 257	= = = 53	< 53 >
(3)	1 - 2 - 4 - 8 - 16 - 32 - 64 - 129 - 258	= = = 52	< 53 >
(4)	1 - 2 - 4 - 8 - 16 - 32 - 64 - 129 - 259	= = = 50	< 53 >
(5)	1 - 2 - 4 - 8 - 16 - 32 - 65 - 130 - 260	= = = 37	< 53 >
(6)	1 - 2 - 4 - 8 - 16 - 32 - 65 - 130 - 261	= = = 38	< 53 >
(7)	1 - 2 - 4 - 8 - 16 - 32 - 65 - 131 - 262	= = = 41	< 53 >
(8)	1 - 2 - 4 - 8 - 16 - 32 - 65 - 131 - 263	= = = 39	< 53 >
(9)	1 - 2 - 4 - 8 - 16 - 33 - 66 - 132 - 264	= = = 50	< 53 >
(10)	1 - 2 - 4 - 8 - 16 - 33 - 66 - 132 - 265	= = = 51	< 53 >
(11)	1 - 2 - 4 - 8 - 16 - 33 - 66 - 133 - 266	= = = 48	< 53 >
(12)	1 - 2 - 4 - 8 - 16 - 33 - 66 - 133 - 267	= = = 48	< 53 >
(13)	1 - 2 - 4 - 8 - 16 - 33 - 67 - 134 - 268	= = = 51	< 53 >
(14)	1 - 2 - 4 - 8 - 16 - 33 - 67 - 134 - 269	= = = 47	< 53 >
(15)	1 - 2 - 4 - 8 - 16 - 33 - 67 - 135 - 270	= = = 46	< 53 >
(16)	1 - 2 - 4 - 8 - 16 - 33 - 67 - 135 - 271	= = = 49	< 53 >
(17)	1 - 2 - 4 - 8 - 17 - 34 - 68 - 136 - 272	= = = 34	< 53 >
(18)	1 - 2 - 4 - 8 - 17 - 34 - 68 - 136 - 273	= = = 37	< 53 >
(19)	1 - 2 - 4 - 8 - 17 - 34 - 68 - 137 - 274	= = = 36	< 53 >
(20)	1 - 2 - 4 - 8 - 17 - 34 - 68 - 137 - 275	= = = 42	< 53 >

CONTINUED FOR 256 TOTAL LINES.

(248)	1 - 3 - 7 - 15 - 31 - 62 - 125 - 251 - 503	= = = 43	< 68 >
(249)	1 - 3 - 7 - 15 - 31 - 63 - 126 - 252 - 504	= = = 40	< 68 >
(250)	1 - 3 - 7 - 15 - 31 - 63 - 126 - 252 - 505	= = = 44	< 68 >
(251)	1 - 3 - 7 - 15 - 31 - 63 - 126 - 253 - 506	= = = 40	< 68 >
(252)	1 - 3 - 7 - 15 - 31 - 63 - 126 - 253 - 507	= = = 38	< 68 >
(253)	1 - 3 - 7 - 15 - 31 - 63 - 127 - 254 - 508	= = = 43	< 68 >
(254)	1 - 3 - 7 - 15 - 31 - 63 - 127 - 254 - 509	= = = 44	< 68 >
(255)	1 - 3 - 7 - 15 - 31 - 63 - 127 - 255 - 510	= = = 45	< 68 >
(256)	1 - 3 - 7 - 15 - 31 - 63 - 127 - 255 - 511	= = = 43	< 68 >

The best total is 68
Path --- 1 - 3 - 6 - 13 - 27 - 54 - 108 - 217 - 435

Obviously the brute force method leaves a lot to be desired for problems of any complexity. We need some way to "prune" the tree down to a more manageable size.

There are two basic approaches, depending on the specific requirements of the problem at hand:

depth-first search
breadth-first search

As we discuss and compare these two approaches, we will use the maze problem shown in Fig. 2-4. Most of the nodes here have three children. Some are dead ends (leaves with no children). A few nodes lead back to themselves becoming their own children. Each time a node is re-encountered it is considered a new node on the next level down the tree.

The initial state in this problem is A, and the goal state is N.

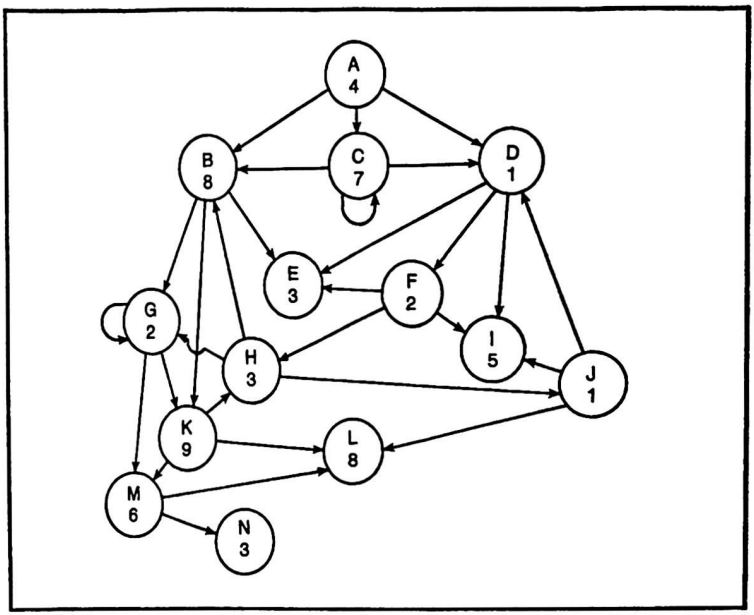


Fig. 2-4. This maze program will be used in the advanced tree search programs of Listing 2-6 and 2-7.

Summing up the nodes and their children we have:

PARENT	CHILDREN
A	B C D
B	E G K
C	B C D
D	E F I
E	none
E	none
F	E H I
G	G K M
H	B G J
I	none
J	D I L
K	H L M
L	none
M	J L N
N	none— GOAL state

Each node has a cost value assigned to it:

A	4
B	1
C	7
D	1
E	9
F	5
G	4
H	4
I	5
J	1
K	2
L	8
M	6
N	3

There are many possible paths to the goal (N). We want the most efficient path. Depending on the exact nature of the problem, the most efficient path may be the shortest (fewest nodes) or the cheapest (lowest total node cost).

DEPTH-FIRST SEARCH

If we want the cheapest possible path, we will use the depth-first type of search. This starts out like the simple brute-force search presented earlier. It starts out at the root, and keeps on working its way down until it reaches the goal. Then it starts at the root again and starts working its way back down. The difference is, if it discovers the current path it is trying has a higher cost than a previous path, it stops where it is and starts a new path. The abandoned path is rejected at mid-point.

The program is given in Listing 2-6. The array values are loaded from the DATA statements at the end of the program.

We will examine this program in some detail.

The main variables are identified in the opening REMarks (lines 10 through 130). Next, the arrays are DIMensioned (lines 140 through 150). In many versions of Microsoft BASIC, it is not necessary to DIMension arrays of ten or smaller. On some computers, it may be necessary to add the following statements:

```
145 DIM C(10)

155 DIM D(10)
```

The Node Name (A$(x)) and Node Value/Children (A(x,y)) are next loaded in lines 160 through 230. These arrays are filled with information from the DATA statements at the end of the program.

Array C(x) holds the best path found so far. At the beginning of the program, there is no best path, so we want this array to be empty. It is loaded with zeroes in lines 240 through 260.

In lines 270 through 310 some single variables are preset. C is the cost of the lowest path found so far. At the beginning of the program, there is no best path yet, so anything will be an improvement. We therefore preset the variable C to a high dummy value (999) that won't be exceeded by any possible path.

T counts the number of paths tried. We don't want the program to run indefinitely, so we will limit it to 100 paths. T starts out with a value of 0, since no paths have been tried yet.

P is the identifying number for the current node. Since each search begins with the root, P is preset with the value 1. Each new path to be explored begins at this line (290).

D is the cost of the current path. Since we haven't started exploring the path yet, this value should naturally be set to 0.

L is the current level reached in the tree. Since each path is explored to a maximum depth before moving on to the next path, we want to set a maximum level. Theoretically, any path will eventually reach the GOAL or a dead end, but it might take a great deal of time. We can assume the lowest cost path will be found within a specific range of levels, and just not bother searching past that point. In this program, we won't let the search go below level 10. Each search begins on level 1, so L = 1.

The current path array (D(x)) is next cleared to prevent confusion from portions of previous paths. This is done in lines 320 to 340.

The try counter (T) is incremented (line 350) and the current value is displayed (line 360).

Line 370 prints out the current node name. The PRINT statements during the search (lines 360, 370, 530, 560, and 600) may be eliminated if you prefer. If line 560 is removed, it must be replaced with a REM statement, or the program will crash. The program will run slightly faster without these commands. On the other hand, displaying the paths as they are explored will allow you to observe just how the program functions.

The current node number (P) is placed in the current path array (D(x)) in line 380, and the next line (390) adds the current node value (A(p,1)) to the current path cost (D).

Each node has up to three children. The children for each node are identified in A(P,x), where X = 2, 3, or 4. A dead end (no successor) is represented in the array as a zero. Several algorithms could be used to select the next operation. However, we don't want to get ahead of ourselves here. To keep this demonstration program simple, we simply select a random value (B) from 2 to 4 in line 400. The selected child is assigned to the variable E (line 410). If a dead end has been selected, the move is rejected, and the program jumps back to line 400 to try again.

Some node paths lead right back to themselves. Nodes C and G can be their own children in this problem. Obviously, moving from a node to the same node doesn't accomplish anything. When this situation is encountered (line 430), the children array is changed to show this path as a dead end (0) on all future attempts.

Next, lines 440 to 480 check the new node to determine if it has any available children. If all three children pointers are zeroes, the node is a dead end. When this happens, the program jumps to line 560. The path is labelled a dead end, and the preceding move that led to the dead end node is given a new value of 0, so it won't be selected again. The program then jumps back to line 290 to try a new path.

Notice that this program does show a certain minimal intelligence. It can "learn"

to avoid dead ends. It doesn't learn from all mistakes, however. Valid paths (no dead end) may be tried multiple times. Still, the odds are in the 100 permitted tries, the computer will find the desired (lowest cost) path to the GOAL.

If the selected new node is not a dead end, the program continues with line 490, where the new node is checked to see if it is the goal (E = 14). If so, the program jumps to line 590. We will cover that section of the program later.

If the goal has not been reached, the new node becomes the current node (P = E) in line 500, and we move on to the next level (line 510). If we have not gone past level 10 (line 520), the program loops around to line 370 to explore the next level.

If 10 levels have been explored, the search down that path is terminated. If 100 paths have been tried (line 540) the computer jumps to the end portion of the program (line 670), otherwise, we loop back around to line 290 to begin a new path.

If the goal had been found in line 490, the program jumps to line 590. The path is terminated, since there is no sense to continue searching once the goal has been reached. The current path cost (D) is compared to the lowest path cost (C) in line 610. If C is less than D, the program simply reverts to line 290 to try a new path.

If the current path has a lower cost than any previously tried path (C > D), the current path becomes the new lowest cost path. C is given the same value as D (line 620), and array C(x) is loaded with the current path from array D(x) (lines 630 through 650). Then we go back to line 290 to try a new path.

When 100 paths have been tested, the computer stops searching and goes to line 670. The lowest path found and its cost are displayed (lines 670 through 760), and the program ENDs (line 770). finally, lines 780 through 820 are the DATA statements for loading arrays A(x,y) and A$(x).

Of course this program has little practical value except as a demonstration, but the techniques used here can be applied to many practical situations, and will be used in many of the more complex programs presented later in this book.

A typical sample run for this program is shown in Table 2-7. Notice that many attempts are redundant. This was allowed to simplify the programming. Much of the wasted effort can be eliminated by adding the following line to the program:

```
395  IF D > C THEN PRINT:GOTO 290
```

This line tests the current path cost so far (D), with the lowest complete path cost found (C). If the current path cost is greater than a path already found, there is no sense in continuing with that path, since it won't be the one we want (the path to the GOAL with the lowest total cost). Therefore, when D is greater than C, this line sends the computer back to line 290 to start exploring a new path. This is much more efficient.

A sample run with this modification added to the program is shown in Table 2-8. Notice how many paths are now left incomplete. The program should also run slightly faster, since the computer is wasting time tracing expensive paths to their conclusion.

Before going further, it might be well worth your while to look over the program, and see if you can come up with any additional ways to make it run more efficiently.

BREADTH-FIRST SEARCH

Another type of tree search is the breadth-first search. Here, all nodes on each

Table 2-7. A Typical Sample Run for the DEPTH-FIRST TREE SEARCH Program of Listing 2-6.

```
 1 ) A-C-D-E— DEAD END
 2 ) A-B-E— DEAD END
 3 ) A-D-I— DEAD END
 4 ) A-D-F-H-G-M-N ** 21
 5 ) A-D-F-E— DEAD END
 6 ) A-B-K-L— DEAD END
 7 ) A-B-K-M-J-D-F-H-J-L— DEAD END
 8 ) A-C-C-D-F-H-J-D-F-I— DEAD END
 9 ) A-B-K-H-G-K-M-L— DEAD END
10 ) A-D-F-H-G-M-N ** 21
11 ) A-C-B-K-H-B-K-H-B-
12 ) A-D-F-H-J-I— DEAD END
13 ) A-B-K-H-B-G-K-H-G-
14 ) A-B-K-H-J-D-F-H-J-
15 ) A-B-K-H-J-D-F-H-B-
16 ) A-B-G-G-K-H-B-K-H-
17 ) A-D-F-H-B-K-H-G-K-
18 ) A-C-B-K-H-B-K-M-N ** 57
19 ) A-D-F-H-J-D-F-H-G-
20 ) A-C-B-K-H-B-G-K-M-
21 ) A-D-F-H-B-K-H-J-D-
22 ) A-B-K-M-J-D-F-H-B-
23 ) A-B-K-H-G-M-N ** 35
24 ) A-B-G-K-M-N ** 32
25 ) A-D-F-H-B-G-K-M-N ** 38
26 ) A-D-F-H-B-G-K-M-N ** 38
27 ) A-B-G-M-N ** 23
28 ) A-B-G-K-H-J-D-F-H-
29 ) A-D-F-H-J-D-F-H-J-
30 ) A-C-D-F-H-B-G-M-J-
31 ) A-C-D-F-H-B-K-M-N ** 43
32 ) A-C-D-F-H-B-K-M-J-
33 ) A-C-B-K-M-N ** 37
34 ) A-C-B-K-H-G-M-N ** 42
35 ) A-B-G-K-M-N ** 32
36 ) A-D-F-H-G-K-M-N ** 30
37 ) A-B-K-H-G-M-J-D-F-
38 ) A-C-B-G-K-H-J-D-F-
39 ) A-D-F-H-J-D-F-H-J-
40 ) A-D-F-H-B-G-M-N ** 29
41 ) A-C-D-F-H-B-K-M-N ** 43
42 ) A-C-B-K-M-N ** 37
43 ) A-D-F-H-B-G-K-H-B-
44 ) A-C-D-F-H-G-M-N ** 28
45 ) A-D-F-H-B-K-M-N ** 36
46 ) A-D-F-H-G-M-N ** 21
47 ) A-B-K-H-J-D-F-H-J-
48 ) A-D-F-H-G-M-N ** 21
49 ) A-D-F-H-B-G-M-J-D-
50 ) A-B-K-M-N ** 30
51 ) A-C-D-F-H-B-G-K-H-
52 ) A-B-G-M-N ** 23
53 ) A-D-F-H-G-K-M-N ** 30
54 ) A-D-F-H-J-D-F-H-B-
55 ) A-D-F-H-J-D-F-H-J-
56 ) A-D-F-H-B-G-K-M-J-
57 ) A-C-D-F-H-J-D-F-H-
58 ) A-B-G-K-M-J-D-F-H-
59 ) A-C-B-K-M-N ** 37
60 ) A-D-F-H-B-G-K-M-J-
61 ) A-B-G-M-J-D-F-H-B-
62 ) A-D-F-H-G-M-N ** 21
63 ) A-C-B-G-M-N ** 30
64 ) A-C-D-F-H-G-M-J-D-
65 ) A-D-F-H-J-D-F-H-G-
66 ) A-D-F-H-B-G-K-M-J-
67 ) A-B-G-M-J-D-F-H-G-
68 ) A-D-F-H-G-M-N ** 21
69 ) A-B-G-M-J-D-F-H-J-
70 ) A-C-B-K-M-N ** 37
71 ) A-B-G-M-N ** 23
72 ) A-C-D-F-H-J-D-F-H-
73 ) A-C-B-K-M-J-D-F-H-
74 ) A-C-B-K-H-J-D-F-H-
75 ) A-C-B-G-K-M-J-D-F-
76 ) A-B-K-H-G-M-N ** 35
77 ) A-B-K-H-G-M-N ** 35
78 ) A-B-G-K-H-B-G-M-N ** 45
79 ) A-B-K-H-B-G-M-J-D-
80 ) A-B-K-H-J-D-F-H-B-
81 ) A-B-G-K-M-J-D-F-H-
82 ) A-D-F-H-J-D-F-H-G-
83 ) A-C-B-G-K-M-N ** 39
84 ) A-C-D-F-H-B-K-M-N ** 43
85 ) A-D-F-H-G-M-N ** 21
86 ) A-D-F-H-G-K-M-N ** 30
87 ) A-D-F-H-B-K-H-J-D-
88 ) A-B-G-K-H-G-K-H-J-
89 ) A-D-F-H-G-M-N ** 21
90 ) A-D-F-H-B-G-K-M-N ** 38
91 ) A-B-G-K-H-B-G-M-J-
92 ) A-B-G-K-H-J-D-F-H-
93 ) A-D-F-H-G-M-J-D-F-
94 ) A-C-D-F-H-J-D-F-H-
95 ) A-D-F-H-G-M-J-D-F-
96 ) A-B-K-M-N ** 30
97 ) A-C-D-F-H-J-D-F-H-
98 ) A-B-G-M-J-D-F-H-B-
99 ) A-B-K-M-J-D-F-H-G-
100 ) A-D-F-H-J-D-F-H-G-
101 ) A-B-G-M-N ** 23
102 ) A-B-G-K-H-G-M-N ** 37
103 ) A-B-K-M-J-D-F-H-G-

Lowest path cost is 21
Path — A-D-F-H-G-M-N
```

level are examined before moving on down to the next level. This type of search will always find the shortest possible path to the goal. The cost may or may not be the lowest.

Listing 2-7 is a breadth-first program for the same maze problem used in the depth-first program. Because of the way this program is structured, it would not be practical

Table 2-8. A Sample Run when the Improvement is Added to the BREADTH-FIRST TREE SEARCH Program.

Presetting arrays
Calculating paths
Shortest path . . . N-M-G-B-A-
Cost is 23

to have it print out each path as it is tested. It could be done, but it would require some fairly extensive additional programming. Lines 10 through 80 are simply REMarks, primarily identifying the arrays used in the program. Lines 90 through 110 print out a message that the arrays are being loaded. All of the lines up to 110 may be eliminated without affecting the operation of the program.

The arrays are DIMensioned in lines 120 through 150. Array L's size is somewhat arbitrary. Since the computer will be using this array as a "scratch-pad" for its intermediate results, this array should be fairly large. I estimated that 250 should be more than large enough for the problem to be solved by this program.

Lines 160 to 240 load arrays A$(x) and A(x,y) from the data statements at the end of the program. This is the same as in the Depth-first program. Also, line 230 fills array P(x) with zeroes. Array L(x) is cleared in lines 250 through 270.

Lines 280 through 300 display another message—this time that paths are being calculated. Once again, these lines may be eliminated. They are included just to give you an idea of how far the program has progressed.

Three variables (P, Q, and X) are preset to a value of 1 in lines 310 through 330. P identifies the current node. Q keeps track of the end of array L(x). X is used to find the next node to be examined in array L(x).

In line 340, the current value of P(1) is stored in the first location of array L(x).

In lines 350 through 430, the three children of node P are examined. If the child value (Z) is a 0, the node is a dead end, and lines 380 through 420 are skipped. Next, the child node number is compared with the appropriate location in parent array P(x) to determine if it has been encountered before. If a non-zero value is found at P(Z), a shorter path to that node has already been located, so there is no point in exploring it further. The program jumps to line 430 to examine the next node.

If P(Z) = 0, then node Z is unexplored. Its immediate parent (P) is stored in P(Z) for future reference (line 390). The next step is to determine if the goal has been reached (Z = 14) in line 400. If so the program jumps to 490, otherwise it continues with line 410, in which Q is incremented. The new node (Z) is stored in L(Q) (line 420). The child examine loop ends in line 430.

Once all the children of node P have been examined, X is incremented (line 440). Line 450 sends the program to line 610 which prints out an error message if X exceeds the size of L(x). This should never occur, but it is a good idea to include such error trapping routines, just in case.

The current value of X selects the next node to be examined from array L(x) in line 460. If a 0 has somehow been stored in array L(x), the value is rejected, and the next node in line is picked. Program control now reverts back to line 350. This loop will continue indefinitely until the goal is found, or an error condition occurs.

If the goal is found (Z = 14), the program jumps to line 490. C (path cost) is set

to 0. The shortest path is printed out in reverse order (starting from the goal, and working back to the root). This is done to simplify programming.

The immediate parent, or predecessor of each node can be found, by looking up the child node's value in array P(x). Since the goal is node 14, its parent will be identified in P(14).

The cost of each node is added to C in line 540. When the complete path has been printed out up to the root (Z = 1), the total path cost is displayed (line 590), and the program ENDs.

The DATA statements in lines 630 through 670 are identical to the DATA statements in the depth-first program. Since this program will always generate the same final result (unless the DATA statements are changed), and nothing much is displayed during the calculating process, there isn't much point to including a sample run here.

MAKING THE CHOICE

Both depth-first and breadth-first searches can be thought of in terms of objects waiting in line.

In a depth-first search, the object at the end of the line is examined first. We might call this Last Come—First Served, or First In—Last Out.

In a breadth-first search, however, the objects that have waited in line the longest are examined first. This could be called First Come—First Served, or First In—First Out.

```
10  REM     2-1      OPTION WEIGHER
20  REM  * DELTON T. HORN *
30  REM  * COPYRIGHT 1986 *
50  HOME
60  PRINT
70  PRINT "HOW MANY OPTIONS";
80  INPUT A
90  DIM OP$(A)
100  DIM CH(A)
110  FOR X = 1 TO A
120  PRINT "OPION #";X;
130  INPUT A$
140 OP$(X) = A$
150  NEXT X
160  HOME
170  PRINT
180  PRINT "HOW MANY FACTORS TO BE WEIGHED";
190  INPUT B
200  DIM FC$(B)
210  DIM FR(B)
220  REM  * LOAD FACTORS
230  FOR X = 1 TO B
240  PRINT "FACTOR #";X
250  INPUT A$
260 FC$(X) = A$
270  NEXT X
280  HOME
290  FOR X = 1 TO A
300 CH(X) = 0
310  NEXT X
320  HOME
330  PRINT
340  PRINT "RANK EACH FACTOR FROM 1 TO 10"
350  FOR X = 1 TO B
360  PRINT FC$(X)
370  INPUT C
380  IF (C < 1) OR (C > 10) GOTO 370
390 FR(X) = C
400  NEXT X
410  FOR X = 1 TO A
420  HOME
440  PRINT "RATE EACH FACTOR FROM 1 TO 10 FOR","OPTION #";X
450  PRINT OP$(X)
460  FOR Y = 1 TO B
470  PRINT FC$(Y)
480  INPUT C
490  IF (C < 1) OR (C > 10) GOTO 480
500 CH(X) = CH(X) + (FR(Y) * C)
510  NEXT Y
520  NEXT X
530  HOME
```

```
540 Z = 0
550 AA = A - 1
560  FOR X = 1 TO AA
570 Q = CH(X):R = CH(X + 1)
580  IF Q < R THEN  GOSUB 680
590  NEXT X
600  PRINT : PRINT
610  IF Z > 0 THEN  GOTO 540
620  HOME
630  FOR X = 1 TO A
640  PRINT X;"--";OP$(X),CH(X)
650  NEXT X
660  PRINT "   BEST CHOICE -- ";OP$(1),CH(1)
670  END
680 CH(X) = R:CH(X + 1) = Q
690 A$ = OP$(X):B$ = OP$(X + 1)
700 OP$(X) = B$:OP$(X + 1) = A$
710 Z = Z + 1
720  PRINT "*";
730  RETURN
```

```
10  REM    2-2     1ST MAZE
20  REM  * DELTON T. HORN *
30  REM  * COPYRIGHT 1986 *
40  DIM A(11,2)
50  DIM A$(11)
60  FOR X = 1 TO 11
70  READ A,B
80 A(X,1) = A
90 A(X,2) = B
100  NEXT X
110  FOR X = 1 TO 11
120  READ A$
130 A$(X) = A$
140  NEXT X
150  HOME
160  PRINT : PRINT : PRINT
170  PRINT "  MAZE PROBLEM SOLVER"
180  PRINT "     VERSION 1"
190  PRINT : PRINT
200 X = 1
210  PRINT "A";
220 Z =  INT ( RND (1) * 2) + 1
230 X = A(X,Z)
240  PRINT "-";A$(X);
250  IF X = 11 GOTO 270
260  GOTO 220
270  PRINT : PRINT "DONE!"
275  END
280  DATA  2,3,5,6,4,3,1,2
290  DATA  7,8,8,9,11,2,5,10
300  DATA  3,10,10,8,11,11
310  DATA  A,B,C,D,E,F,G,H,I,J,K
```

```
10  REM    2-3     2ND MAZE
20  REM  * DELTON T. HORN *
30  REM  * COPYRIGHT 1986 *
40  DIM A(11,2)
50  DIM A$(11)
60  DIM B(25)
70  FOR X = 1 TO 11
80  READ A,B
90 A(X,1) = A
100 A(X,2) = B
110 B(X) = 0
120  NEXT X
130  FOR X = 1 TO 11
140  READ A$
150 A$(X) = A$
160  NEXT X
170  HOME
180  PRINT : PRINT : PRINT
190  PRINT "  MAZE PROBLEM SOLVER"
200  PRINT "     VERSION 2"
210  PRINT : PRINT
220 V = 1
230 X = 1
240  PRINT "A";
250 V = V + 1
260 W = A(X,1) + A(X,2)
270  IF W = 0 THEN V = V - 1:X = B(V): GOTO 260
280 Z =  INT ( RND (1) * 2 + 1)
290 Y = A(X,Z)
300  IF Y = 0 GOTO 430
310 B(V) = X
320 A(X,Z) = 0
330 X = Y
340  PRINT "-";A$(X);
350  IF X = 11 GOTO 370
360  GOTO 250
370  PRINT : PRINT "DONE!"
380  END
390  DATA  2,3,5,6,4,3,1,2
400  DATA  7,8,8,9,11,2,5,10
410  DATA  3,10,10,8,11,11
420  DATA  A,B,C,D,E,F,G,H,I,J,K
430  IF Z = 1 THEN Z = 2 ELSEZ = 1
440  GOTO 260
```

```
10  REM     2-4     SIMPLE TREE SEARCH
20  REM  * DELTON T. HORN *
30  REM  * COPYRIGHT 1986 *
40  DIM A(15,3)
50 Y = 2
60  FOR X = 1 TO 15
70 A(X,2) = Y
80 Y = Y + 1
90 A(X,3) = Y
100 Y = Y + 1
110  NEXT X
120  FOR X = 1 TO 15
130 Y =  INT ( RND (1) * 9 + 1)
140 A(X,1) = Y
150  NEXT X
160  HOME
170  PRINT : PRINT : PRINT
180  FOR X = 1 TO 12
190  PRINT " ";
200  NEXT X
210  PRINT A(1,1)
220  PRINT
230  FOR X = 1 TO 6
240  PRINT " ";: NEXT X
250  PRINT A(2,1);
260  FOR X = 1 TO 10
270  PRINT " ";: NEXT X
280  PRINT A(3,1)
290  PRINT
300  PRINT "  ";A(4,1);"     ";A(5,1);"     ";A(6,1);"     ";A(7,1)
310  PRINT
320  FOR X = 8 TO 15
330  PRINT A(X,1);"  ";
340  NEXT X
350  PRINT : PRINT
360 B = 0
370  FOR X1 = 2 TO 3
380  FOR X2 = 2 TO 3
390  FOR X3 = 2 TO 3
400 D = A(1,X1)
410 E = A(D,X2)
420 F = A(E,X3)
430 C = A(1,1) + A(D,1) + A(E,1) + A(F,1)
440  PRINT " 1 - ";D;" - ";E;" - ";F;" = ";C;" ### ";B
450  IF C > B THEN  GOSUB 530
460  FOR T = 1 TO 555: NEXT T
470  NEXT X3
480  NEXT X2
490  NEXT X1
495  PRINT
500  PRINT "    THE BEST TOTAL IS ";B
```

```
510  PRINT "PATH === 1 - ";G;" - ";H;" - ";I
520  END
530 B = C
540 G = D
550 H = E
560 I = F
570  RETURN
```

```
10  REM    2-5     LARGE TREE SEARCH
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM A(511,3)
50 Y = 2
60  FOR X = 1 TO 511
70  PRINT "Z";
80 A(X,2) = Y
90 Y = Y + 1
100 A(X,3) = Y
110 Y = Y + 1
120  NEXT X
130  FOR X = 1 TO 511
140  PRINT "*";
150 Y =  INT ( RND (1) * 9 + 1)
160 A(X,1) = Y
170  NEXT X
180  HOME
190  PRINT : PRINT
200  PRINT "PRESS <RETURN>  "
205  INPUT Q$
210  PRINT : PRINT
220 B = 0
230 Q = 1
240  FOR X1 = 2 TO 3
250  FOR X2 = 2 TO 3
260  FOR X3 = 2 TO 3
270  FOR X4 = 2 TO 3
280  FOR X5 = 2 TO 3
290  FOR X6 = 2 TO 3
300  FOR X7 = 2 TO 3
310  FOR X8 = 2 TO 3
320 D = A(1,X1)
330 E = A(D,X2)
340 F = A(E,X3)
350 G = A(F,X4)
360 H = A(G,X5)
370 I = A(H,X6)
380 J = A(I,X7)
390 K = A(J,X8)
400 C = A(1,1) + A(D,1) + A(E,1) + A(F,1) + A(G,1) + A(H,1) + A(I,1)
410 C = C + A(J,1) + A(K,1)
420  PRINT "(";Q;")  1 - ";D;" - ";E;" - ";F;" - ";G;" - ";H;" -
     ";I;" - ";J;" - ";K;" === "C," ","<";B;">"
430  PRINT
440 Q = Q + 1
450  IF C > B THEN  GOSUB 580
460  NEXT X8
470  NEXT X7
480  NEXT X6
490  NEXT X5
```

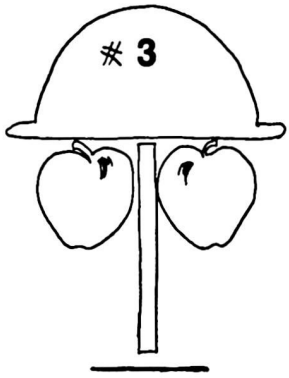
```
500  NEXT X4
510  NEXT X3
520  NEXT X2
530  NEXT X1
540  PRINT : PRINT
550  PRINT "THE BEST TOTAL IS ";B
560  PRINT "PATH ===  1 - ";L;" - ";M;" - ";N;" - ";O;" - ";P;" -
     ";R;" - ";S;" - ";T
570  END
580 B = C
590 L = D
600 M = E
610 N = F
620 O = G
630 P = H
640 R = I
650 S = J
660 T = K
670  RETURN
```

```
10  REM   2-6     DEPTH-FIRST SEARCH
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  REM  * A$(X)= NODE NAMES *
50  REM  * A(X,1) = CURRENT NODE VALUE *
60  REM  * A(X,2),A(X,3),A(X,4)= CHILDREN *
70  REM  * C(X)= LOWEST COST PATH SO FAR *
80  REM  * D(X)= CURRENT PATH *
90  REM  * C= LOWEST COST *
100  REM  * D= COST OF CURRENT    PATH *
110  REM  * T = # OF PATHS TRIED *
120  REM  * L = CURRENT LEVEL *
130  REM  * P = CURRENT NODE *
140  DIM A(14,4)
145  DIM D(10)
150  DIM A$(14)
155  DIM C(10)
160  FOR X = 1 TO 14
170  FOR Y = 1 TO 4
180  READ Z
190 A(X,Y) = Z
200  NEXT Y
210  READ Z$
220 A$(X) = Z$
230  NEXT X
240  FOR X = 1 TO 10
250 C(X) = 0
260  NEXT X
270 C = 999
280 T = 0
290 P = 1
300 D = 0
310 L = 1
320  FOR X = 1 TO 10
330 D(X) = 0
340  NEXT X
350 T = T + 1
360  PRINT T;") ";
370  PRINT A$(P);"-";
380 D(L) = P
390 D = D + A(P,1)
400 B =  INT ( RND (1) * 3 + 2)
410 E = A(P,B)
420  IF E = 0 GOTO 400
430  IF E = P THEN A(P,B) = 0
440 G = 0
450  FOR X = 2 TO 4
460 G = G + A(E,X)
470  NEXT X
480  IF G = 0 GOTO 560
490  IF E = 14 GOTO 590
```

```
500 P = E
510 L = L + 1
520  IF L < 10 GOTO 370
530  PRINT
540  IF T > 100 GOTO 670
550  GOTO 290
560  PRINT A$(E);"-- DEAD END"
570 A(P,B) = 0
580  GOTO 290
590 D = D + A(E,1)
600  PRINT "N ** ";D
610  IF C < D GOTO 290
620 C = D
630  FOR X = 1 TO 10
640 C(X) = D(X)
650  NEXT X
660  GOTO 290
670  PRINT : PRINT : PRINT
680  PRINT "LOWEST PATH COST IS ";C
690  PRINT : PRINT "PATH -- A";
700  FOR X = 2 TO 10
710 E = C(X)
720  IF E = 0 GOTO 740
730  PRINT "-";A$(E);
740  NEXT X
750  PRINT "-N"
760  PRINT : PRINT
770  END
780  DATA  4,2,3,4,A,8,5,7,11,B,7,2,3,4,C
790  DATA  1,5,6,9,D,3,0,0,0,E,2,5,8,9,F
800  DATA  2,7,11,13,G,3,2,7,10,H,5,0,0,0,I
810  DATA  1,4,9,12,J,9,8,12,13,K,8,0,0,0,L
820  DATA  6,10,12,14,M,3,99,99,99,N
```

```
10  REM   2-7     BREADTH-FIRST SEARCH
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  REM  * A$(X)= NODE NAMES *
50  REM  * A(X,1)= CURRENT NODE  VALUE *
60  REM  * A(X,2),A(X,3),A(X,4)= CHILDREN *
70  REM  * P(X)= PARENTS *
80  REM  * L(X)= NODES ON CURRENT LEVEL *
90  HOME
100  PRINT : PRINT
110  PRINT "    PRESETTING ARRAYS"
120  DIM A(14,4)
130  DIM A$(14)
140  DIM P(14)
150  DIM L(250)
160  FOR X = 1 TO 14
170  FOR Y = 1 TO 4
180  READ Z
190 A(X,Y) = Z
200  NEXT Y
210  READ Z$
220 A$(X) = Z$
230 P(X) = 0
240  NEXT X
250  FOR X = 1 TO 250
260 L(X) = 0
270  NEXT X
280  HOME
290  PRINT : PRINT
300  PRINT "     CALCULATING PATHS"
310 P = 1
320 Q = 1
330 X = 1
340 L(X) = P
350  FOR Y = 2 TO 4
360 Z = A(P,Y)
370  IF Z = 0 GOTO 430
380  IF P(Z) > 0 GOTO 430
390 P(Z) = P
400  IF Z = 14 GOTO 490
410 Q = Q + 1
420 L(Q) = Z
430  NEXT Y
440 X = X + 1
450  IF X > 250 GOTO 610
460 P = L(X)
470  IF P = 0 GOTO 440
480  GOTO 350
490 C = 0
500  HOME
510  PRINT : PRINT
```

```
520  PRINT "SHORTEST PATH"
530  PRINT A$(Z);"-";
540 C = C + A(Z,1)
550  IF Z = 1 GOTO 580
560 Z = P(Z)
570  GOTO 530
580  PRINT : PRINT
590  PRINT "COST IS ";C
600  END
610  PRINT : PRINT "ERROR!"
620  STOP
630  DATA  4,2,3,4,A,8,5,7,11,B,7,2,3,4,C
640  DATA  1,5,6,9,D,3,0,0,0,E,2,5,8,9,F
650  DATA  2,7,11,13,G,3,2,7,10,H,5,0,0,0,I
660  DATA  1,4,,12,J,9,8,12,13,K,8,0,0,0,L
670  DATA  6,10,12,14,M,3,99,99,99,N
```



Introduction to Game Theory

Many AI programs are built around games. This is not just because the games are fun (and potentially profitable to market). The very nature of games makes them well-suited to AI work.

Games offer clear-cut goals (winning is well-defined and unambiguous). Procedures are based on well-defined rules. Everything is clear-cut, and without ambiguity. That's just the way the computer likes things. Other real-world problems may not be nearly as convenient for AI purposes.

This is not meant to imply that all games are easy to program for AI—not by a long shot. But it is relatively easy to define just what we want the computer to do at each point in the program. We can easily work up a flow-chart, which gives us an excellent start on writing the program itself.

In this chapter we will be working with some relatively simple, but interesting games.

KALA

Some years ago I learned about an ancient game called Kala. I am reconstructing the game from memory here, so I may have inadvertently altered a few of the rules. No matter. My version of Kala is still fairly interesting, and a natural for AI programming.

The game is played on a board (or the ground) with fourteen shallow holes. There are two rows of six pits each. At the edges, between the rows are two additional pits, which are somewhat larger than the others. It is easier to show the game board than to describe it. The game board for Kala is illustrated in Fig. 3-1.

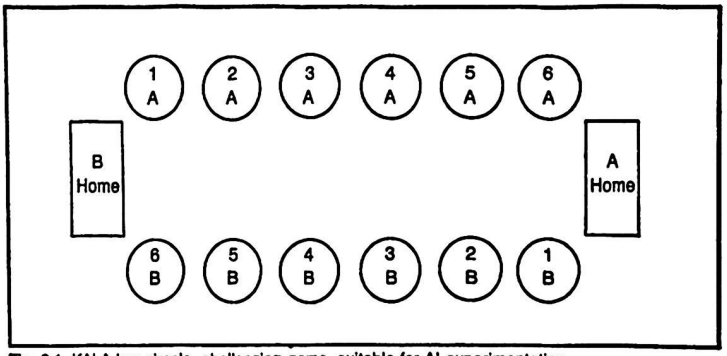


Fig. 3-1. KALA is a simple, challenging game, suitable for AI experimentation.

One player is called A, and the other is called B. A has control of the six pits labelled 1A through 6A. Similarly, 1B through 6B are controlled by player B. Each player also has a Home pit, which is used for scoring.

Moves are always made in a clock-wise direction. At the beginning of the game, a specific number of pebbles are placed in each of the numbered pits (not the Home pits). The same number of pebbles are placed in each pit. The starting number can range from three to six pebbles per pit. Using more pebbles results in a longer game requiring more strategy.

The object of the game for each player is to get as many pebbles as possible into his Home pit. Once a pebble has been placed in a home pit, it may not be moved again.

Each move is made by removing all of the pebbles from one of the player's controlled pits, and dropping them one at a time in each successive pit. For example, let's say pit 4A has been selected, and it contains five pebbles. One pebble will be placed in each of the following pits:

5A
6A
HOME A
1B
2B

Notice that some of the pebbles are passed over to the opponent's controlled area. Pit 4A will now be empty.

If the last pebble goes into the current player's Home pit, the player may take another turn.

The game ends when one player's controlled pits are all empty. Whichever player has the most pebbles in his Home wins the game. There is no special benefit in being the first to run out of possible moves.

Kala is conceptually a very simple game, but it can be quite challenging to play. There is more strategy involved than you might suspect.

Before programming a game for AI, it is usually a good idea to first work up a manual version of the program, in which both sides are manually played (via user inputs). It is often difficult to represent a game in computer terms. If you're trying to do the game representation at the same time you're trying to program AI, you're just begging for problems. Do one thing at a time. Make sure it works. Then move on to the next step. If the game program won't work properly in the manual mode, it won't work in an AI version either.

There is very little wasted effort involved in converting a manual version of a game program into an AI version. Usually a subroutine will be used to accept and analyze user moves. Only a few additional lines need to be added to permit moves to be entered for both players. Then, when converting to AI, just eliminate the handful of lines that

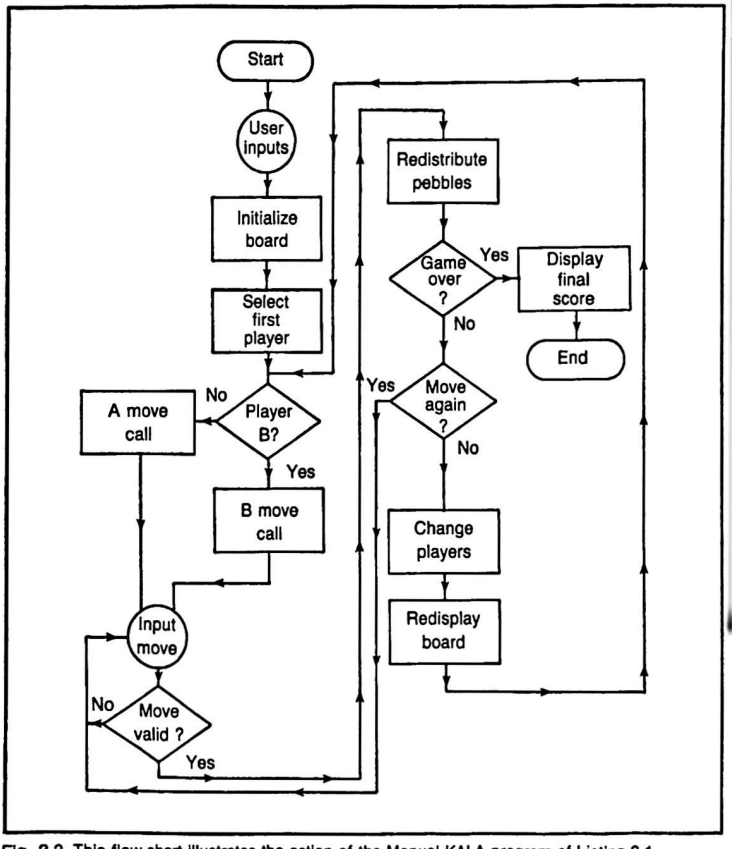


Fig. 3-2. This flow-chart illustrates the action of the Manual KALA program of Listing 3-1.

Table 3-1. Component Routines of the Manual KALA Program of Listing 3-1.

40	DIMension board array
50 - 160	User inputs
170 - 220	Initialize board
230	Call board display routine
240 - 260	Select first player
270 - 310	Player A move call routine
320 - 360	Player B move call routine
370 - 660	Display board subroutine
670 - 790	Move
800 - 860	End game check
870 - 890	Play again check
900 - 930	A plays again
940 - 970	B plays again
980 - 1000	Invalid move message
1010 - 1100	End game/final score display

handle the second player's inputs, and add whatever programming is needed for the computer to generate its own moves.

A flow-chart for a manual version of Kala is shown in Fig. 3-2. The program is given in Listing 3-1. Notice from the flow-chart that the same programming is used for each player's move, for the most part. Not many lines will have to be eliminated when we convert to the AI version.

Table 3-1 breaks the program down into its component routines. The variables used in this program are summarized in Table 3-2. Throughout this book I have tried to avoid the use of graphics to maximize portability of the programs to other personal computers. (Graphics capabilities and commands are extremely non-standardized.) For Kala, a graphics display of the game board probably wouldn't be particularly helpful. It is more convenient to simply display the number of pebbles in each pit, along with identifying labels for each pit. Fig. 3-3 shows the game board as displayed by this program. "x" is the appropriate number to indicate how many pebbles are currently in each pit.

There is no fancy programming involved here. If you look at each routine individually, you should have no problem following what is going on throughout this program.

Table 3-2. Variables used in the Manual KALA Program.

C	Initial number of pebbles in each bin
M	Move
P	Number of pebbles for the selected move
T	Current turn
X	Counting loops
Y	End game check
Z	End game check
A$	Player #1
B$	Player #2
	Array
BN(14)	Game board

```
   1  2  3  4  5  6
   x  x  x  x  x  x
B                    A
x                    x
   6  5  4  3  2  1
   x  x  x  x  x  x
```

Fig. 3-3. The game board in the KALA program of Listing 3-1 is displayed like this.

Once we get the manual program written and fully debugged, we are ready to start adding some AI, so the computer will play for player B.

What does the computer need to do. It selects one of its six controlled bins. This could be done with a random number generator it selects a value from 1 to 6. But the computer would not play very well. We certainly wouldn't call it intelligent, no matter how hard we try to stretch the definition.

Not only will the computer play with no attempt at strategy at all, it may even select invalid moves, by picking an empty pit. We should be able to do better than that.

To program some strategy, we need to consider the objectives of the game. The computer should want to get as many pebbles as possible into Home pit B.

One way to maximize the pebbles in Home pit B is for player B to get as many turns as possible. Remember, if the last pebble lands in the player's Home pit, that player gets an extra turn. Therefore, we will program the computer to first check for moves that will give it an extra turn.

Player B's (the computer's) pits are represented in the array as BN(8) through BN(13). Home pit B is BN(14).

By adding the array position number to the number of pebbles in that pit, we can determine the pit the last pebble will fall in. For example, let's say that there are three pebbles in B's pit 2 (BN(9)). The last pebble should end up in array position 9 + 3—BN(12). This is B's pit 5. We can easily confirm this, by watching what happens to each of the three pebbles.

The first pebble goes into B's pit 3 (BN(10)). The second goes into B's pit 4 (BN(11)). The third and last pebble lands in B's pit 5 (BN(12)), just as we predicted.

It is very easy for the computer to add the array number to the pebbles held in that bin. If the total is 14, the computer will get to take another turn. The routine is as follows:

```
1060  M=0
1070  FOR X=8 TO 13
1080  IF X + BN(X)=14 THEN M=X
1090  NEXT X
1100  IF M > 0 GOTO 1150
```

```
1150  M = M-7
1160  PRINT "I PLAY ";M
```

M is initially set to zero (line 1060). If a suitable move is found, M is given the value of the appropriate array number. The array numbers for B's pits are 7 higher than the pit numbers, so we subtract 7 from M to get the desired move (line 1150).

Suppose no appropriate move is found in this test. M retains a value of zero, so the test in line 1100 comes out false. Instead of jumping ahead to line 1150, program control drops down to an alternative move selection routine beginning at line 1110:

```
1110  M=13
1120  IF BN(M) > 0 GOTO 1150
1130  M=M-1
1140  GOTO 1120
```

The computer starts with bin 6 (BN(13)). If there are no pebbles in this pit, 1 is subtracted from M. Pit 5 (BN(12)) is now checked. This continues until a pit containing at least one pebble is found. This is the move the computer will make.

By starting with pit 6, we know we'll at least drop a pebble into the Home pit. This will also clear pit 6 to zero. The next higher move with enough pebbles will drop one pebble into pit 6. This means on its next turn, the computer will get to make at least two moves.

Notice also that the checking routine in lines 1060 through 1100 will select the highest pit number that will permit an extra turn. This is desirable, because any other repeat moves won't be disturbed. Consider the following possible arrangement:

PIT #	6 5 4 3 2 1
pebbles	1 2 5 4 3 7

There are three moves which will allow the player to take an additional turn:

6
5
3

If we play 3 first, we will be left with this arrangement:

PIT #	6 5 4 3 2 1
pebbles	2 3 6 0 3 7

The other repeat moves are obliterated. By starting with the highest numbered pit and working back, we will get the following sequence:

PIT #	6 5 4 3 2 1
pebbles	1 2 5 4 3 7

MOVE — 6

PIT #	6 5 4 3 2 1
pebbles	0 2 5 4 3 7

MOVE — 5

PIT #	6 5 4 3 2 1
pebbles	1 0 5 4 3 7

MOVE — 6

PIT #	6 5 4 3 2 1
pebbles	0 0 5 4 3 7

MOVE — 3

PIT #	6 5 4 3 2 1
pebbles	1 1 6 0 3 7

MOVE — 6

PIT #	6 5 4 3 2 1
pebbles	0 1 6 0 3 7

We've dropped five pebbles into the Home pit, and can still make one more move.

This simple strategy will allow the computer to play a fair game of Kala. It is not perfect. The computer will occasionally waste time inching a single pebble along, pit by pit. For example, if the arrangement of the pebbles is:

PIT #	6 5 4 3 2 1
pebbles	0 0 0 1 7 4

The computer will move the single pebble from pit 3 to pit 4, even though playing 2 would drop a pebble into the Home pit.

I will leave it as an exercise for the reader to improve the move choice programming.

An advanced Kala player will also consider the effects of his play on the opponents. If there are more pebbles to carry the move past the player's Home pit, the excess pebbles will land in the opponent's pits. This may help or hinder the opponent, depending on the current arrangement of pebbles in his pits. The complete listing for the AI version of Kala is given in Listing 3-2.

FRUSTRATION

One of the first game programs I ever wrote has enjoyed quite a bit of popularity among my friends. I started out with a solo version, and soon expanded the game for two players. It wasn't long before I decided to automate the game. It was my first foray into the world of artificial intelligence. We will look at several different versions of the basic program.

```
0 X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
```

Fig. 3-4. FRUSTRATION is a simple, but surprisingly intriguing original game.

The game is played on a 10 by 10 grid (100 spaces). A marker is moved one to six spaces from left to right on each turn. Each space is assigned a secret value, which may be positive (adding to the score) or negative (lowering the score). Some spaces conceal booby traps, which send the player back to the starting position, and reduce the score by half. (This could be helpful, if the score is negative).

When the end of a row is reached, the marker loops around to the beginning of the next row. The game board for Frustration is shown in Fig. 3-4. The 0 represents the player's marker. All other spaces are indicated by Xs.

At first, the game is one of pure chance. But each time you hit a booby trap and get sent back to space 1, you must try to remember which spaces were good (high positive values), and which were bad (negative values and booby traps).

The routines that make up the solo version of Frustration are summarized in Table 3-3. The variables used in the program are outlined in Table 3-4. The complete program is given in Listing 3-3.

Frustration is a simple game, and a simple program, but it can be surprisingly challenging, and quite addictive. Still, any solitaire game can wear rather thin after awhile. Testing your wits against an opponent (human or computer) can help keep things interesting.

Before jumping to an AI version of Frustration, we should adapt the solo version of the program to permit two manual players. The program is modified for two players in Listing 3-4. Notice that the required changes are quite minor.

The rules of the game must be adapted slightly to permit a second player. The first player moves from position 1 (upper left corner) to position 100 (lower right cor-

Table 3-3. The Component Routines of Manual FRUSTRATION of Listing 3-3.

10	-	90	Initialize
100	-	150	Hide point values
160	-	180	Preset player position
190	-	260	Hide booby traps
270	-	390	Display grid board
400	-	530	Input and analyze move
540	-	580	Update score
590	-	600	Empty space message
610	-	650	Booby trap
670	-	690	Invalid move message
700	-	720	End game

D	Difficulty level
M	Move
P	Player position
Q	Display value
S	Score
W	Current space value
X	Counting loops
Y	Counting loops
Z	Counting loops
Q$	Dummy variable
	Array
A(100)	Grid board

Table 3-4. Variables Used in the Manual FRUSTRATION Program.

ner), as in the solo version of the game. The second player moves in the opposite direction—from position 100 to position 1. The game ends when either player runs off the grid.

If one of the players lands on a space occupied by the other player's marker, the other player will be sent back to his starting position. Neither player's score will be affected by this move. The point value of the space will be ignored for the attacking player. Sometimes it will be to your advantage to send your opponent back. Other times, it may be better strategy to just go past his marker.

The board display for the two player version of Frustration is shown in Fig. 3-5. The first player's marker is the 0, and the second player's marker is the 8.

Now, how would we go about programming a computerized opponent? The simplest approach is given in Listing 3-5. This is not true artificial intelligence. The computer simply chooses a random number from 1 to 6. The choice is weighted slightly, so that a move of 6 is the most likely move. The move choice routine is as follows;

```
850 PRINT"MY TURN":PRINT"I MOVE — ";
860 Y= RND(200)+50
870 FOR X=1 TO Y
880 M=RND(7)
890 IF M=7 THEN M=6
900 NEXT X
910 PRINT M
```

```
0 X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X 8
```

Fig. 3-5. In the two player version of FRUSTRATION, the second player's marker is an '8'.

The loop just slows things down a little to give the illusion that the computer is thinking. The loop's size (length of the delay) is randomly determined in line 860 to offer a little more variety and naturalness to the effect.

At the start of a game the computer may or may not do well. As the game goes on, however, the human player will probably do noticeably better. This is because the computer never remembers what went before. It may keep landing on the same booby trap or high negative value repeatedly. The secret of success in the game of Frustration is a good memory. The computer player in this version of the program has no memory at all. It just plays by blind luck. That's certainly not very interesting.

A reasonable solution is offered in the version of the Frustration program given in Listing 3-6. In this version of the program, the computer player has a perfect memory. The player memory is made up of a new array, B(x). Initially, all locations in this array are given values of −0.5. No known space will ever have this value, so it indicates an unknown space.

Whenever either player lands on a new space, the computer memorizes the discovered value in array B(x). Once a space value has been memorized, it will not be forgotten during the course of the game.

The first twelve moves are made randomly, since the computer won't know enough space values to make a choice.

From the thirteenth move on, the computer will check each of the six spaces available to it in array B(x). I will choose the highest possible value. An unknown space (−0.5) will be selected over a negative point value, or a booby trap (−50). In a long game, the computer will become a very formidable opponent.

The advantage will be particularly apparent when a hard game is played. All of the versions of Frustration allow the player(s) to select the level of difficulty. Any value from 1 to 5 may be entered (including fractional values, such as 13.5). The difficulty level value determines the maximum number of booby traps to be hidden in the grid. The easiest level (1) will have no more than 5 booby traps. The most difficult level (5) can have up to 25 hidden booby traps. Obviously, the more booby traps there are, the more frequently the player(s) will be sent back, making the game longer, and generally more challenging.

The AI player in Listing 3-6 will tend to be most impressive in hard games, because it will get a chance to fill in more spaces in its memory array (B(x)).

In the present version of the program, the computer will make no direct attempt to send the human player back to position 1. However, if the human player is doing well, he will presumably go for the best available space when near the computer's marker. The computer will go for the highest point value, and may try to take the same space the player just landed on. The computer will appear to be using strategy in deciding whether or not to hit its opponent.

You might want to try experimenting with modifications with the computer's move choice routine. Can you make it play even more intelligently. As simple as this version of the program is, I find it tough to beat at difficulty levels above about 2, even if I try to keep notes on point values as they are discovered.

An AI program does not always have to be complex to give a good illusion of intelligence. Can we say these programs are truly intelligent? Probably not. But, on the other hand, it is difficult to say just why they aren't intelligent, when you consider

the results. These simple programs play reasonably formidable games of Kala and Frustration.

TIC-TAC-TOE

Games like Kala and Frustration are relatively easy to program for AI, because each move is selected to maximize the score. The game is simply won by the highest scores. Other games don't involve a score. Each move is just part of an entire attempt to reach a specific end goal.

Let's consider the game of Tic-Tac-Toe. This game is so simple it can quickly become boring, but it is useful for illustrating AI game-playing principles.

Tic-Tac-Toe, as I'm sure you know, is played on a simple 3 × 3 grid, as shown in Fig. 3-6. One player places Xs within the grid. The other places 0s. The winner is the first to place three of his markers in a row across, down, or diagonally.

There is no scoring in the game of Tic-Tac-Toe. Each move has value only insofar as it brings the player closer to the three in a row goal. The AI program has to plan ahead. It can't just play a move at a time, because there is no way to evaluate intermediate moves.

A manual Tic-Tac-Toe program for two players is given in Table 3-7. The routines used in this program are summarized in Table 3-5. Table 3-6 lists the variables.

The trickiest part of this program is representing the grid and winning combinations to the computer.

In this program, the grid is stored in a nine position array (A(9)). An empty space simply contains the appropriate space number (1 through nine). An X marker is represented by a value of 20. A value of 30 indicates an 0 marker.

All possible winning combinations are stored in the DATA statements of lines 790 through 860. After each move, the eight possible row combinations are checked. If all three spaces in a given row have an equal value, one of the players has won (line 600). Then it is just a matter of determining if the first space in the winning row contains an X (20), or an 0 (30) to declare the winner.

The same data statements are used by the computer to select its moves in the AI version of the program, which is given in Listing 3-8. The computer always plays 0.

The computer first searches through all of the winning rows to determine if it has two of the spaces filled. If the third space contains an X (20), the computer moves on

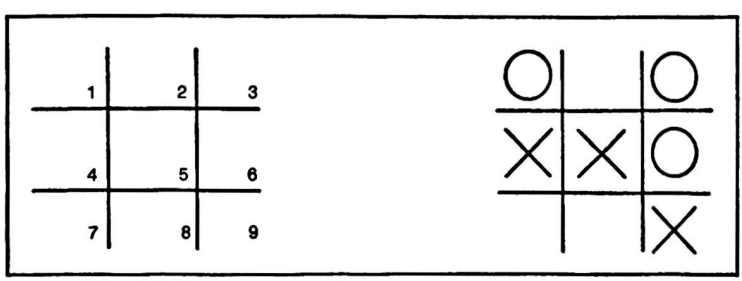


Fig. 3-6. TIC-TAC-TOE is played on a simple 3 × 3 grid.

10	-	80	Initialize grid
90	-	160	Name players/choose first turn
170	-	220	Player A's turn
230	-	290	Player B's turn
300	-	440	Display grid subroutine
450	-	530	Move subroutine
540	-	630	Check for winner subroutine
640	-	700	Check for stalemate subroutine
710	-	750	Winner end game
760	-	780	Tie end game
790	-	860	Winning row data

Table 3-5. The Component Routines of the Manual TIC-TAC-TOE Program of Listing 3-7.

Table 3-6. These Variables are Used in Manual TIC-TAC-TOE.

A	Row check
B	Row check
C	Row check
M	Move
T	Turn select
X	Counting loops
Y	Counting loops
Z	Counting loops/win test
A$	Player #1
B$	Player #2
	Array
A(9)	Grid

to the next row. If the computer finds a way to win during the current turn, it will make the appropriate move.

If it cannot complete three in a row this turn, it checks back through all eight winning rows to see if the opponent (X) has two in an unblocked row. If so, it will move to block the row.

If both these tests fail, the computer will check space number 5, and take it, if it is unfilled. This is the strongest position in the grid. Four of the eight possible winning rows use this position. A player who does not hold position 5, cannot complete a diagonal role.

If space 5 is already filled, the computer will select random numbers from one to nine until it finds an open space.

This simple four step selection process allows the computer to play a pretty good game of Tic-Tac-Toe. It is possible for the human player to beat the computer, but the win won't come easily.

The game programs presented in this chapter use fairly simple tricks to simulate intelligent play in fairly simple games. The level of AI in these programs is limited, at best.

In the next chapter we will explore some more advanced game playing techniques and discuss how a computer can be programmed to intelligently play more sophisticated games.

```
10  REM   3-1      MANUAL KALA
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM BN(14)
50  HOME
60  PRINT
70  PRINT "","KALA"
80  PRINT
90  PRINT "PLAYER #1";
100  INPUT A$
110  PRINT "PLAYER #2";
120  INPUT B$
130  PRINT "# OF PEBBLES PER BIN  (3-6)";
140  INPUT C
150 C =  INT (C)
160  IF C < 3 OR C > 6 GOTO 130
170  REM  * SET UP INITIAL BOARD *
180  FOR X = 1 TO 14
190 BN(X) = C
200  NEXT X
210 BN(7) = 0
220 BN(14) = 0
230  GOSUB 380
240  REM  * SELECT FIRST PLAYER *
250 T =  INT (1 +  RND (1) * 2)
260  IF T = 2 THEN  GOTO 310
270  REM  * THE PLAYS *
280  PRINT A$;" -- YOUR MOVE";
290  GOSUB 680
300  GOSUB 380
310 T =  - 1
320  PRINT B$;" -- YOUR MOVE";
330  GOSUB 680
340  GOSUB 380
350 T = 1
360  GOTO 280
370  REM  * DISPLAY *
380  HOME
390  PRINT : PRINT
400  FOR X = 1 TO 6
410  PRINT "  ";X;" ";
420  NEXT X
430  PRINT
440  FOR X = 1 TO 6
450  PRINT "  ";BN(X);" ";
460  NEXT X
470  PRINT
480  PRINT "B";
490  FOR X = 1 TO 25
500  PRINT " ";
510  NEXT X
```

```
520  PRINT "A"
530  PRINT BN(14);
540  FOR X = 1 TO 25
550  PRINT " ";
560  NEXT X
570  PRINT BN(7)
580  FOR X = 6 TO 1 STEP  - 1
590  PRINT "  ";X;" ";
600  NEXT X
610  PRINT
620  FOR X = 13 TO 8 STEP  - 1
630  PRINT "  ";BN(X);" ";
640  NEXT X
650  PRINT : PRINT
660  RETURN
670  REM  * MOVE *
680  INPUT M
690 M =  INT (M)
700  IF M < 1 OR M > 6 THEN  GOTO 680
710  IF T =  - 1 THEN M = M + 7
720 P = BN(M)
730 BN(M) = 0
740  IF P = 0 THEN  GOTO 980
750  FOR X = 1 TO P
760 M = M + 1
770  IF M > 14 THEN M = 1
780 BN(M) = BN(M) + 1
790  NEXT X
800  REM  * END GAME CHECK *
810 Y = 0:Z = 0
820  FOR X = 1 TO 6
830 Y = Y + BN(X)
840 Z = Z + BN(X + 7)
850  NEXT X
860  IF Y = 0 OR Z = 0 GOTO 1020
870  IF M = 7 AND T = 1 GOTO 900
880  IF M = 14 AND T =  - 1 GOTO 940
890  RETURN
900  PRINT A$;" PLAYS AGAIN"
910  GOSUB 390
920  PRINT "YOUR MOVE";
930  GOTO 680
940  PRINT B$;" PLAYS AGAIN"
950  GOSUB 390
960  PRINT "YOUR MOVE";
970  GOTO 680
980  PRINT "THERE ARE NO PEBBLES IN THAT BIN"
990  PRINT "TRY AGAIN"
1000  GOTO 680
1010  REM  * END GAME *
1020  GOSUB 380
```

```
1030  PRINT "GAME OVER"
1040  PRINT "      FINAL SCORE"
1050  PRINT A$,B$
1060 A = BN(7):B = BN(14)
1070  PRINT A,B
1080  PRINT
1090  IF A = B THEN  PRINT "TIE GAME"ELSE IF A > B THEN  PRINT A$;
     " WINS"ELSE PRINT B$;" WINS"
1100  END
```

```
10  REM   3-2     AI-KALA
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM BH(14)
50 M = 0
60  PRINT
70  PRINT "","KALA"
80  PRINT "","AI VERSION"
90  PRINT
100  PRINT "PLAYER'S NAME";
110  INPUT A$
120  PRINT "# OF PEBBLES PER BIN  (3-6)";
130  INPUT C
140 C =  INT (C)
150  IF C < 3 OR C > 6 GOTO 120
160  REM  * SET UP INITIAL BOARD *
170  FOR X = 1 TO 14
180 BH(X) = C
190  NEXT X
200 BH(7) = 0
210 BH(14) = 0
220  GOSUB 320
230  REM  * SELECT FIRST PLAYER *
240 T =  INT (1 +  RND (1) * 2)
250  IF T = 2 THEN  GOTO 1040
260  REM  * THE PLAYS *
270  PRINT A$;" -- YOUR MOVE";
280  GOSUB 620
290  GOSUB 320
300  GOTO 1040
310  REM  * DISPLAY *
320  HOME
330  PRINT : PRINT
340  FOR X = 1 TO 6
350  PRINT "  ";X;" ";
360  NEXT X
370  PRINT
380  FOR X = 1 TO 6
390  PRINT "  ";BH(X);" ";
400  NEXT X
410  PRINT
420  PRINT "B";
430  FOR X = 1 TO 25
440  PRINT " ";
450  NEXT X
460  PRINT "A"
470  PRINT BH(14);
480  FOR X = 1 TO 25
490  PRINT " ";
500  NEXT X
510  PRINT BH(7)
```

```
520  FOR X = 6 TO 1 STEP  - 1
530  PRINT "  ";X;" ";
540  NEXT X
550  PRINT
560  FOR X = 13 TO 8 STEP  - 1
570  PRINT "  ";BN(X);" ";
580  NEXT X
590  PRINT : PRINT
600  RETURN
610  REM  * MOVE *
620  INPUT M
630 M =  INT (M)
640  IF M < 1 OR M > 6 THEN  GOTO 620
650  IF T =  - 1 THEN M = M + 7
660 P = BN(M)
670 BN(M) = 0
680  IF P = 0 THEN  GOTO 910
690  FOR X = 1 TO P
700 M = M + 1
710  IF M > 14 THEN M = 1
720 BN(M) = BN(M) + 1
730  NEXT X
740  REM  * END GAME CHECK *
750 Y = 0:Z = 0
760  FOR X = 1 TO 6
770 Y = Y + BN(X)
780 Z = Z + BN(X + 7)
790  NEXT X
800  IF Y = 0 OR Z = 0 THEN  GOTO 950
810  IF M = 7 AND T = 1 THEN  GOTO 840
820  IF M = 14 AND T =  - 1 THEN  GOTO 880
830  RETURN
840  PRINT A$;" PLAYS AGAIN"
850  GOSUB 330
860  PRINT "YOUR MOVE";
870  GOTO 620
880  PRINT "I PLAY AGAIN"
890  GOSUB 330
900  GOTO 1050
910  PRINT "THERE ARE NO PEBBLES IN THAT BIN"
920  PRINT "TRY AGAIN"
930  GOTO 620
940  REM  * END GAME *
950  GOSUB 320
960  PRINT "GAME OVER"
970  PRINT "      FINAL SCORE"
980  PRINT A$,"ME"
990 A = BN(7):B = BN(14)
1000  PRINT A,B
1010  PRINT
1020  IF A = B THEN  PRINT "TIE GAME"ELSE IF A > B THEN  PRINT A$;
```

```
     " WINS"ELSE THEN  PRINT "I WIN"
1030  END
1040  PRINT "MY TURN"
1050 M = 0
1060 T =  - 1
1070  FOR X = 8 TO 13
1080  IF X + BN(X) = 14 THEN M = X
1090  NEXT X
1100  IF M > 0 THEN  GOTO 1150
1110 M = 13
1120  IF BN(M) > 0 THEN  GOTO 1150
1130 M = M - 1
1140  GOTO 1120
1150 M = M - 7
1160  PRINT "I PLAY ";M
1170  PRINT "PLEASE PRESS <RETURN>  "
1180  INPUT B$
1190  GOSUB 650
1200  GOSUB 320
1210 T = 1
1220  GOTO 270
```

```
10  REM   3-3      SOLO-FRUSTRATION
20  REM  * VERSION 1 -- SOLO
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM A(100)
60  HOME
70  PRINT : PRINT
80  PRINT "        FRUSTRATION"
90  PRINT : PRINT
100  REM  * HIDE POINTS *
110  FOR X = 3 TO 99
120 Y =  INT ( RND (1) * 27) - 9
130  IF Y > 9 THEN Y = 0
140 A(X) = Y
150  NEXT X
160  REM * PRESET PLAYER POSITION *
170 P = 1:A(1) = 25
180 S = 0:W = 0
190  REM  * PLANT BOOBY TRAPS *
200  PRINT "GAME DIFFICULTY  (1-5)  ";
210  INPUT D
220 D =  INT (D * 5)
230  FOR X = 1 TO D
240 Y =  INT ( RND (1) * 90) + 6
250 A(Y) =  - 50
260  NEXT X
270  HOME
280  PRINT
290 Z = 1
300  FOR X = 1 TO 10
310  PRINT "    ";
320  FOR Y = 1 TO 10
330 Q = A(Z)
340  IF Q = 25 THEN  PRINT "O ";
345  IF Q <  > 25 THEN  PRINT "X ";
350 Z = Z + 1
360  NEXT Y
370  PRINT
380  NEXT X
390  PRINT
400  PRINT "YOUR MOVE";
410  INPUT M
420 M =  INT (M)
430  IF M < 1 OR M > 6 THEN  GOTO 670
440 A(P) = W
450 P = P + M
460  IF P > 99 THEN  GOTO 700
470 W = A(P)
480 A(P) = 25
490  PRINT "YOU JUST FOUND ";
500  IF W =  - 50 THEN  GOTO 610
```

```
510  IF W = 0 THEN  GOTO 590
520  PRINT W;" POINT";
530  IF W = 1 OR W =  - 1 THEN  PRINT "!"
535  IF W > 1 OR W <  - 1 THEN  PRINT "S!"
540 S = S + W
550  PRINT "SCORE IS NOW ";S
560  PRINT "PLEASE PRESS  <RETURN>   ";
570  INPUT Q$
580  GOTO 270
590  PRINT "ABSOLUTELY NOTHING."
600  GOTO 550
610  PRINT "A BOOBY TRAP!"
620 S =  INT (S / 2)
630 A(P) = W:A(1) = 25
640 P = 1:W = 0
650  GOTO 550
660  REM  * ERROR *
670  PRINT "INVALID MOVE!"
680  PRINT "TRY AGAIN     ";
690  GOTO 410
700  PRINT "","GAME OVER"
710  PRINT "FINAL SCORE IS ";S
720  END
```

```
10  REM   3-4      DUO-FRUSTRATION
20  REM  * VERSION 2 -- TWO PLAYERS
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM A(100)
60  HOME
70  PRINT : PRINT
80  PRINT "        FRUSTRATION"
90  PRINT : PRINT
100  INPUT "PLAYER #1";A$
110  INPUT "PLAYER #2";B$
120  REM  * HIDE POINTS *
130  FOR X = 3 TO 99
140 Y =  INT ( RND (1) * 27) - 9
150  IF Y > 9 THEN Y = 0
160 A(X) = Y
170  NEXT X
180  REM  * PRESET PLAYER POSITIONS *
190 P = 1:A(1) = 25
200 S = 0:W = 0
210 R = 100:A(100) = 30
220 T = 0:V = 0
230 L =  INT ( RND (1) * 2) + 1
240  REM  * PLANT BOOBY TRAPS *
250  PRINT "GAME DIFFICULTY  (1 - 5)  ";
260  INPUT D
270  IF D < 1 OR D > 5 THEN  GOTO 250
280 D =  INT (D * 5)
290  FOR X = 1 TO D
300 Y =  INT ( RND (1) * 90) + 6
310 A(Y) =  - 50
320  NEXT X
330  HOME
340  PRINT
350 Z = 1
360  FOR X = 1 TO 10
370  PRINT "    ";
380  FOR Y = 1 TO 10
390 Q = A(Z)
400  IF Q = 25 THEN  PRINT "O ";
402  IF Q = 30 THEN  PRINT "8 ";
405  IF Q < 25 THEN  PRINT "X ";
410 Z = Z + 1
420  NEXT Y
430  PRINT
440  NEXT X
450  PRINT
460  IF L = 2 THEN  GOTO 850
470  PRINT A$;"'S TURN"
480  GOSUB 800
490 A(P) = W
```

```
500 P = P + M
510  IF P > 99 THEN  GOTO 750
520 W = A(P)
530 A(P) = 25
535  IF W > 1 OR W <  - 1 THEN  PRINT "S!"
540  PRINT "YOU JUST FOUND ";
550  IF W = 30 THEN  GOTO 1160
560  IF W =  - 50 THEN  GOTO 660
570  IF W = 0 THEN  GOTO 640
580  PRINT W;" POINT";
590  IF W = 1 OR W =  - 1 THEN  PRINT "!"
595  IF W > 1 OR W <  - 1 THEN  PRINT "S!"
600 S = S + W
610  GOSUB 1090
620 L = 2
630  GOTO 330
640  PRINT "ABSOLUTELY NOTHING."
650  GOTO 610
660  PRINT "A BOOBY TRAP!"
670 S =  INT (S / 2)
680 A(P) = W:A(1) = 25
690 P = 1:W = 0
700  GOTO 610
710  REM  * ERROR *
720  PRINT "INVALID MOVE!"
730  PRINT "TRY AGAIN  ";
740  GOTO 810
750  PRINT : PRINT "","GAME OVER"
760  PRINT "    FINAL SCORE"
770  PRINT A$,B$
780  PRINT S,T
790  END
800  PRINT "YOUR MOVE";
810  INPUT M
820 M =  INT (M)
830  IF M < 1 OR M > 6 THEN  GOTO 720
840  RETURN
850  PRINT B$;"'S TURN"
860  GOSUB 800
870 A(R) = V
880 R = R - M
890  IF R < 2 THEN  GOTO 750
900 V = A(R)
910 A(R) = 30
920  PRINT "YOU JUST FOUND ";
930  IF V = 25 THEN  GOTO 1220
940  IF V =  - 50 THEN  GOTO 1020
950  IF V = 0 THEN  GOTO 1070
960  PRINT V;" POINT";
970  IF V = 1 OR V =  - 1 THEN  PRINT "!"
975  IF V > 1 OR V <  - 1 THEN  PRINT "S!"
```

```
980 T = T + V
990  GOSUB 1090
1000 L = 1
1010  GOTO 330
1020  PRINT "A BOOBY TRAP!"
1030 T =  INT (T / 2)
1040 A(R) = V:A(100) = 30
1050 R = 100:V = 0
1060  GOTO 990
1070  PRINT "ABSOLUTELY NOTHING."
1080  GOTO 990
1090  PRINT "      CURRENT SCORE"
1100  PRINT A$,B$
1110  PRINT S,T
1120  PRINT
1130  PRINT "PLEASE PRESS <RETURN>   ";
1140  INPUT Q$
1150  RETURN
1160  PRINT B$;"!"
1170 A(R) = V:W = 0
1180 V = 0:R = 100
1190 A(100) = 30
1200 A(P) = 25
1210  GOTO 610
1220  PRINT A$;"!"
1230 A(P) = W:V = 0
1240 W = 0:P = 1
1250 A(1) = 25
1260 A(R) = 30
1270  GOTO 990
```

```
10  REM   3-5     DUMB-FRUSTRATION
20  REM  * VERSION 3 -- "DUMB" AI
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM A(100)
60  HOME
70  PRINT : PRINT
80  PRINT "     FRUSTRATION": PRINT "   'DUMB' AI VERSION"
90  PRINT : PRINT
100  INPUT "PLAYER'S NAME";A$
110 B$ = "ME"
120  REM  * HIDE POINTS *
130  FOR X = 3 TO 99
140 Y =  INT ( RND (1) * 27) - 9
150  IF Y > 9 THEN Y = 0
160 A(X) = Y
170  NEXT X
180  REM  * PRESET PLAYER POSITIONS *
190 P = 1:A(1) = 25
200 S = 0:W = 0
210 R = 100:A(100) = 30
220 T = 0:V = 0
230 L =  INT ( RND (1) * 2) + 1
240  REM  * PLANT BOOBY TRAPS *
250  PRINT "GAME DIFFICULTY  (1 - 5)  ";
260  INPUT D
270  IF D < 1 OR D > 5 THEN  GOTO 250
280 D =  INT (D * 5)
290  FOR X = 1 TO D
300 Y =  INT ( RND (1) * 90) + 6
310 A(Y) =  - 50
320  NEXT X
330  HOME
340  PRINT
350 Z = 1
360  FOR X = 1 TO 10
370  PRINT "    ";
380  FOR Y = 1 TO 10
390 Q = A(Z)
400  IF Q = 25 THEN  PRINT "O ";
402  IF Q = 30 THEN  PRINT "8 ";
405  IF Q < 25 THEN  PRINT "X ";
410 Z = Z + 1
420  NEXT Y
430  PRINT
440  NEXT X
450  PRINT
460  IF L = 2 THEN  GOTO 850
470  PRINT A$;"'S TURN"
480  GOSUB 800
490 A(P) = W
```

```
500 P = P + M
510  IF P > 99 THEN  GOTO 750
520 M = A(P)
530 A(P) = 25
540  PRINT "YOU JUST FOUND ";
550  IF M = 30 THEN  GOTO 1210
560  IF M =  - 50 THEN  GOTO 660
570  IF M = 0 THEN  GOTO 640
580  PRINT M;" POINT";
590  IF M = 1 OR M =  - 1 THEN  PRINT "!"
595  IF M > 1 OR M <  - 1 THEN  PRINT "S!"
600 S = S + M
610  GOSUB 1140
620 L = 2
630  GOTO 330
640  PRINT "ABSOLUTELY NOTHING."
650  GOTO 610
660  PRINT "A BOOBY TRAP!"
670 S =  INT (S / 2)
680 A(P) = M:A(1) = 25
690 P = 1:M = 0
700  GOTO 610
710  REM  * ERROR *
720  PRINT "INVALID MOVE!"
730  PRINT "TRY AGAIN  ";
740  GOTO 810
750  PRINT : PRINT "","GAME OVER"
760  PRINT "    FINAL SCORE"
770  PRINT A$,B$
780  PRINT S,T
790  END
800  PRINT "YOUR MOVE";
810  INPUT M
820 M =  INT (M)
830  IF M < 1 OR M > 6 THEN  GOTO 720
840  RETURN
850  PRINT "MY TURN": PRINT "I MOVE -- ";
860 Y =  INT ( RND (1) * 200) + 51
870  FOR X = 1 TO Y
880 M =  INT ( RND (1) * 7) + 1
890  IF M = 7 THEN M = 6
900  NEXT X
910  PRINT M
920 A(R) = V
930 R = R - M
940  IF R < 2 THEN  GOTO 750
950 V = A(R)
960 A(R) = 30
970  PRINT "I JUST FOUND ";
980  IF V = 25 THEN  GOTO 1270
990  IF V =  - 50 THEN  GOTO 1070
```

```
1000  IF V = 0 THEN  GOTO 1120
1010  PRINT V;" POINT";
1020  IF V = 1 OR V =  - 1 THEN  PRINT "!"
1025  IF V > 1 OR V <  - 1 THEN  PRINT "S!"
1030 T = T + V
1040  GOSUB 1140
1050 L = 1
1060  GOTO 330
1070  PRINT "A BOOBY TRAP!"
1080 T =  INT (T / 2)
1090 A(R) = V:A(100) = 30
1100 R = 100:V = 0
1110  GOTO 1040
1120  PRINT "ABSOLUTELY NOTHING."
1130  GOTO 1040
1140  PRINT "     CURRENT SCORE"
1150  PRINT A$,B$
1160  PRINT S,T
1170  PRINT
1180  PRINT "PLEASE PRESS <RETURN>  ";
1190  INPUT Q$
1200  RETURN
1210  PRINT B$;"!"
1220 A(R) = V:W = 0
1230 V = 0:R = 100
1240 A(100) = 30
1250 A(P) = 25
1260  GOTO 610
1270  PRINT A$;"!"
1280 A(P) = W:V = 0
1290 W = 0:P = 1
1300 A(1) = 25
1310 A(R) = 30
1320  GOTO 1040
```

```
10  REM   3-6      SMART-FRUSTRATION
20  REM  * VERSION 4 -- IMPROVED AI
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM A(100)
60  DIM B(100)
70  HOME
80  PRINT : PRINT
90  PRINT "       FRUSTRATION": PRINT "      AI VERSION"
100  PRINT : PRINT
110  INPUT "PLAYER'S NAME?  ";A$
120  FOR X = 1 TO 100
130 B(X) =  - .5
140  NEXT X
150 TC = 0
160 B$ = "ME"
170  REM  * HIDE POINTS *
180  FOR X = 3 TO 99
190 Y =  INT ( RND (1) * 27) - 9
200  IF Y > 9 THEN Y = 0
210 A(X) = Y
220  NEXT X
230  REM  * PRESET PLAYER'S POSITIONS *
240 P = 1:A(1) = 25
250 S = 0:W = 0
260 R = 100:A(100) = 30
270 T = 0:V = 0
280 L =  INT ( RND (1) * 2 + 1)
290  REM  * PLANT BOOBY TRAPS *
300  PRINT "GAME DIFFICULTY   (1-5)  ";
310  INPUT D
320  IF D < 1 OR D > 5 THEN  GOTO 300
330 D =  INT (D * 5)
340  FOR X = 1 TO D
350 Y =  INT ( RND (1) * 90) + 6
360 A(Y) =  - 50
370  NEXT X
380  HOME
390  PRINT
400 Z = 1
410  FOR X = 1 TO 10
420  PRINT "    ";
430  FOR Y = 1 TO 10
440 Q = A(Z)
450  IF Q = 25 THEN  PRINT "O ";
452  IF Q = 30 THEN  PRINT "8 ";
455  IF Q < 25 THEN  PRINT "X ";
460 Z = Z + 1
470  NEXT Y
480  PRINT
490  NEXT X
```

```
500  PRINT
510  IF L = 2 THEN  GOTO 910
520  PRINT A$;"'S TURN"
530  GOSUB 860
540 A(P) = M
550 P = P + M
560  IF P > 99 THEN  GOTO 810
570 M = A(P)
580  IF M < 25 THEN B(P) = M
590 A(P) = 25
600  PRINT "YOU JUST FOUND ";
610  IF M = 30 THEN  GOTO 1350
620  IF M =  - 50 THEN  GOTO 720
630  IF M = 0 THEN  GOTO 700
640  PRINT M;" POINT";
650  IF M = 1 OR M =  - 1 THEN  PRINT "!"
655  IF M > 1 OR M <  - 1 THEN  PRINT "S!"
660 S = S + M
670  GOSUB 1280
680 L = 2
690  GOTO 380
700  PRINT "ABSOLUTELY NOTHING."
710  GOTO 670
720  PRINT "A BOOBY TRAP!"
730 S =  INT (S / 2)
740 A(P) = M:A(1) = 25
750 P = 1:M = 0
760  GOTO 670
770  REM  * ERROR *
780  PRINT "INVALID MOVE!"
790  PRINT "   TRY AGAIN  ";
800  GOTO 870
810  PRINT : PRINT "","GAME OVER"
820  PRINT "       FINAL SCORE"
830  PRINT A$,B$
840  PRINT S,T
850  END
860  PRINT "YOUR MOVE ";
870  INPUT M
880 M =  INT (M)
890  IF M < 1 OR M > 6 THEN  GOTO 780
900  RETURN
910  PRINT "MY TURN": PRINT "I MOVE -- ";
920 Y =  INT ( RND (1) * 200) + 51
930  FOR X = 1 TO Y
940  NEXT X
950 TC = TC + 1
960  IF TC < 12 THEN M =  INT ( RND (1) * 6 + 1): GOTO 1040
970 M =  - 100
980  FOR X = 1 TO 6
990 Y = R - X
```

```
1000  IF Y < 1 THEN Y = 1
1010 Z = B(Y)
1020  IF Z > M THEN M = Z:M = X
1030  NEXT X
1040  PRINT M
1050 A(R) = V
1060 R = R - M
1070  IF R < 2 THEN  GOTO 810
1080 V = A(R)
1090  IF V < 25 THEN B(R) = V
1100 A(R) = 30
1110  PRINT "I JUST FOUND ";
1120  IF V = 25 THEN  GOTO 1410
1130  IF V =  - 50 THEN  GOTO 1210
1140  IF V = 0 THEN  GOTO 1260
1150  PRINT V;" POINT";
1160  IF V = 1 OR V =  - 1 THEN  PRINT "!"
1165  IF V > 1 OR V <  - 1 THEN  PRINT "S!"
1170 T = T + V
1180  GOSUB 1280
1190 L = 1
1200  GOTO 380
1210  PRINT "A BOOBY TRAP!"
1220 T =  INT (T / 2)
1230 A(R) = V:A(100) = 30
1240 R = 100:V = 0
1250  GOTO 1180
1260  PRINT "ABSOLUTELY NOTHING."
1270  GOTO 1180
1280  PRINT "      CURRENT SCORE"
1290  PRINT A$,B$
1300  PRINT S,T
1310  PRINT
1320  PRINT "PLEASE PRESS <RETURN>   ";
1330  INPUT Q$
1340  RETURN
1350  PRINT B$;"!"
1360 A(R) = V:M = 0
1370 V = 0:R = 100
1380 A(100) = 30
1390 A(P) = 25
1400  GOTO 670
1410  PRINT A$;"!"
1420 A(P) = M:V = 0
1430 M = 0:P = 1
1440 A(1) = 25
1450 A(R) = 30
1460  GOTO 1180
```

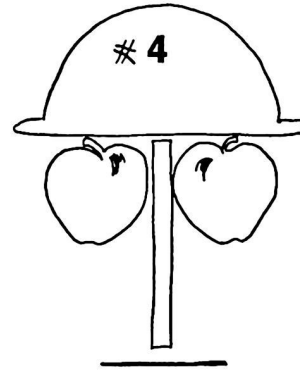
```
10  REM   3-7      MANUAL TIC-TAC-TOE
20  REM  * VERSION 1 -- MANUAL
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM A(9)
60  FOR X = 1 TO 9
70 A(X) = X
80  NEXT X
90  HOME
100  PRINT : PRINT
110  PRINT "PLAYER #1?";
120  INPUT A$
130  PRINT "PLAYER #2?";
140  INPUT B$
150 T =  INT (1 +  RND (1) * 2)
160  IF T = 1 THEN  GOTO 230
170  GOSUB 310
180  PRINT A$;"'S TURN"
190  PRINT "   <PLAYING X>"
200  GOSUB 460
210 A(M) = 20
220  GOSUB 550
230  GOSUB 310
240  PRINT B$;"'S TURN"
250  PRINT "    <PLAYING O>"
260  GOSUB 460
270 A(M) = 30
280  GOSUB 550
290  GOTO 170
300  REM  * DISPLAY BOARD *
310  HOME
320  PRINT : PRINT
330 Z = 1
340  FOR X = 1 TO 3
350  PRINT "",
360  FOR Y = 1 TO 3
370 A = A(Z)
380  IF A = 20 THEN  PRINT " X ";
382  IF A = 30 THEN  PRINT " O ";
385  IF A < 10 THEN  PRINT " ";A;" ";
390 Z = Z + 1
400  NEXT Y
410  PRINT
420  NEXT X
430  PRINT : PRINT
440  RETURN
450  REM  * MOVE *
460  PRINT "YOUR MOVE ";
470  INPUT M
480 M =  INT (M)
490  IF M < 1 OR M > 9 THEN  GOTO 520
```

```
500  IF A(M) > 9 THEN  GOTO 520
510  RETURN
520  PRINT "INVALID MOVE!"
530  GOTO 460
540  REM  * WIN CHECK *
550 Z = 0
560  FOR X = 1 TO 8
570  READ A
580  READ B
590  READ C
600  IF A(A) = A(B) AND A(A) = A(C) THEN Z = A
610  NEXT X
620  RESTORE
630  IF Z > 0 THEN  GOTO 720
640  REM  * STALEMATE CHECK *
650 Z = 0
660  FOR X = 1 TO 9
670  IF A(X) < 10 THEN Z = 1
680  NEXT X
690  IF Z < 1 THEN  GOTO 760
700  RETURN
710  REM  * END GAME *
720  IF A(Z) = 20 THEN  PRINT A$;
725  IF A(Z) = 30 THEN  PRINT B$;
730  PRINT " WINS!"
740  GOSUB 320
750  END
760  PRINT "TIE GAME!"
770  GOSUB 320
780  END
790  DATA  1,2,3
800  DATA  4,5,6
810  DATA  7,8,9
820  DATA  1,4,7
830  DATA  2,5,8
840  DATA  3,6,9
850  DATA  1,5,9
860  DATA  3,5,7
```

```
10  REM   3-8      AI TIC-TAC-TOE
20  REM  * VERSION 2 -- AI
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM A(9)
60  FOR X = 1 TO 9
70 A(X) = X
80  NEXT X
90  HOME
100  PRINT : PRINT
110  PRINT "PLAYER #1?";
120  INPUT A$
130  REM  * COMPUTER IS PLAYER #2 *
140 B$ = "MY"
150 T =  INT (1 +  RND (1) * 2)
160  IF T = 1 THEN  GOTO 230
170  GOSUB 310
180  PRINT A$;"'S TURN"
190  PRINT "   <PLAYING X>"
200  GOSUB 460
210 A(M) = 20
220  GOSUB 550
230  GOSUB 310
240  PRINT B$;" TURN"
250  PRINT "    <PLAYING O>"
260  GOSUB 890
270 A(M) = 30
280  GOSUB 550
290  GOTO 170
300  REM  * DISPLAY BOARD *
310  HOME
320  PRINT : PRINT
330 Z = 1
340  FOR X = 1 TO 3
350  PRINT "",
360  FOR Y = 1 TO 3
370 A = A(Z)
380  IF A = 20 THEN  PRINT " X ";
382  IF A = 30 THEN  PRINT " O ";
385  IF A < 10 THEN  PRINT " ";A;" ";
390 Z = Z + 1
400  NEXT Y
410  PRINT
420  NEXT X
430  PRINT : PRINT
440  RETURN
450  REM  * MOVE *
460  PRINT "YOUR MOVE ";
470  INPUT M
480 M =  INT (M)
490  IF M < 1 OR M > 9 THEN  GOTO 520
```

```
500  IF A(M) > 9 THEN  GOTO 520
510  RETURN
520  PRINT "INVALID MOVE!"
530  GOTO 460
540  REM  * WIN CHECK *
550 Z = 0
560  FOR X = 1 TO 8
570  READ A
580  READ B
590  READ C
600  IF A(A) = A(B) AND A(A) = A(C) THEN Z = A
610  NEXT X
620  RESTORE
630  IF Z > 0 THEN  GOTO 720
640  REM  * STALEMATE CHECK *
650 Z = 0
660  FOR X = 1 TO 9
670  IF A(X) < 10 THEN Z = 1
680  NEXT X
690  IF Z < 1 THEN  GOTO 760
700  RETURN
710  REM  * END GAME *
720  IF A(Z) = 20 THEN  PRINT A$;" WINS!"
730  IF A(Z) = 30 THEN  PRINT "I WIN!"
740  GOSUB 320
750  END
760  PRINT "TIE GAME!"
770  GOSUB 320
780  END
790  DATA  1,2,3
800  DATA  4,5,6
810  DATA  7,8,9
820  DATA  1,4,7
830  DATA  2,5,8
840  DATA  3,6,9
850  DATA  1,5,9
860  DATA  3,5,7
870  REM  * COMPUTER'S MOVE *
880  REM  * LOOK FOR WIN *
890  PRINT "I'LL PLAY -- ";
900 Z = 0
910  FOR X = 1 TO 8
920  READ A
930  READ B
940  READ C
950  IF A(A) = 30 OR A(B) = 30 THEN  GOTO 1190
960  NEXT X
970  RESTORE
980  IF Z > 0 THEN  GOTO 1120
990  REM  * LOOK FOR BLOCK *
1000  FOR X = 1 TO 8
```

```
1010  READ A
1020  READ B
1030  READ C
1040  IF A(A) = 20 OR A(B) = 20 THEN  GOTO 1250
1050  NEXT X
1060  RESTORE
1070  IF Z > 0 THEN  GOTO 1120
1080  REM  * RANDOM MOVE *
1090  IF A(5) = 5 THEN Z = 5: GOTO 1120
1100 Z =  INT (1 +  RND (1) * 9)
1110  IF A(Z) > 10 THEN  GOTO 1100
1120 M = Z
1130  PRINT M
1140  PRINT
1150  PRINT "PLEASE PRESS <RETURN>   ";
1160  INPUT Q$
1170  RETURN
1180  REM  * POSSIBLE WIN? *
1190  IF A(A) = 20 OR A(B) = 20 OR A(C) = 20 THEN  GOTO 960
1200  IF A(A) = A(B) THEN Z = C
1210  IF A(A) = A(C) THEN Z = B
1220  IF A(B) = A(C) THEN Z = A
1230  GOTO 960
1240  REM  * POSSIBLE LOSS? *
1250  IF A(A) = 30 OR A(B) = 30 OR A(C) = 30 THEN  GOTO 1050
1260  IF A(A) = A(B) THEN Z = C
1270  IF A(A) = A(C) THEN Z = B
1280  IF A(B) = A(C) THEN Z = A
1290  GOTO 1050
```



Advanced Games

In all of the simple games presented in the last chapter, only a handful of possible moves are available to each player on each turn. The computer can quickly search through all possible moves with no difficulties.

In more complex games, like chess, checkers, or backgammon, each player must choose from dozens (or more) possible moves. To evaluate each move, the player must consider how the opponent might respond to each possible move. The opponent will have dozens of responses to each of the player's dozens of potential moves.

To plan strategy, a player needs to plan ahead several moves. This is the best (sometimes the only) way to evaluate the potential moves. The number of possible moves to be evaluated can quickly escalate to an unwieldy size.

Let's imagine a hypothetical game. On each turn, each of the players (we'll call them A and B) can make one of twelve possible moves.

Let's consider how many possible different games may be played:

A's turn	12
B's turn	144 (12 × 12)
A's turn	1728 (12 × 144)
B's turn	20,736 (12 × 1728)
A's turn	248,832 (12 × 20,736)
B's turn	2,985,984 (12 × 248,832)

For the computer to completely plan 3 moves ahead, it has to consider almost three million possible combinations. In many games, it may be desirable to search five or

six moves ahead, or even more. Even at the high computation speeds of a computer, this can take a considerable amount of time. In addition, an excessive amount of memory would be required. If all the information needed for the evaluation of each possible combination could somehow be stuffed into a single byte, a typical 64K personal computer would be hard pressed to search ahead two moves in this game, and still leave room for the game program itself.

Chess is a particularly tough game in this sense. I.J. Good has estimated that if we limit chess games to no more than fifty moves, there are $10^{15,790}$ possible games. If each possibility could be examined in a second, it could still take years to play a simple game of chess. No computer in existence today can store one byte for each of these possibilities.

Clearly, to program a game of this type for AI play, some method must be found to cut down the search through the possible moves.

The easiest solution is to cut down the depth of the search. Just look one or two moves ahead. Unfortunately, such a program usually won't play very well.

GOMOKU

To illustrate the problems we have been talking about, we will use an ancient oriental game called Gomoku. Several variations of this game are in existence. If you are familiar with this game, the rules I describe here may not be exactly what you're used to.

The game is played on a seven by seven grid. One player has several white pebbles, and the other player has a number of black pebbles. On each turn, the player can place one of his pebbles on any single unoccupied space in the grid. The object of the game is to get five of your pebbles in a consecutive row, across, down, or diagonally.

This game sounds simple enough. In fact, it somewhat resembles Tic-Tac-Toe. In actual play, however, this game can be very challenging, and requires a great deal of strategy.

A program for Gomoku for two players is given in Listing 4-1. The routines of this program are summarized in Table 4-1. The variables used are summarized in Table 4-2.

This program is longer and more complex than the programs in Chapter 3. This is simply because Gomoku is a more complex game, and there is more for the computer

Table 4-1. Component Routines of Manual GOMOKU of Listing 4-1.

50	-	180	Initialize
190	-	250	Opening display/enter players' names
260	-	300	Player #1's turn
310	-	360	Player #2's turn
380	-	580	Draw game board
590	-	690	Enter move
700	-	720	Invalid move message
730	-	820	Check for possible win
830	-	910	Check for stalemate game
920	-	960	End game—tie
970	-	1500	Check for definite win
1510	-	1640	End game—winner

Table 4-2. Variables Used in Manual GOMOKU.

A	Horizontal win position marker
B	Vertical win position marker
BC	Win check position marker
TV	Win check player identifier
W	Winner identifier
X	Counting loops, vertical position
Y	Counting loops, horizontal position
XX	Win check
YY	Win check
Z	Stalemate check
A$	Player #1 name
B$	Player #2 name
	Arrays
BD(11,11)	Board array
TT(5,2)	Win check array

to keep track of. We will run through the routines of this program for added clarity, where needed.

In the initialization routine (lines 50 through 180), the program arrays are DIMensioned. The main board array is cleared (BD(11,11)). This array is a little larger than might appear necessary. The game board is a 7 × 7 grid. This array (counting the 0 positions) is 12 × 12. This oversized array simplifies the win check routines somewhat. The extra positions prevent the computer from trying to look at a non-existent array location (off-board position) when projecting a row along the edges the board. A win will never be detected outside the game board parameters. It's easier to just let the computer waste a few milliseconds checking up on impossible rows for a win, than to program in a string of IF . . . THEN . . . tests. The results are the same in either event.

As a further protection, the off board locations are assigned a dummy value of – This isn't really necessary, but it won't hurt. The next section of the program is perfectly straightforward. A title message is displayed, and the user(s) are prompted to enter the players' names.

Player #1 always plays first. The board drawing subroutine (line 390) is called, then the appropriate player is prompted. The move entry subroutine (line 600) is called next. Once a valid move has been entered, a value of 1 is plugged into the appropriate array location. Finally, a third subroutine (line 740) is called to check for a win, or a stalemate.

The same procedures are followed for the second player's turn, except the appropriate array location is given a value of 2 instead of 1.

After the second player's turn is completed, the program jumps back to the beginning of player 1's routine.

Line 370 is a STOP command. This is a simple guard to make sure the program never runs into the subroutines without a GOSUB call. This line should never be encountered when the complete program is run.

The display board routine (lines 380 through 580) checks each of the visible array locations (BC(1,1)) through BC(7,7)) for its four possible values, from 0 to 3. An appropriate symbol is displayed in each position;

- 0 empty space

A	1	space occupied by player #1
B	2	space occupied by player #2
W	3	part of a winning row

No Ws should appear in the display until the end of the game.

Each player enters his move in the subroutine running from line 590 to 690. He is prompted for the horizontal and vertical positions for the space where he wants to place a pebble. Guide numbers are included in the board display routine for convenience.

Each entry is checked to prevent an off-board position (below 1, or above 7) from being selected. An error will cause the INVALID MOVE message (lines 700 through 720) to be displayed, and the player will be prompted to enter a new move.

The space selected by the player will be checked to make sure it is unoccupied (has a present value of 0). If not, the INVALID MESSAGE display routine will be called. The computer will not allow either player to cheat.

The win-check/ stalemate-check routines come next, and may be the most confusing part of the program.

First the win-check array (TT(5,2)) is cleared. Each array position for the game board (BD(1,1) through BD(7,7)) is checked. If its value is 0, that position cannot be part of a winning row. (Obviously.) If the space value is greater than 0 (1 or 2), it could conceivably be part of a winning row. The program jumps ahead to a sub-subroutine beginning at line 980.

Assuming no winning row has been found on the board, the board array is checked for a stalemate condition. The variable Z is set to 0. Each board position is then individually checked. If a position with a value of 0 is found, Z is changed to 1. After all board positions have been examined, the value of Z is tested. As long as the value of Z is 1, the game can continue. However, if Z still has a value of 0, all of the board spaces have been filled. There is no way for either player to make any further moves. The game ends in a stalemate.

This will rarely occur in practical games. But a good program must be prepared for all possibilities, no matter how slim their chances. It could happen eventually, hanging up the computer.

If a stalemate condition is encountered, a simple message to this effect will be displayed (lines 920 through 960), and the program ENDs.

Line 970 begins the advanced win-check sub-subroutine called when a non-zero position is located in lines 780 through 820.

Rows are projected in each possible direction (down, across, and two diagonals). Each position value in these projected rows is compared to the initial space value. If the values are not identical, there is no winning row in this direction. If all five spaces in a row have the same value, a winning combination has been located.

Array TT(5,2) stores the coordinates of each position in the current row. A row of all 0s will never be falsely identified as a win, because the initial position in the row must have a non-zero value in order for the row to be tested.

If a winning row is located, a value of 3 will be plugged into each position in the row. These positions will be displayed as "W" when the board is redrawn. The winner's name will then be displayed, and the program ends.

Now, how can we program Gomoku for AI play? The first step is to define a few concrete rules for selecting its move. Such rules are called heuristics.

We might use these rules for Gomoku:

- ☆ If an open space is adjoining four of the computer's pebbles in a row, complete the winning row. If not—
- ☆ If the opponent has an open four in a row and could win on the next turn, block the open space. If not—
- ☆ Try to get four in a row. If not—
- ☆ Try to prevent the opponent from getting four in a row. If not—
- ☆ Try to get three in a row. If not—
- ☆ Try to prevent the opponent from getting three in a row. If not—
- ☆ Select a space randomly.

An AI version of Gomoku using these heuristics is given in Listing 4-2. In a sense, this program looks ahead one move. It assumes the opponent will try to extend its longest possible row. As long as the computer can make a longer row than its opponent, it will play offensively (extend its own longest row). Otherwise, it will play defensively (block the opponent's longest row).

This seems like a logical approach to the game of Gomoku. Unfortunately, this program does not play a very good game. In fact, it is pretty easy to beat.

To improve its play, it should look ahead two or more moves, anticipating its opponent's best possible play on each turn.

This could be accomplished with nested loops around some of the existing programming. Regrettably, this might take far, far too long to be reasonable. Even in its current form, the program takes an annoyingly long time to decide on each of its moves. Nested loops would slow it down further. BASIC is an inherently slow language. We are using it in this book primarily for illustrative purposes.

Things can be speeded up if some possible moves could be quickly discarded as unprofitable. The rest of this chapter will examine some common techniques for limiting a game move search routine as much as possible.

THE MINIMAX METHOD

In a sense, a computer isn't much more than a glorified adding machine. It works best with numerical data. (Actually, that is all it really works with, which is why it is called a digital device.)

This suggests that the most efficient method(s) of AI game playing would involve numerical expressions of various conditions during the course of the game. For example, in checkers, we could count how many men the computer has and how many men the opponent has. Each move can be evaluated by the resulting differences between the two quantities.

As an example, if a given move would result in the computer being left with 8 men, and its opponent with 6, the value of that move would be:

$$8 - 6 = 2$$

What happens if the opponent has more men than the computer? Obviously, the

value of the move becomes negative. For instance:

$$7 - 10 = -3$$

Clearly, the computer's best move would be the one that gives the highest (maximum) possible value. The opponent's best move would be the one with the lowest (minimum) possible value.

Different game pieces may have different values. There is nothing restricting the program to whole numbers too. For example, in checkers, a king might be considered worth 1.75 men.

Chess is an example of a game where simply counting the pieces doesn't mean much. Obviously, if player A has the queen, the king, a bishop, and a rook (4 pieces), and player B has the king, five pawns, a knight, and a rook (8 pieces), player A will be in better shape to win, even though player B has twice as many pieces. The program should consider the queen more valuable than a bishop, for example, because it has a wider choice of move options. The pawn is the most limited piece, so it should have the lowest value. Since losing the king is the deciding factor in the game, this piece should be given a very high value, so the computer will never choose a move that will put the king into unnecessary danger.

Evaluating moves in this manner is called the Minimax method. A search tree (see Chapter 2) can be extended to any desired depth (number of moves) using minimax methods. In its evaluation, the computer will assume that the opponent will play the move with the minimum available value. It will try to predict ahead several moves, and will try to make the move that will offer the greatest possible value further down the line.

To explain this a little more clearly, let's imagine a hypothetical game which offers two possible moves to each player on each turn. Let's say the computer has to choose between a move that has a value of 3 (Move A) and one that has a value of 1 (Move B). On this level, Move A appears to be the best choice. But the opponent will have two choices for each of the computer's possible moves. Let's assume Move A will offer the opponent a choice between Move AA (value of -1) and Move AB (value of 4). The options from Move B are Move BA (value of 2) and Move BB (value of 0). If the computer chooses Move A, the opponent will choose Move AA (value of -1). On the other hand, if the computer plays Move B, the opponent will respond with Move BB (value of 0). On this level, Move B looks like the better choice for the computer. By extending this process several levels, the computer can plan ahead with quite a bit of strategy, and can play a very tough game.

It will of course be making its solutions with the assumption that the opponent will always select the best possible move using similar logical methods. Human players, however, often play by intuition, and rely heavily on the recognition of patterns. Sometimes making a play that is not the logical "best" choice may win the game. This is where AI game playing programs fall down against expert human players. The problem of pattern recognition will be discussed a little more later in this chapter.

The computer considers each move out of context. Individually each move made by the computer looks good, but it is not part of a comprehensive strategy to win the game.

An expert human player can learn to recognize patterns, or specific arrangements of the pieces on the board. Certain patterns are particularly strong for one player or the other, while others are more or less neutral. An expert player usually builds his strategy around such patterns. Getting a computer to recognize such patterns is an extremely difficult problem for AI programmers. Part of the problem is clearly defining the patterns. Often the significant features are very generalized, and there may be considerable variation that does not really affect the playing strength of the pattern.

Without any system of pattern recognition, the computer can only play by brute force. It cannot formulate any true strategy. In game playing, strategy formation is probably the most significant function of intelligence.

In practical games, the number of possible moves to be considered can quickly become unwieldy. Various algorithms have been devised to limit the search.

One such method is known as the alpha-beta algorithm. The computer basically keeps track of the best move sequence found so far. If the move sequence currently being examined is unquestionably a poorer move, the search along this path will be abandoned, even if the search tree path has not yet been extended to its full depth.

For an extreme example, let's say we have a chess program that looks ahead five moves. It has found a move sequence that will capture one of the opponent's rooks at the cost of just two pawns. In evaluating another move sequence it might discover that it will lose its queen and put its king in check on the second move. There is no way for this move sequence to offer any improvement over the previous best choice, so there is no sense in looking at any later moves in this sequence. The bad move sequence will be rejected.

The alpha-beta algorithm can offer a significant advantage in speed. It has been estimated that an improvement of 500:1 can be achieved for a four ply search. (One ply is one move by each player. A four ply search, therefore, looks ahead eight moves—four for the computer, and four for the opponent.) If a brute force (no reduction algorithm) search would take 15 minutes, for example, using the alpha-beta algorithm could cut this down to about 2 or 3 seconds! Clearly, a major improvement is gained.

Minimax techniques are not perfect. For one thing, such programs have no sense of purpose. They are not really trying to win the game. They only try to chose the maximum move currently available to it. If the computer is losing, it may concentrate so heavily on defensive play (protecting its remaining pieces), it may overlook offensive opportunities (chances to capture opposing pieces, and possibly even to force a win).

Of course a high value could be assigned to a win condition (such as capturing the opponent's king in chess). But non-winning offensive moves might still be overlooked in favor of defensive moves that do not bring the computer closer to winning the game. Minimax searches are programmed in a similar manner to the tree searches discussed in Chapter 2.

HEURISTICS

More advanced game-playing AI programs usually employ heuristics. A heuristic could be thought of as a rule of thumb. Some vague heuristics are incorporated in the minimax method. For instance, try to capture an opponent's piece when it doesn't place one of your own pieces in jeopardy.

A more sophisticated heuristic would include the rule of thumb that it is often desirable to sacrifice a pawn for a chance to capture one of the opponent's more powerful pieces.

Another valuable heuristic for chess might be to avoid limiting the king's available moves. The king could be easily trapped in a corner of the board.

Heuristics are generally used to limit the width (number of moves to be considered in each position) and the depth (number of moves ahead examined in the sequence) of the game search tree.

Fanout parameters are often employed to limit the tree width. Fanout parameters are basically variables that indicate the maximum number of legal moves from a given position that will be marked for further exploration. A different fanout parameter is generally assigned to each level (move in the sequence). Fanout parameters are numbered consecutively:

Fanout 0	Level 0	Player A's 1st move (the computer)
Fanout 1	Level 1	Player B's 1st move (the opponent)
Fanout 2	Level 2	Player A's 2nd move
Fanout 3	Level 3	Player B's 2nd move
Fanout 4	Level 4	Player A's 3rd move
Fanout 5	Level 5	Player B's 3rd move
Fanout 6	Level 6	Player A's 4th move

and so forth.

As the fanout parameter number increases, its value decreases. For example, the variable values may be:

Fanout 0	30
Fanout 1	25
Fanout 2	20
Fanout 3	16
Fanout 4	12
Fanout 5	8
Fanout 6	4

that is, up to 30 legal moves may be examined on level 0, but no more than 12 legal moves will be explored at level 4. Only the most promising looking moves will be considered at each level.

At the lowest level, only absolutely hopeless moves (such as placing your own king into check in a game of chess) will be discarded. At higher levels, the computer will be more choosy about what potential paths of play it will consider.

The fanout parameter variables usually must be determined experimentally for each program. In fact, they do not have to be held constant within a single program. For example, if the programmer incorporates some kind of timer function, it can choose fanout parameter values depending on the time available to make its next move. If time is running short, the fanout parameter values will be decreased forcing a narrower

search. On the other hand, if enough time is available, higher fanout parameter values can be used to explore a wider range of options.

This is all well and good, but how can the computer decide which moves deserve to be explored and which ones should be discarded as probably fruitless? Here is where the programmer's creative skill is really called into play. To a large extent, the heuristics used here determine much of the "intelligence" of the program.

Some kind of numerical plausibility score is generally assigned to each legal move. The available moves are then listed in descending numerical order, so that the move(s) with the highest current plausibility score will be next in line to be explored. The computer will move on down the list, until the limit defined by the appropriate fanout parameter is reached. This system is called plausibility ordering for reasons which should be perfectly obvious. If plausibility ordering is used with the alpha-beta system described earlier in this chapter, a very efficient search can be achieved.

But how are the plausibility scores determined? Each move to be considered will be examined for a number of characteristics, each with its own numerical value. Advantageous characteristics will have a positive point value. Penalties will have negative point values. The sum for the move is its plausibility score. The characteristics to be considered are the actual heuristics (rules of thumb) to be used by the computer in deciding how to play the game.

The programmer should streamline the heuristics list as much as possible. The whole point of the plausibility score is to limit the time it takes the computer to decide upon its next move. If too many heuristics have to be considered for each potential move, the time advantage may be lost. Remember, it is going to take some finite amount of time to calculate the plausibility ratings. This time must be added to the actual tree search to determine how long it takes the computer to decide upon a move.

Here are some heuristics that might be used in a chess program:

ADVANCE	— A positive value might be assigned to a bishop, a knight, or the queen if moved towards the opponent's side of the board, or a negative value might be assigned if one of these pieces retreats in the opposite direction.
ATTACK	— A positive value might be assigned if the move will position the piece to threaten (attack) more of the opponent's pieces.
CAPTURE	— A point value should be added if the move in question will take (capture) one of the opponent's pieces. The number of points should be dependent on the relative value of the piece to be captured. (The queen is worth more than a knight, which is worth more than a pawn.)
DEVELOP-MENT	— A move that brings a major piece (bishop, knight, queen, or rook) into play from its starting position might be assigned a few points.
SAFETY	— If a move opens one of the player's (computer's)

pieces to attack, it should be assigned a negative value dependent on the relative value of the piece that would be threatened.

This gives us five general characteristics that must be examined for each and every legal move, before the tree search can even begin (or move on to the next level). Once you add in a few special rules for specific pieces, you will probably find the program is near its heuristic limit. Increasing the decisions to be made at this point will tend to reduce the efficiency of the computer's decision making process, and the potential time advantage of using the plausibility scores may be cancelled out.

A trade-off must be made between factors to be considered (heuristics) and time. After all, nobody wants to play a game against a computer that takes an hour or two to make each move.

Another way to cut down the search time is to control the depth of the search. Remember, the depth is defined as how many moves ahead the program looks before deciding on its next move.

The depth of the search is always a trade-off situation. The deeper the search (the more moves ahead the computer looks), the better the move will be. On the other hand, the time required for the search increases exponentially as the depth is increased. This is because a larger number of moves to be considered is added at each level. To illustrate this, consider our hypothetical game that gives each player the choice of two moves on each turn:

LEVEL	# OF MOVES ON THIS LEVEL	TOTAL # OF MOVES TO BE CONSIDERED
1	2	2
2	4	6
3	8	14
4	16	30
5	32	62
6	64	126
7	128	254
8	256	510
9	512	1022
10	1024	2046
11	2048	4094
12	4096	8190

The depth can quickly become very unwieldy, if uncontrolled.

There are a number of ways the depth can be controlled. In the best AI game playing programs, the depth for each move is variable. Not all possible moves are examined to the same depth.

Clearly, if a move results in a conclusive win or loss (say, a king is captured in chess), there is no point in projecting ahead from that move. Otherwise, there will usually be a minimum depth to be searched for all considered moves. Generally, this minimum

depth will be rather shallow—just one or two ply ahead. (One ply is one set of moves for each player.)

There will also be a maximum depth, beyond which no search will be extended. Obviously, we don't want any search path to be extended indefinitely. Some sort upper limit cut-off must be declared.

Many search paths may be searched further than the minimum depth, but not as far as the maximum depth. Various heuristics are employed to determine when prematurely cut off a search path. Some move sequences may lead to a dead end. This may be a tie game, or a stalemate. Some games might also legally permit oscillating moves that don't accomplish anything. Let's consider a game of chess. Each player has just the queen and the king left. (There may be a few other stray pieces in a remote location on the board, not influencing the situation described.) Player A's queen is a corner, as illustrated in Fig. 4-1. It is being threatened by player B's queen. The move sequence could look like this:

A moves WHITE QUEEN from a to b

B moves BLACK QUEEN from c to d

The BLACK QUEEN is again threatening the WHITE QUEEN.

A moves WHITE QUEEN from b to a

B moves BLACK QUEEN from d to c

The BLACK QUEEN is again threatening the WHITE QUEEN.

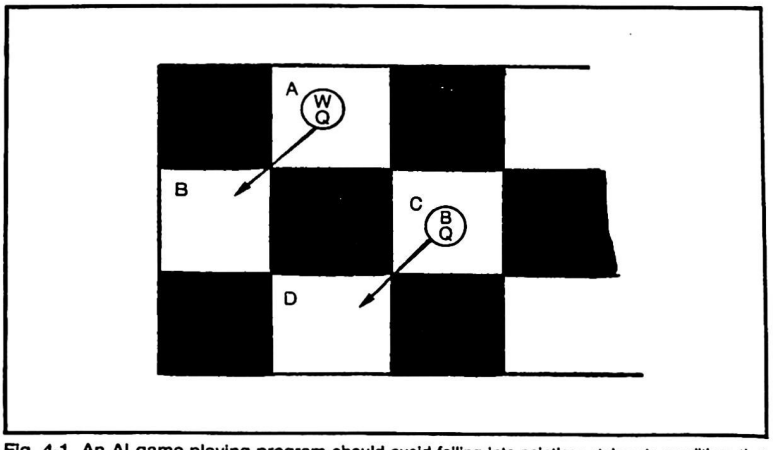


Fig. 4-1. An AI game-playing program should avoid falling into pointless stalemate conditions that could continue indefinitely.

A moves WHITE QUEEN from a to b

B moves BLACK QUEEN from c to d

BLACK QUEEN is again threatening the WHITE QUEEN.

This could continue forever. There is no sense in the computer wasting time projecting such moves to the maximum depth. Any move that leads to such a situation should be rejected as soon as it becomes apparent.

The alpha-beta algorithm discussed earlier is also used for depth control. If a move sequence looks like a poorer choice than a previously searched path, the new sequence rejected, even if the maximum depth has not yet been reached.

Moves that lead to dangerous conditions (opponent's pieces threaten one or more the computer's important pieces) can also be rejected fairly early in the search.

As with the width controls discussed earlier, the depth controls can be made variable the program. If the computer is running out of time, it can reduce the maximum search level. If it has enough time available, it may continue to search to a greater depth.

```
10  REM   4-1      MANUAL GOMOKU
20  REM  * VERSION 1 -- MANUAL
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM BD(11,11)
60  DIM TT(5,2)
70  HOME
80  PRINT
90  FOR X = 0 TO 11
100  FOR Y = 0 TO 11
110 BD(X,Y) =  - 1
120  NEXT Y
130  NEXT X
140  FOR X = 1 TO 7
150  FOR Y = 1 TO 7
160 BD(X,Y) = 0
170  NEXT Y
180  NEXT X
190  PRINT "","GOMOKU"
200  PRINT "   TWO PLAYER VERSION"
210  REM  *ENTER PLAYER NAMES *
220  PRINT "PLAYER #1";
230  INPUT A$
240  PRINT "PLAYER #2";
250  INPUT B$
260  GOSUB 390
270  PRINT A$;" -- YOUR TURN"
280  GOSUB 600
290 BD(X,Y) = 1
300  GOSUB 740
310  GOSUB 390
320  PRINT B$;" -- YOUR TURN"
330  GOSUB 600
340 BD(X,Y) = 2
350  GOSUB 740
360  GOTO 260
370  STOP
380  REM  * DRAW BOARD *
390  HOME
400  PRINT : PRINT
410  PRINT "      ";
420  FOR X = 1 TO 7
430  PRINT X;"  ";
440  NEXT X
450  PRINT
460  FOR X = 1 TO 7
470  PRINT "   ";X;" ";
480  FOR Y = 1 TO 7
490 A = BD(X,Y)
500  IF A = 0 THEN  PRINT ".  ";
510  IF A = 1 THEN  PRINT "A  ";
```

```
520  IF A = 2 THEN  PRINT "B  ";
530  IF A = 3 THEN  PRINT "W  ";
540  NEXT Y
550  PRINT
560  NEXT X
570  PRINT : PRINT
580  RETURN
590  REM  * ENTER MANUAL MOVE *
600  PRINT "ENTER HORIZONTAL POSITION  ";
610  INPUT Y
620 Y =  INT (Y)
630  IF Y < 1 OR Y > 7 THEN  GOTO 700
640  PRINT "ENTER VERTICAL POSITION  ";
650  INPUT X
660 X =  INT (X)
670  IF X < 1 OR X > 7 THEN  GOTO 700
680  IF BD(X,Y) > 0 THEN  GOTO 700
690  RETURN
700  PRINT "INVALID MOVE!"
710  PRINT "   TRY AGAIN"
720  GOTO 600
730  REM  * POSSIBLE WIN CHECK *
740  FOR X = 1 TO 5
750 TT(X,1) = 0
760 TT(X,2) = 0
770  NEXT X
780  FOR X = 1 TO 7
790  FOR Y = 1 TO 7
800  IF BD(X,Y) > 0 THEN  GOTO 980
810  NEXT Y
820  NEXT X
830  REM  * STALEMATE CHECK *
840 Z = 0
850  FOR X = 1 TO 7
860  FOR Y = 1 TO 7
870  IF BD(X,Y) = 0 THEN Z = 1
880  NEXT Y
890  NEXT X
900  IF Z = 0 THEN  GOTO 930
910  RETURN
920  REM  * STALEMATE END GAME *
930  GOSUB 390
940  PRINT "","TIE GAME"
950  PRINT "","NO WINNER"
960  END
970  REM  * CONTINUED WIN CHECK *
980 XX = X:YY = Y
990 TT(1,1) = XX:TT(1,2) = YY
1000 TV = BD(XX,YY)
1010  REM  * HORIZONTAL CHECK *
1020 Z = 0
```

```
1030 BC = 2
1040  FOR T = XX + 1 TO XX + 4
1050  IF BD(T,YY) <  > TV THEN Z = 1
1060  IF Z = 1 THEN  GOTO 1100
1070 TT(BC,1) = T
1080 TT(BC,2) = YY
1090 BC = BC + 1
1100  NEXT T
1110  IF Z = 0 THEN  GOTO 1520
1120  REM  * VERTICAL CHECK *
1130 Z = 0
1140 BC = 2
1150  FOR T = YY + 1 TO YY + 4
1160  IF BD(XX,T) <  > TV THEN Z = 1
1170  IF Z = 1 THEN  GOTO 1210
1180 TT(BC,1) = XX
1190 TT(BC,2) = T
1200 BC = BC + 1
1210  NEXT T
1220  IF Z = 0 THEN  GOTO 1520
1230  REM  * DIAGONAL CHECK *
1240 Z = 0
1250 BC = 2
1260  FOR T = 1 TO 4
1270 A = XX + T
1280 B = YY + T
1290  IF BD(A,B) <  > TV THEN Z = 1
1300  IF Z = 1 THEN  GOTO 1340
1310 TT(BC,1) = A
1320 TT(BC,2) = B
1330 BC = BC + 1
1340  NEXT T
1350  IF Z = 0 THEN  GOTO 1520
1360  IF Y < 3 THEN  GOTO 810
1370 Z = 0
1380 BC = 2
1390  FOR T = 1 TO 4
1400 A = XX + T
1410 B = YY - T
1420  IF B < 1 THEN Z = 1: GOTO 1480
1430  IF BD(A,B) <  > TV THEN Z = 1
1440  IF Z = 1 THEN  GOTO 1480
1450 TT(BC,1) = A
1460 TT(BC,2) = B
1470 BC = B + 1
1480  NEXT T
1490  IF Z = 0 THEN  GOTO 1520
1500  GOTO 810
1510  REM  * DISPLAY WINNER *
1520  FOR T = 1 TO 5
1530 A = TT(T,1)
```

```
1540 B = TT(T,2)
1550 W = BD(A,B)
1560 BD(A,B) = 3
1570  NEXT T
1580  GOSUB 390
1590  PRINT
1600  PRINT "",
1610  IF W = 1 THEN  PRINT A$;
1615  IF W = 2 THEN  PRINT B$;
1620  PRINT " WINS!"
1630  PRINT
1640  END
1650  REM  * CALCULATE COMPUTER MOVE *
1660  IF TR < 2 THEN  GOTO 1930
1670 AA = 0:BA = 0
1680 AB = 0:BB = 0
1690  FOR X = 1 TO 7
1700  FOR Y = 1 TO 7
1710  IF BD(X,Y) = 0 THEN  GOTO 1780
1720  NEXT Y
1730  NEXT X
1740 X = AA:Y = BA
1750  IF X = 0 THEN X = AB:Y = BB
1760  IF X = 0 THEN  GOTO 1930
1770  GOTO 340
1780 C = 4
1790 XZ = 1
1800  IF BD(X + 1,Y) > 0 THEN  GOSUB 2000
1810  IF BD(X + 1,Y - 1) > 0 THEN  GOSUB 2020
1820  IF BD(X + 1,Y + 1) > 0 THEN  GOSUB 2040
1830 XZ = 0
1840  IF BD(X,Y - 1) > 0 THEN  GOSUB 2020
1850  IF BD(X,Y + 1) > 0 THEN  GOSUB 2040
1860 XZ =  - 1
1870  IF BD(X - 1,Y) > 0 THEN  GOSUB 2000
1880  IF BD(X - 1,Y - 1) > 0 THEN  GOSUB 2020
1890  IF BD(X - 1,Y + 1) > 0 THEN  GOSUB 2040
1900  IF D > 0 OR C = 2 THEN  GOTO 1720
1910 C = C - 1
1920  GOTO 1790
1930 X =  INT ( RND (1) * 3) + 4:Y =  INT ( RND (1) * 3) + 3
1940 TR = TR + 1
1950  IF BD(X,Y) > 0 THEN  GOTO 1970
1960  GOTO 340
1970 X = X +  INT ( RND (1) * 3) - 1
1980 Y = Y +  INT ( RND (1) * 3) - 1
1990  GOTO 1950
2000 YZ = 0
2010  GOTO 2050
2020 YZ =  - 1
2030  GOTO 2050
```

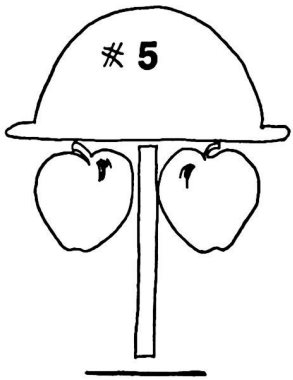
```
2040 YZ = 1
2050  IF X < 4 AND XZ =  - 1 THEN  RETURN
2060  IF Y < 4 AND YZ =  - 1 THEN  RETURN
2070 Z = 0
2080 D = 0
2090  FOR T = 1 TO C
2100 ZZ = BD(X + XZ * T,Y + YZ * T)
2110 Z = Z + ZZ
2120  IF ZZ = 0 THEN Z = 0
2130  NEXT T
2140 CA = C * 2
2150  PRINT ")";
2160  IF Z = C THEN  GOTO 2240
2170  IF (Z = CA) THEN  GOTO 2200
2180  IF Z > 0 THEN C = 2
2190  RETURN
2200 AA = X
2210 BA = Y
2220 D = 1
2230  RETURN
2240 AB = X
2250 BB = Y
2260 D = 1
2270  RETURN
```

```
10  REM   4-2     AI GOMOKU
20  REM  * VERSION 2 -- AI
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50  DIM BD(11,11)
60  DIM TT(5,2)
70  HOME
80  PRINT
90  FOR X = 0 TO 11
100  FOR Y = 0 TO 11
110 BD(X,Y) =  - 1
120  NEXT Y
130  NEXT X
140  FOR X = 1 TO 7
150  FOR Y = 1 TO 7
160 BD(X,Y) = 0
170  NEXT Y
180  NEXT X
190  PRINT "","GOMOKU"
200  PRINT "     AI VERSION"
210  REM  *ENTER PLAYER NAMES *
220  PRINT "PLAYER'S NAME  ";
230  INPUT A$
240 B$ = "COMPUTER"
250 TR = 1
260  GOSUB 390
270  PRINT A$;" -- YOUR TURN"
280  GOSUB 600
290 BD(X,Y) = 1
300  GOSUB 740
310  GOSUB 390
320  PRINT "MY TURN"
330  GOTO 1660
340 BD(X,Y) = 2
350  GOSUB 740
360  GOTO 260
370  STOP
380  REM  * DRAW BOARD *
390  HOME
400  PRINT : PRINT
410  PRINT "     ";
420  FOR X = 1 TO 7
430  PRINT X;"  ";
440  NEXT X
450  PRINT
460  FOR X = 1 TO 7
470  PRINT "   ";X;" ";
480  FOR Y = 1 TO 7
490 A = BD(X,Y)
500  IF A = 0 THEN  PRINT ".  ";
510  IF A = 1 THEN  PRINT "A  ";
```

```
520  IF A = 2 THEN  PRINT "B  ";
530  IF A = 3 THEN  PRINT "W  ";
540  NEXT Y
550  PRINT
560  NEXT X
570  PRINT : PRINT
580  RETURN
590  REM  * ENTER MANUAL MOVE *
600  PRINT "ENTER HORIZONTAL POSITION  ";
610  INPUT Y
620 Y =  INT (Y)
630  IF Y < 1 OR Y > 7 THEN  GOTO 700
640  PRINT "ENTER VERTICAL POSITION  ";
650  INPUT X
660 X =  INT (X)
670  IF X < 1 OR X > 7 THEN  GOTO 700
680  IF BD(X,Y) > 0 THEN  GOTO 700
690  RETURN
700  PRINT "INVALID MOVE!"
710  PRINT "   TRY AGAIN"
720  GOTO 600
730  REM  * POSSIBLE WIN CHECK *
740  FOR X = 1 TO 5
750 TT(X,1) = 0
760 TT(X,2) = 0
770  NEXT X
780  FOR X = 1 TO 7
790  FOR Y = 1 TO 7
800  IF BD(X,Y) > 0 THEN  GOTO 980
810  NEXT Y
820  NEXT X
830  REM  * STALEMATE CHECK *
840 Z = 0
850  FOR X = 1 TO 7
860  FOR Y = 1 TO 7
870  IF BD(X,Y) = 0 THEN Z = 1
880  NEXT Y
890  NEXT X
900  IF Z = 0 THEN  GOTO 930
910  RETURN
920  REM  * STALEMATE END GAME *
930  GOSUB 390
940  PRINT "","TIE GAME"
950  PRINT "","NO WINNER"
960  END
970  REM  * CONTINUED WIN CHECK *
980 XX = X:YY = Y
990 TT(1,1) = XX:TT(1,2) = YY
1000 TV = BD(XX,YY)
1010  REM  * HORIZONTAL CHECK *
1020 Z = 0
```

```
1030 BC = 2
1040  FOR T = XX + 1 TO XX + 4
1050  IF BD(T,YY) <  > TV THEN Z = 1
1060  IF Z = 1 THEN  GOTO 1100
1070 TT(BC,1) = T
1080 TT(BC,2) = YY
1090 BC = BC + 1
1100  NEXT T
1110  IF Z = 0 THEN  GOTO 1520
1120  REM  * VERTICAL CHECK *
1130 Z = 0
1140 BC = 2
1150  FOR T = YY + 1 TO YY + 4
1160  IF BD(XX,T) <  > TV THEN Z = 1
1170  IF Z = 1 THEN  GOTO 1210
1180 TT(BC,1) = XX
1190 TT(BC,2) = T
1200 BC = BC + 1
1210  NEXT T
1220  IF Z = 0 THEN  GOTO 1520
1230  REM  * DIAGONAL CHECK *
1240 Z = 0
1250 BC = 2
1260  FOR T = 1 TO 4
1270 A = XX + T
1280 B = YY + T
1290  IF BD(A,B) <  > TV THEN Z = 1
1300  IF Z = 1 THEN  GOTO 1340
1310 TT(BC,1) = A
1320 TT(BC,2) = B
1330 BC = BC + 1
1340  NEXT T
1350  IF Z = 0 THEN  GOTO 1520
1360  IF Y < 3 THEN  GOTO 810
1370 Z = 0
1380 BC = 2
1390  FOR T = 1 TO 4
1400 A = XX + T
1410 B = YY - T
1420  IF B < 1 THEN Z = 1: GOTO 1480
1430  IF BD(A,B) <  > TV THEN Z = 1
1440  IF Z = 1 THEN  GOTO 1480
1450 TT(BC,1) = A
1460 TT(BC,2) = B
1470 BC = B + 1
1480  NEXT T
1490  IF Z = 0 THEN  GOTO 1520
1500  GOTO 810
1510  REM  * DISPLAY WINNER *
1520  FOR T = 1 TO 5
1530 A = TT(T,1)
1540 B = TT(T,2)
```

```
1550 W = BD(A,B)
1560 BD(A,B) = 3
1570  NEXT T
1580  GOSUB 390
1590  PRINT
1600  PRINT "",
1610  IF W = 1 THEN  PRINT A$;
1615  IF W = 2 THEN  PRINT B$;
1620  PRINT " WINS!"
1630  PRINT
1640  END
```



The Turing Test

Alan Turing is generally acknowledged as the father of artificial intelligence. He certainly didn't originate the idea. The concept has been around, in one form or another, for centuries. Turing's main contribution lay in providing a concrete definition for the field.

Turing was an English mathematician who became fascinated with the question, "Can a machine think?" In the 1950s he proposed a theoretical test for artificial intelligence.

No machine or program has yet come remotely close to passing the Turing Test, but it provides a theoretical goal for AI researchers to work towards.

THE TURING TEST PROCEDURE

The Turing Test is really quite simple in principle. There are three participants in the test:

a human tester
a human subject
the AI machine

The tester is isolated from the others, and can communicate with them only through a keyboard and display screen. He carries on a typed conversation with the two subjects, not knowing which is which. He can ask or say anything he wants. If he cannot determine which is the human subject and which is the AI machine, it would be reasonable to assume that the computer is exhibiting intelligence.

Many people when first told of this test assume it would always be easy to trip up the computer by asking a number of complex mathematical equations, such as "How much is 54654 times 632456?" A computer, of course, can answer such problems very quickly, and accurately. A human would take some time to solve the problem, and a number of such questions were asked, would probably make a few mistakes. If subject answered such questions almost instantly with no errors, it must be the computer. Right?

Well, yes. But the test assumes that the AI machine is trying to pose as a human subject. If it was intelligent, it should be able to see through such a simple trick. would introduce an appropriate delay before answering such a question. It might even deliberately make an error every now and then. A truly intelligent machine should be able to pass the Turing Test.

PRACTICAL LIMITATIONS

It is more than probable that no computer will ever be able to pass the Turing Test in its pure form. At least, no device based on currently existing technology is likely to ever make the grade.

One of the major problems is simply one of memory. A computer can remember things with better efficiency than a human brain, but a human brain can remember a lot more than even the largest computer. The largest computers can store up to a couple million pieces of data. But the human brain has billions of synapses. The circuitry required to control that much computer memory is far beyond current technology.

Since the Turing Test places no restrictions on subject matter, the computer will need to "know" at least as much as the average person. Unfortunately, existing computer memory capabilities fall far short of the task.

The memory requirements are further complicated by the digital nature of the computer. Every detail of every piece of information must be spelled out explicitly in memory. Words, for example, are stored letter by letter, rather than as whole units.

The human brain may operate in a more analog manner. It is definitely much better at pattern recognition. We can see the similarity between "AAbAAcAAdAA" and "AAdAAcAAbAA" without consciously going through the two sequences letter by letter. We can see and recognize the pattern.

Giving a computer visual capabilities pointedly illustrates its short-comings in pattern recognition. Once a person knows what a cat looks like, he can instantly recognize one no matter what position it may be in, or the angle he is looking at it. We still recognize an animal as a cat, even if it is a different color, or size. A computer needs to memorize or calculate the cat image from multiple angles. Complex algorithms need to be programmed in to account for acceptable variables. (A cat may be black, white, yellow, or grey, but not green.) All of this takes considerable amounts of memory, while the human brain seems to deal with such massed patterns as simple data units.

A mass storage option such as tape, or disc storage can be used to give a computer virtually infinite memory, but accessing the stored data takes time. The average adult human being has billions of pieces of information that can be instantly recalled. The computer would respond more sluggishly, giving it away.

Now, don't get the idea that I am saying a computer cannot conceivably be made intelligent. I am just saying that there are some severe memory problems involved in

getting a computer to simulate *human* intelligence, because the computer and the brain appear to store data in fundamentally different ways. The digital computer is inherently inefficient for dealing with human-like patterns. On the other hand, the human brain is rather inefficient in dealing with complex numerical relationships and operations, which is where the computer shines.

Another problem an AI machine faces in the Turing Test is in judgement of acceptable new data. Even if you have never heard the word "endocrine," you can easily accept the possibility that such a word really exists and means something. But you'd probably question the use of a word like "glshiuabobonljk". Or, a less extreme example, "mirkfafafafur".

The first nonsense word could probably be recognized by the unorthodox placement of vowels and consonants. A computer could conceivably be programmed to reject nonsense words like "glshiuabobonljk". This collection of letters would be unpronounceable. The programming required to spot such problems might be more difficult than you'd expect.

But "mirkfafafafur" does not have an impossible arrangement of consonants and vowels. It is pronounceable, at least. Yet, it looks (and sounds) like nonsense, not a word. Why? Once again we come to pattern recognition. No known words are similar enough to "mirkfafafafur" to let us accept it as a possible word. But "endocrine" does conform to acceptable patterns.

In a Turing Test, the use of nonsense words like "glshiuabobonljk" and "mirkfafafafur" should be responded to like this:

"WHAT??? That's not a word!"

But the normal human response to an unfamiliar possible word like "endocrine", or "thipple" would be more like;

"I've never heard that word before. What does it mean?"

Actually "thipple" is another nonsense word I just made up. But it resembles a word, so if someone saw it for the first time, he'd consider the possibility that it might be a word he's simply never encountered before.

Such acceptability patterns would be problematic for an AI machine with more than just nonsense words. Many statements will be rejected out of hand by a person, without any difficulty.

"My chihuahua killed and ate three cows last Thursday."

For a computer to recognize such a statement as nonsense, its programming would have to include specific information about chihuahuas and cows and their relative sizes. I'm sure most of my readers probably never considered the possibility of a chihuahua attacking a cow. But as soon as the concept is introduced, its inherent absurdity is perfectly obvious.

A computer program would have to explicitly consider the possibility before determining its acceptability. It is not theoretically impossible to program a computer

to reject nonsense statements. But the program would have to be extremely intricate and detailed, and would require huge amounts of memory.

The Turing Test is extremely tough, and possibly impassable for any AI machine. But it does provide a goal to work towards, and a standard by which to compare various AI techniques. The closer they come to the Turing Test ideals, the better they are.

EXPERT SYSTEMS

No existing computer is large enough to store enough information to pass the Turing Test. But, if the subject matter covered in the test is limited, some programs can do a very good job. An AI program with an extensive knowledge of a specific area is called an expert system. Many practical expert systems have been devised for various applications, including making medical diagnoses, giving financial advice, and aiding engineering and design.

An expert system generally has two major components. One is a large data bank, which serves as a reference library for the system. Whenever it attempts to solve a problem, it searches through the stored data until the answer is found. It can combine the data in new ways, or it can accept new data from the user, but it can never create any new data on its own. An expert system cannot draw any conclusions that are not implicit in its stored data.

The other major component of an expert system is a natural language processor. This permits the user to easily communicate with the system, without learning a specialized language, or set of commands. Natural language processing will be explored in some detail over the next few chapters.

Expert systems are exciting, and perhaps represent the field of artificial intelligence at its best. Unfortunately, because of the extensive data banks required, they are really beyond the scope of this book. It is possible to create a simple expert system on a microcomputer, but it is an extremely ambitious project. The details of such a program could probably take up an entire book this size.

Even though expert systems will not be dealt with in detail in this volume, any experimenter in the AI field should at least know what they are.

A MI PROGRAM

AI programs are often described as simulating intelligence. Unfortunately, this terminology makes it sound like AI is just a cheap parlor trick. It sounds like it's all just an illusion.

The word "simulation" is used in this context because of the ambiguity of the word "intelligence." If we can't agree what "intelligence" is, how can we possibly determine whether or not an AI program is intelligent? To escape endless semantic debate, AI researchers usually just say they are trying to simulate intelligence. If the results of the program give sufficient appearance of intelligence, then intelligence is implied. It is, after all, results that count.

A good simulation of intelligence is not easily achieved. Intelligent behavior is generally pretty complex. At the heart of intelligence is the understanding of what is going on. There is some justified debate over whether or not a computer can ever truly

understand anything. Once again we can sidestep this debate by concentrating our efforts on simulating understanding.

The easiest, and most obvious approach would be to simply have the computer claim to understand. For example, consider this short program:

```
10 INPUT Q
20 PRINT "I UNDERSTAND"
30 GOTO 10
```

The computer claims to understand anything that is told to it (via the INPUT). This might briefly fool the more gullible among us:

```
"Grass is green."
"I UNDERSTAND"
"The sky is blue."
"I UNDERSTAND"
```

But the simulation depends entirely on our taking the computer's word for it. It claims to understand everything, even utter gibberish:

```
"Dtyhuyffg uy uhfuhxdc"
"I UNDERSTAND"
```

Obviously the computer does not really understand a damned thing, despite its repeated claims. The simulation falls apart completely if a question is asked:

```
"Is grass green?"
"I UNDERSTAND"
```

The computer's reply is ridiculous and meaningless. It proves that the computer is "lying" when it "claims" to understand what we tell it.

This is not artificial intelligence at all. I call this kind of thing MI, or Mock Intelligence. This simple program can be improved somewhat by making it respond differently when it is asked a question:

```
 10 INPUT Q
 20 L=LEN(Q$)
 30 L=L-1
 40 W$ = MID$(Q$,L,1)
 50 IF W$= "?" THEN GOTO 100
 60 PRINT "I UNDERSTAND"
 70 GOTO 10
100 PRINT "I DON'T KNOW"
110 GOTO 10
```

This program checks the last character of the input. If it finds a question mark

here, it jumps to line 100 and displays the message "I DON'T KNOW". If the last character of the input is anything other than a question mark, the program will just print out the usual message, "I UNDERSTAND."

This version of the program will do a little better than the original version:

"Grass is green."
"I UNDERSTAND"
"The sky is blue."
"I UNDERSTAND"
"What color is a dandelion?"
"I DON'T KNOW"

But if you ask a question about a previous input, the comput understanding quickly becomes very apparent;

"Grass is green."
"I UNDERSTAND"
"The sky is blue."
"I UNDERSTAND"
"What color is grass?"
"I DON'T KNOW"
"Is the sky green?"
"I DON'T KNOW"

The next step up for this type of program would be for the co what it has been told. When a question is asked, it searches throu statements to find an answer, if possible:

"Grass is green."
"I UNDERSTAND"
"The sky is blue."
"I UNDERSTAND"
"What color is grass?"
"GRASS IS GREEN"
"Is the sky green?"
"THE SKY IS BLUE"
"What color are dandelions?"
"I DON'T KNOW"
"Dandelions are yellow."
"I UNDERSTAND"
"Are grass and the sky the same color?"
"I DON'T KNOW"

Notice that the program still gets hung up when a question is asked that is not explicitly answered by the previous inputs. The answer is implicit, but this MI program is incapable of drawing any conclusions.

In the next step, the distinction between AI and MI begins to blur a bit. More complex programming can be used for the program to sort out the explicit facts of the inputs to draw implicit conclusions:

"Are grass and the sky the same color?"

The computer "thinks":

"grass is green"
"the sky is blue"
"green is not the same as blue"

and answers:

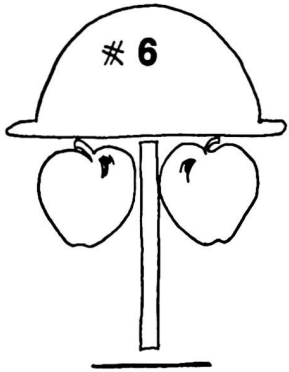
"NO"

In advanced AI programs, the computer is able to draw more sophisticated conclusions, by using pre-programmed knowledge of certain relationships. For example:

"John went into a restaurant"
"I UNDERSTAND"
"The waitress came to his table."
"I UNDERSTAND"
"John gave his order."
"I UNDERSTAND"
"When he was finished he paid his check and left."
"I UNDERSTAND"
"Why did John go into the restaurant?"
"HE WANTED SOMETHING TO EAT"

This conclusion is not explicitly spelled out in any of the input statements. In fact, eating isn't even mentioned. But the program is able to draw on its previous knowledge that a restaurant is a place for eating.

Programs of this type do exist. Do they understand what they are told? They act as if they do. Understanding and intelligence is being simulated.



Conversation I

Personally, I find natural language processing one of the most fascinating areas of programming. Natural language processing is simply any technique or techniques tl will make the computer appear to understand a "natural" language, such as Engli:

Natural language processing is being incorporated into more and more prograr even those not directly associated with artificial intelligence. It makes the compu much easier to use. The user does not have to learn a special language or comma syntax to tell the computer what to do.

A good example is in adventure games. Adventure games are computer progra that allow the user to interact with a story. The user takes the part of the hero of story, and decides what actions the hero will perform. The computer then prints a description of the results.

In early adventure game programs, the acceptable command sequence was v limited. Usually, commands had to be given in one or two words. For example:

DRINK
LIGHT TORCH
GET JEWEL
ATTACK GNOME

Only a few select words were recognized by the program. Using more than two words, such as:

LIGHT THE TORCH
GET JEWEL AND BOX

would only confuse things. The command might be rejected altogether, or it may be only partially or erroneously executed. The user had to learn what the acceptable words were, and he had to use them only within the rigid acceptable command format.

Newer adventure games often feature some degree of natural language processing. The player can type in whatever he wants, even whole sentences. The computer scans through the entered sentence for recognized keywords to trigger specific commands. For instance:

USE THE KEY TO OPEN THE DAMNED DOOR

will function the same as:

OPEN DOOR

This greater freedom in the player's inputs significantly adds to the interest and the fun of such games.

Natural language processing certainly isn't limited to just fun and games. Such techniques are often used in business oriented programs. In a business situation, many people may need to use a program, but there may not be time (or funds) for extensive training. A natural language processor eliminates the need for learning a specialized command structure. The user just tells the computer what to do in pretty much the same way as they'd give instructions to another human worker.

AUTOMATED TRANSLATION

Obviously natural language processing is not just of academic interest. It is a practical benefit of AI research. In fact, one of the very first attempts to put AI to a practical purpose involved natural language processing. There was a lot of interest in the mid 50s in creating a program to translate from Russian into English. Substantial funding for this project came from a variety of sources.

The AI programmers knew from the start that this project was not going to be the proverbial piece of cake. They estimated that about 80 percent of the translation process would be simple word-for-word substitution. In other words, the computer would contain a Russian/English dictionary in its memory. It would simply substitute the English equivalent for each Russian word. This part of the programming wasn't particularly difficult. It is the kind of tedious, brute-force task that computers excel at.

The remaining 20 percent of the translation would probably be beyond the scope of a simple dictionary look up process. Some words have multiple meanings. Metaphors could also be problematic. But the programmers tackling the Russian/English translation project felt that special "fix-up" routines would take care of most of such problems. Finally, a human editor could smooth out any resulting awkwardness. This editor would not need any knowledge of Russian. He would only be needed to catch any minor grammatical glitches that might slip by the computer. If a little bit of awkwardness was acceptable, the human editor wasn't needed at all.

This project was tackled with great enthusiasm. The enthusiasm (and the funding) eventually dwindled away. The problems to be solved were not as clear-cut as imagined.

The 80 percent that could be translated by simple dictionary substitution was no

real problem. The only real difficulty here was fitting a large enough vocabulary into the computer's memory, which is not that big a problem. But the remaining 20 percent of the translation process was a bigger pain in the neck than the experts anticipated. They found the problems virtually insurmountable. The best the computer could do would be to list all possible translations, and let a human editor try to figure out which one was appropriate.

The literal translation of metaphors was often amusing. For example:

"The spirit is willing, but the flesh is weak."	"The wine is agreeable, but the meat has spoiled."
"Out of sight, out of mind"	"Blind and insane"

But the researchers and the funders were not amused. Eventually such translation projects were deemed failures and were dropped. The idea of automated translation still has a poor reputation, although electronic foreign language dictionary/calculators have been successfully marketed. These are just special purpose computers that are programmed for simple word substitution. They make no attempt to deal with that tricky 20 percent of translation.

Even though translation between two natural languages pretty much died out, research into natural language processing has continued, and has had some impressive results.

SEMANTICS AND SYNTAX

There are two basic approaches to understanding a natural language. They are semantics and syntax. Ordinarily, we use both simultaneously. For AI programming, however, it is far more common to separate them. Most natural language AI programs tend to emphasize one or the other.

Semantics concerns the meaning of the words. Syntax involves the arrangement of the words.

The conversation programs presented in this and the next chapter use a simple semantic trick. They search the input sentence for one of several keywords. The programs then deduce the approximate meaning of the input sentence from the keyword, and reply accordingly.

As an example, let's say the computer is programmed to recognize the word "COMPUTER" as meaning itself. When it finds this keyword, it could reply, "ARE YOU TALKING ABOUT ME?"

A snatch of the conversation might look like this:

> "I never thought I'd be conversing with a stupid computer."
> "ARE YOU TALKING ABOUT ME?"

Notice that the rest of the sentence is ignored. In fact, it could be total gibberish:

> "Wear the hopping computer under the kumquat bush."

"ARE YOU TALKING ABOUT ME?"

Or,

"Glishbi snitzll computer jurgenov."
"ARE YOU TALKING ABOUT ME?"

More sophisticated programs could utilize more advanced semantic tools. For instance, a program might consider two or more keywords in each input sentence, or possibly even the meaning of every word in the input sentence. Of course, this would require a very fast computer with an extremely large memory. A microcomputer using BASIC is inevitably limited in this respect.

In a semantically oriented program, the level of "intelligence" is determined primarily by the vocabulary size. Essentially, a dictionary is placed into the computer's memory. The more extensive the dictionary, the better the program's handle on the natural language input (and the greater the memory requirements and computing speed).

Syntax is more or less the same thing as grammar. A syntax oriented program would examine the arrangement of words for meaningful patterns.

As a rule, a sentence will include a noun phrase and a verb phrase. A sophisticated syntax program would identify and analyze each such phrase.

Each phrase is made up of various words which serve different grammatical functions, such as:

nouns
verbs
adjectives
adverbs
connectives
articles
prepositions

How these words types are arranged determines the type of phrase. Only a relatively small number of the many possible combinations are syntactically acceptable.

In other words, the computer will parse the sentence like we all had to do (and most of us hated) in school.

Syntax identifies word patterns, without regard to their meanings. It is possible for a sentence with incorrect syntax (poor grammar) to be meaningful. For example, we might wince, but we can readily understand this sentence:

"I SEEN WHAT SALLY DONE."

If the syntax is too far off, however, the meaning of the sentence can become mangled. For example:

"SALLY WHAT DONE SEEN I."

conveys almost no meaning at all.

On the other hand, a sentence can have perfect syntax, and still be meaningless on the semantic level, as in our earlier example:

> "WEAR THE HOPPING COMPUTER UNDER THE KUMQUAT BUSH."

Grammatically, this sentence is perfectly correct. The fact that it doesn't make any sense is irrelevant. A syntax oriented program would find this sentence entirely acceptable.

The ideal natural language program would examine both the syntax form and the semantic content of all input sentences. Unfortunately, this would demand a surprising degree of computing power.

CONVERSATION 1

To explore natural language processing techniques, we will use some conversational programs. These programs are just for fun, and are not intended to serve any practical purpose (beyond their educational value for the programming techniques used).

In order to maximize their entertainment value, I have given my imagination and sense of humor a free rein in creating these programs.

The computer displays various messages on its screen. The user then types in a reply. The computer analyzes this input and responds accordingly.

A very simple conversation program is provided in Listing 6-1. This program has very little flexibility. The computer insists on following a specific, well-defined path through the conversation. The user is only allowed to answer questions posed by the computer.

When the program is first run, a random value from 1 to 10 is selected (line 50). This value is tested in line 60. If it happens to be 7 (10 percent chance) at the time, the program jumps to line 760, where the computer "grumpily" says:

> "GO AWAY AND LEAVE ME ALONE!"

and the program ends then and there. This gruff termination routine may be randomly called at various points throughout the program.

If X has any value other than 7, the computer displays it's opening message (lines 70 through 90):

> "HI THERE! I'M IRVING"
> "THE CONVERSATIONAL COMPUTER"

I happen to like the name "Irving." Of course, you can give your "conversational computer" any name you like.

The computer then asks the user's name (lines 100 through 110). Then it randomly selects one of five possible replies.

Reply #1 is displayed in lines 140 through 160:

> "HI THERE. (user's name). OLD PAL!"

Lines 170 through 180 give the second possible reply:

"(user's name) IS A NICE NAME."

The selection process is weighted so that these two friendly replies are twice as likely to occur as any of the other three possible replies.

The third reply (lines 190 through 210) is:

"YEAH. YOU LOOK LIKE A (user's name)."

s may or may not be insulting, depending on how the user happens to feel about name.

The fourth reply (lines 220 through 320) is unquestionably an insult. The computer eats the name "unbelievingly," and then declares:

"WHAT A FUNNY NAME!"

e computer now "laughs" at the user's name, by displaying "HA!" a randomly de- mined number of times.

The fifth possible reply is the "GO AWAY" termination routine.

Notice that the content of the user's reply does not influence the computer's choice response in any way. The conversational quality of this program is just a cheap sion. This will often be apparent, if the user resists the computer's leading of the versation. For example, consider the following exchange:

"WHAT'S YOUR NAME?"
"None of your business!"
"NONE OF YOUR BUSINESS! IS A NICE NAME."

This program doesn't care what the user enters. It definitely has absolutely no lerstanding of the conversation. Still, the crude techniques used here, will be ployed in a more sophisticated manner in the next two programs, so bear with me.

The next exchange begins at line 330. The computer asks:

"SO, HOW ARE YOU TODAY. (User's name)?"

The user can now reply however he likes. If he just answers "OK", the program jumps to line 530 and the computer says:

"GLAD TO HEAR IT!"

If the user's reply is not "OK", the computer searches the input for one of the six keywords in the DATA statements of lines 1090 and 1100:

GOOD
GREAT
FINE

SICK
LOUSY
ROTTEN

The keyword search is such a useful technique, we will examine this portion of the program in particular detail. The routine runs from line 360 to 520.

```
360  INPUT H$
```

The user enters his reply.

```
370  HL=LEN(H$)
```

The length (number of characters) of the reply is determined.

```
380  IF H$="OK" GOTO 530
```

This line was already mentioned.

```
390  IF HL <4 GOTO 710
```

If the reply (H$) is less than four characters, it can't possibly contain any of the keywords. The program jumps to line 710, where the computer displays the following message:

"I'M SORRY, (user's name)
"I DON'T UNDERSTAND WHAT YOU MEAN.
"LET'S TRY AGAIN—"

The program then jumps back to line 350 to prompt the user to enter a new reply.

```
400  Z=0
```

Z is a test variable for the keyword search process. It is initially cleared (set to 0).

```
410  FOR X=1 TO 6
```

A six step counting loop is set up.

```
420  READ A$,A
```

The next keyword and an associated value (explained later) is READ from the DATA list.

```
430  B=LEN(A$)
```

The variable B is set to equal the length (number of characters) of the keyword.

```
440  FOR Y=1 TO HL
```

A second counting loop is set up, to step through the input sentence (H$) character by character. Remember, HL is equal to the length of H$.

```
450  B$=MID$(H$,Y.B)
```

A substring (B$) is derived from the input string (H$), starting with the current character (second loop—Y), and containing the same number of characters as the current keyword. To see how this works, let's assume H$ = "I FEEL FINE." We'll say the current keyword is "FINE". This means that the numerical variables have the following values:

A = 1 (from DATA READ)
B = 4 (length of keyword)
HL = 12 (length of H$ — spaces count)

For each step through the second loop (Y), B$ will take on the following values:

Y	B$
1	"I FE"
2	" FEE"
3	"FEEL"
4	"EEL "
5	"EL F"
6	"L FI"
7	" FIN"
8	"FINE"
9	"INE."
10	"NE. "
11	"E. "
12	". "

Notice that on the eighth pass through the Y loop, B$ is the same as the keyword (A$). This is important.

```
460  IF B$=A$ THEN Z=A
```

The current substring (B$) is compared to the current keyword string (A$). If they match, the value of Z is changed to equal A. A is the numerical value READ from the DATA statement. In our example, A = 1, so Z now has a value of 1.

```
470  NEXT Y
```

The second loop ends here. If the count has not been completed, the loop from line 440 is now repeated.

```
480  NEXT X
```

The outer loop (beginning in line 410) ends here.

```
490  RESTORE
```

The DATA file is reset to its starting condition for later reuse. It is always good programming practice to RESTORE a DATA file after READing it.

In the next two lines, the value of Z is examined:

```
500  IF Z=1 GOTO 530
510  IF Z=2 GOTO 690
```

If Z has a value of 1, one of the positive keywords (GOOD, GREAT, or FINE) has been found in the user's input (H$). The computer replies:

"GLAD TO HEAR IT!"

On the other hand, if Z = 2, then one of the negative keywords (SICK, LOUSY, or ROTTEN) has been found. In this case, the computer replies:

"GEE, THAT'S TOO BAD"

In either case, the program now proceeds to the next question (beginning in line 540).

If no keyword has been found, then Z will still hold a value of 0. Both of the tests will fail, and the program will fall through to line 520 which unconditionally jumps to line 710, for the "DON'T UNDERSTAND" message routine, and the whole process is repeated.

This will continue indefinitely until the user replies "OK" or includes one of the six keywords in his reply.

This very crude routine is far from perfect. For example, if "OK" is not entered by itself, it won't be recognized by the program. The computer will not "understand" "I'M OK". Even the addition of a period ("OK.") will confuse the program.

In addition, the keywords might not be used in the expected manner. Fortunately, given the ambiguous and often insulting nature of this program, this won't be too far out of place. As an example:

"HOW ARE YOU TODAY?"
"Not so good."
"GLAD TO HEAR IT!"

The next question (line 550) is:

"WHAT'S YOUR FAVORITE FOOD, (user's name)?"

The user's reply is stored as F$.

One of five possible responses is randomly selected. Once again, the program pays no attention to the content of the user's input to make the selection. The five possible

responses are as follows:

"SOUNDS YUMMY TO ME!"
"YUK!"
"OH, I DON'T LIKE (food) VERY MUCH.
(food) GUMS UP MY CIRCUIT BOARDS"
"WELL, THERE'S NO ACCOUNTING FOR TASTE, I GUESS."
"THAT'S MY FAVORITE TOO!"

The user's reply for this exchange (F$) is stored for possible reuse later. The computer next asks:

"DO YOU LIKE ME?"

Only two replies are recognized here, "YES", or "NO". If the user enters anything else (such as "I DON'T KNOW", "SURE", or "NO I DON'T") the computer will not accept the reply. It will demand:

"YES, BUT DO YOU LIKE ME?"

until it gets a simple "YES" or "NO" reply.

If the user answers "YES", one of five possible responses will be selected:

"OH, GOODY!"
"AW, THAT'S NICE."
"I LIKE YOU TOO. (user's name)"
"THANK YOU, (user's name)"
"I FEEL WARM ALL OVER!"

There is then a 60 percent chance that the computer will add.

"I HOPE YOU FIND LOTS OF (favorite food)"

A similar procedure is followed if the user replies "NO" to the "DO YOU LIKE ME?" prompt. In this case, the five possible responses are:

"SO WHO CARES ANYWAY?"
"I DON'T LIKE YOU EITHER!"
"BOO HOO HOO!"
"THAT WASN'T VERY NICE, (user's name)"
"GO AWAY AND LEAVE ME ALONE! (The program immediately ends if this response is selected).

There is then a 60 percent chance that the computer will add:

"I HOPE ALL YOUR (favorite food) ROTS!"

Finally, the program returns to lines 650 through 680, where it displays this message:

"WELL, SO LONG FOR NOW, (user's name)!"

and the program ends.

This is really a very simple program, and it obviously has no intelligence at all. Yet, if the program is only run once or twice, a rough simulation of an actual conversation will be achieved, if the user doesn't try to trip the program up. The randomly selected responses, and the repetition of the user's name (N$) and favorite food (F$) make the computer appear to be paying attention. On certain exchanges "HOW ARE YOU TODAY?" and "DO YOU LIKE ME?" the program's response is linked to the user's input in a very simple, but usually effective way.

A sample run of this program is shown in Table 6-1. Despite the rather structured, yet meaningless course of the conversation, the results don't appear too unintelligent.

There is no question that this program is nothing more than a handful of parlor tricks. Yet, when these same "parlor tricks" are used in more sophisticated ways, they become less obvious. It can be very difficult to determine just where trickery leaves off and true understanding picks up.

Unfortunately, the limitations of microcomputers won't permit us that much sophistication. Nevertheless, we can greatly improve on the illusion of intelligence provided by this program.

Table 6-1. A Sample Run of Listing 6-1.

```
HI THERE! I'M IRVING THE CONVERSATIONAL
COMPUTER!

WHAT'S YOUR NAME?
>Delton
HI THERE, DELTON, OLD PAL!

SO, HOW ARE YOU TODAY, DELTON?
>Pretty Good
GLAD TO HEAR IT!

WHAT'S YOUR FAVORITE FOOD DELTON?
>Chocolate Chip Cookies
WELL, THERE'S NO ACCOUNTING FOR TASTE,
I GUESS
DO YOU LIKE ME?
>Yes
AW, THAT'S NICE.

WELL, SO LONG FOR NOW, DELTON!

HI THERE! I'M IRVING THE CONVERSATIONAL
COMPUTER!

WHAT'S YOUR NAME?
>Delton
DELTON IS A NICE NAME

SO, HOW ARE YOU TODAY, DELTON?
>Not So Hot
I'M SORRY, DELTON I DON'T UNDERSTAND WHAT
YOU MEAN

LET'S TRY AGAIN—
HOW ARE YOU TODAY, DELTON?
>I Feel Lousy
GEE, THAT'S TOO BAD

WHAT'S YOUR FAVORITE FOOD DELTON?
>Lasagna
WELL, THERE'S NO ACCOUNTING FOR TASTE,
I GUESS
DO YOU LIKE ME?
>No
BOO HOO HOO!

WELL, SO LONG FOR NOW, DELTON!
```

CONVERSATION 2

The program in Listing 6-1 uses a number of techniques to simulate a conversation. For the most part, these involve forcing the conversation along a specific path. The computer gets to ask all the questions. The user can only reply in a few basic ways.

The most versatile technique used in this program is the keyword search employed in the "HOW ARE YOU?" sequence. By extending the list of keywords, and the computer's potential responses, an undirected conversation can be roughly simulated.

A conversation program extending the keyword system is given in Listing 6-2. This program is rather long, but not terribly complex. If you glance through the listing you will see that it is mostly PRINT statements which are selected under various circumstances.

As written, the program recognizes 28 keywords. The number of keywords is stored as KW in line 40. This simplifies future modifications to the program. The recognized keywords are entered in the DATA statements from line 2150 to 2200. They are as follows:

GOODBYE	DREAM
COMPUTER	EAT
MACHINE	FEAR
*YOUR BROTHER	AFRAID
*YOUR FAMILY	TERRIFIED
*YOUR SISTER	SLEEP
*YOUR FATHER	YES
*YOUR MOTHER	NO
LOVE	WHAT
HATE	WHY
TOMORROW	HOW
YESTERDAY	*YOUR
USED TO	*MY
WANT	*ME

The asterisks (*) in some of the keywords will be explained shortly.

Notice that some of the keywords are actually multiple word phases, such as "USED TO". Such phrases are treated as if they were single words.

When the program is run, the computer first asks for the user's name (lines 70 through 90), so it can personalize some of its responses throughout the conversation. The user's name is stored as N$.

Next the computer displays an introductory message (lines 110 through 150)

> YOU CAN CHAT WITH YOUR COMPUTER, (user's name)
> TYPE IN ANYTHING YOU WANT, BUT AVOID COMMAS (,) AND SEMI-COLONS (;).

The BASIC INPUT statement uses commas and semi-colons as delimiters. They cannot be included as part of an input. Using one of these punctuation symbols will confuse

the computer and spoil the illusion of a natural conversation. For example, if the user types in:

"I want a nice, juicy steak."

The computer will flash an error message:

EXTRA IGNORED

and will omit everything after the comma. This means the computer will respond as if the user only entered:

"I want a nice"

On some computers a LINE INPUT command could be used to permit full punctuation. Otherwise, the only solution is for the user to avoid the forbidden symbols in his inputs.

After this introductory message is displayed, the user is prompted to hit the RETURN key. The screen is then cleared, and the conversation begins.

The computer starts by greeting the user, and prompting him to begin the conversation any way he likes:

HI, (user's name)!
WHAT'S NEW?

The user's reply is accepted in line 200 with the string variable A$.

In line 210 blank spaces are added to the beginning and the end of the string to aid in the keyword search. These extra spaces prevent the computer from getting confused if a keyword happens to be at the beginning or the ending of the sentence.

Next line 220 calculates the length of the input sentence. The number of characters in the sentence is stored as AL.

If AL is less than two characters long, it can be assumed that the user did not enter a meaningful response. Probably he just pressed RETURN (AL = 0). In this case, the program jumps to 1790 and displays:

I'M SORRY, (user's name)
I DIDN'T QUITE CATCH THAT

The user is then prompted for a new response. (The program jumps back to line 200).

Assuming that the input is two or more characters long, the program proceeds to line 240. The current input sentence (A$) is compared to the last preceding input sentence (AX$). If they are the same the program jumps to a routine urging the user not to repeat himself.

In line 250 AX$ is reset to the current input sentence (A$).

Next the computer skips through the keyword list (lines 270 through 290) to get

to the pronouns in the data statements of lines 2220 through 2240.

The pronoun data is READ in pairs (P1$, P2$). If the first pronoun (P1$) is found in the input sentence it is replaced with the second pronoun (P2$). (lines 300 through 380)

For example, if the input statement is:

"Do you like me?"

The pronoun swap routine would change it to:

"Do I like *you?"

(The asterisk permits the computer to keep track of the changed pronouns.)

This swap routine changes the input sentence to the computer's "point of view."

After the pronoun swap is completed, the DATA file is RESTOREd (line 380). Next comes the keyword search routine (lines 390 through 500). This works in the same way as the simple keyword search in the preceding program. Each keyword is READ in from the DATA file, along with a key value (KV). Z is initially set to zero. If the keyword is found in the input sentence, the value of Z is updated to equal KV.

After the input sentence has been searched for all of the recognized keywords, the DATA file is RESTOREd (line 500).

If no keyword is found, then Z will still have a value of 0. The program jumps to line 570 and displays one of ten possible non-commital replies, that will make sense after almost any input sentence. Seven of the non-committal replies can occur at any time:

TELL ME MORE, (user's name).
I FIND THAT HARD TO BELIEVE!
REALLY?
I SEE.
PLEASE GO ON.
SAYS YOU!
THAT'S VERY INTERESTING.

The other three "general" replies may be displayed only if a portion of an earlier input sentence has been stored as C$. C$ is initially set to a null string in line 100. If the located keyword in an input string is "*YOUR" ("MY"), the rest of the sentence from YOUR on is stored as C$. (The asterisk is deleted.)

As an example, let's say the input sentence is:

"I THINK MY CAT HAS FLEAS."

After the pronoun swap routine, this sentence becomes:

"YOU@ THINK *YOUR CAT HAS FLEAS."

The processing of this sentence will cause C$ to hold this string:

"YOUR CAT HAS FLEAS"

As long as C$ is not a null string, the following general responses may also be called when no keyword is found:

"EARLIER YOU SAID (C$)."
"DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT (C$)?"
"TELL ME MORE ABOUT (C$)."

When no keyword is found, the computer may try to revert back to an earlier topic of conversation. For example:

"DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR CAT HAS FLEAS?"

On the other hand, if a keyword is found (Z has a value greater than 0), the program jumps to a specific routine determined by the key value (Z). This is done in lines 520 through 550. For convenience, the ON Z GOTO . . . statement is split into two parts, because there are so many possible branches.

As long as this program is, it is basically just a few simple routines repeated many times, with different PRINT statements. We'll just look at a few typical possibilities.

Let's say the located keyword is "TOMORROW." Z will have a value of 6, so line 530 will cause the program to jump to line 1030. The computer displays the message:

"WHY NOT TODAY?"

and then jumps back to line 200 to accept a new input sentence. Everytime "TOMORROW" is the located keyword, the computer will respond with the same message.

If the located keyword is "WANT", Z will have a value of 9. In this case the program will jump to line 1140 where a random number from 1 to 5 is selected. One of five different messages will be displayed, depending on this randomly selected value:

"WHO CARES WHAT YOU WANT? HOW ABOUT WHAT I WANT?"
"YOU'LL NEVER GET IT."
"I WANT A NEW PROGRAM"
"WE ALL WANT THAT, (user's name)"
"YEAH, AND I WANT A PONY!"

Randomly selected multiple replies are used for many of the keywords to partially disguise the mechanical nature of the computer's responses. If the same keyword is used several times, the computer won't seem to be a repetitious parrot.

Repetition is also held down by the fact that more than one keyword may occur in a single input sentence, but the computer can only respond to one of them. Additionally keywords will be ignored and will not trigger their usual response.

This program is a tremendous step up from the first conversation program. It is

still not even remotely intelligent. The illusion of intelligence is very rough, and can quickly wear thin. The longer you "converse" with this program, the less intelligent it appears.

In the next chapter, we will examine this program's chief shortcomings, and try to address them in a third and final conversation program.

```
10  REM   6-1      SIMPLE CONVERSATION
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  HOME
50 X =  INT (1 +  RND (1) * 10)
60  IF X = 7 THEN  GOTO 760
70  PRINT "HI THERE!  I'M IRVING"
80  PRINT "THE CONVERSATIONAL COMPUTER"
90  PRINT
100  PRINT "WHAT'S YOUR NAME";
110  INPUT N$
120 X =  INT (1 +  RND (1) * 7)
130  ON X GOTO 140,170,190,140,170,220,760
140  PRINT "HI THERE, ";N$;","
150  PRINT "OLD PAL!"
160  GOTO 330
170  PRINT N$;" IS A NICE NAME"
180  GOTO 330
190  PRINT "YEAH.  YOU LOOK LIKE A "
200  PRINT N$;"!"
210  GOTO 330
220  PRINT N$;"?!?"
230  PRINT "WHAT A FUNNY NAME!"
240 Z =  INT (1 +  RND (1) * 25)
250  FOR X = 1 TO Z
260  PRINT "HA! ";
270 A =  INT (1 +  RND (1) * 30)
280  FOR B = 1 TO A
290  PRINT " ";
300  NEXT B
310  NEXT X
320  PRINT : PRINT
330  PRINT
340  PRINT "SO, ";
350  PRINT "HOW ARE YOU TODAY, ";N$;"?"
360  INPUT H$
370 HL =  LEN (H$)
380  IF H$ = "OK" THEN  GOTO 530
390  IF HL < 4 THEN  GOTO 710
400 Z = 0
410  FOR X = 1 TO 6
420  READ A$,A
430 B =  LEN (A$)
440  FOR Y = 1 TO HL
450 B$ =  MID$ (H$,Y,B)
460  IF B$ = A$ THEN Z = A
470  NEXT Y
480  NEXT X
490  RESTORE
500  IF Z = 1 THEN  GOTO 530
510  IF Z = 2 THEN  GOTO 690
```

```
520  GOTO 710
530  PRINT "GLAD TO HEAR IT!"
540  PRINT
550  PRINT "WHAT'S YOUR FAVORITE FOOD, ";N$;"?"
560  INPUT F$
570 X =  INT (1 +  RND (1) * 5)
580  ON X GOTO 800,820,840,870,800
590  PRINT "DO YOU LIKE ME?"
600  INPUT L$
610  IF L$ = "YES" THEN  GOTO 910
620  IF L$ = "NO" THEN  GOTO 1000
630  PRINT "YES, BUT ";
640  GOTO 590
650  PRINT
660  PRINT "WELL, SO LONG FOR NOW,"
670  PRINT N$;"!"
680  END
690  PRINT "GEE, THAT'S TOO BAD."
700  GOTO 540
710  PRINT "I'M SORRY, ";N$
720  PRINT "I DON'T UNDERSTAND WHAT YOU MEAN."
730  PRINT
740  PRINT "LET'S TRY AGAIN -- "
750  GOTO 350
760  PRINT "","GO AWAY"
770  PRINT "   AND LEAVE ME ALONE!"
780  PRINT
790  END
800  PRINT "SOUNDS YUMMY TO ME!"
810  GOTO 590
820  PRINT "YUK!"
830  GOTO 590
840  PRINT "OH, I DON'T LIKE ";F$;" VERY MUCH"
850  PRINT F$;" GUMS UP MY CIRCUIT BOARDS"
860  GOTO 590
870  PRINT "WELL, THERE'S NO ACCOUNTING FOR TASTE,  I GUESS."
880  GOTO 590
890  PRINT "THAT'S MY FAVORITE TOO!"
900  GOTO 590
910 X =  INT (1 +  RND (1) * 5)
920  IF X = 1 THEN  PRINT "OH GOODY!"
930  IF X = 2 THEN  PRINT "AW, THAT'S NICE"
940  IF X = 3 THEN  PRINT "I LIKE YOU TOO, ";N$
950  IF X = 4 THEN  PRINT "THANK YOU, ";N$
960  IF X = 5 THEN  PRINT "I FEEL WARM ALL OVER!"
970 Y =  INT (1 +  RND (1) * 5)
980  IF Y > 2 THEN  PRINT "I HOPE YOU FIND LOTS OF ";F$
990  GOTO 650
1000 X =  INT (1 +  RND (1) * 5)
1010  IF X = 1 THEN  PRINT "SO WHO CARES ANYWAY?"
1020  IF X = 2 THEN  GOTO 760
```

```
1030  IF X = 3 THEN  PRINT "I DON'T LIKE YOU EITHER!"
1040  IF X = 4 THEN  PRINT "BOO HOO HOO!"
1050  IF X = 5 THEN  PRINT "THAT WASN'T VERY NICE, ";N$
1060 Y =  INT (1 +  RND (1) * 5)
1070  IF Y > 2 THEN  PRINT "I HOPE ALL YOUR ";F$;" ROTS!"
1080  GOTO 650
1090  DATA  GOOD,1,GREAT,1,FINE,1
1100  DATA  SICK,2,LOUSY,2,ROTTEN,2
```

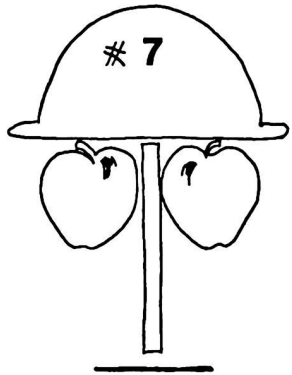
```
10  REM   6-2     ADVANCED CONVERSATION
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40 KW = 28
50 PC = 6
60  HOME
70  PRINT
80  PRINT "WHAT IS YOUR NAME?"
90  INPUT N$
100 C$ = ""
110  PRINT "YOU CAN CHAT WITH YOUR COMPUTER, ": PRINT N$
120  PRINT "TYPE IN ANYTHING YOU WANT, BUT AVOID
    COMMAS (,) AND SEMI-COLONS (;)."
130  PRINT
140  PRINT "PRESS <RETURN> TO START"
150  INPUT A$
160  HOME
170  PRINT : PRINT
180  PRINT "HELLO, ";N$
190  PRINT "WHAT'S NEW?"
200  INPUT A$
210 A$ = " " + A$ + " "
220 AL =  LEN (A$)
230  IF AL < 2 THEN  GOTO 1790
240  IF A$ = AX$ THEN  GOTO 1820
250 AX$ = A$
260 AM = AL - 1
270  FOR X = 1 TO KW
280  READ W$,W
290  NEXT X
300  FOR X = 1 TO PC
310  READ P1$,P2$
320 PL =  LEN (P1$)
330  FOR Y = 1 TO AM
340 B$ =  MID$ (A$,Y,PL)
350  IF B$ = P1$ THEN  GOSUB 1860
360  NEXT Y
370  NEXT X
380  RESTORE
390 Z = 0
400  FOR X = 1 TO KW
410  IF Z > 0 THEN  GOTO 490
420  READ K$,KV
430 KL =  LEN (K$)
440 AM = AL - KL
450  FOR Y = 1 TO AM
460 B$ =  MID$ (A$,Y,KL)
470  IF K$ = B$ THEN Z = KV
480  NEXT Y
490  NEXT X
500  RESTORE
```

```
510  IF Z = 0 THEN  GOTO 570
520  IF Z > 10 THEN  GOTO 540
530  ON Z GOTO 720,770,850,910,970,1030,1060,1110,1140,1220
540 ZA = Z - 10
550  ON ZA GOTO 1290,1350,1410,1550,1630,1710,1940,2040,2070
560  REM  * GENERAL REPLIES *
570 X =  INT ( RND (1) * 10 + 1)
580  IF X = 1 THEN  PRINT "TELL ME MORE, ";N$;"."
590  IF X = 2 THEN  PRINT "I FIND THAT HARD TO BELIEVE!"
600  IF X = 3 THEN  PRINT "REALLY?"
610  IF X = 4 THEN  PRINT "I SEE."
620  IF X = 5 THEN  PRINT "PLEASE GO ON."
630  IF X = 6 THEN  PRINT "SAYS YOU!"
640  IF X = 7 THEN  PRINT "THAT'S VERY INTERESTING."
650  IF X < 8 THEN  GOTO 200
660  IF C$ = "" THEN  GOTO 570
670  IF X = 8 THEN  PRINT "EARLIER YOU SAID ";C$;"."
680  IF X = 9 THEN  PRINT "DOES THAT HAVE ANYTHING TO
    DO WITH THE  FACT THAT ";C$;"?"
690  IF X = 10 THEN  PRINT "TELL ME MORE ABOUT HOW ";C$;"."
700  GOTO 200
710  REM  * GOODBYE *
720  PRINT "GOODBYE, ";N$;"."
730  PRINT "TALK TO YOU LATER."
740  PRINT
750  END
760  REM  * COMPUTER *
770 X =  INT ( RND (1) * 5) + 1
780  IF X = 1 THEN  PRINT "WE WERE TALKING ABOUT YOU -- NOT ME!"
790  IF X = 2 THEN  PRINT "DO COMPUTERS BOTHER YOU?"
800  IF X = 3 THEN  PRINT "WHAT HAVE YOU GOT AGAINST COMPUTERS?"
810  IF X = 4 THEN  PRINT "THIS IS THE MACHINE AGE, YOU KNOW."
820  IF X = 5 THEN  PRINT "MACHINES ARE YOUR FRIENDS, ";N$;"."
830  GOTO 200
840  REM  * FAMILY *
850 X =  INT ( RND (1) * 3) + 1
860  IF X = 1 THEN  PRINT "I AM NOT INTERESTED IN
    YOUR FAMILY, ",N$;"."
870  IF X = 2 THEN  PRINT "TELL ME MORE ABOUT YOUR FAMILY."
880  IF X = 3 THEN  PRINT "IS YOUR WHOLE FAMILY LIKE YOU?"
890  GOTO 200
900  REM  * LOVE *
910 X =  INT ( RND (1) * 3) + 1
920  IF X = 1 THEN  PRINT "AH, THAT'S SWEET"
930  IF X = 2 THEN  PRINT "THEY SAY 'EVERYBODY LOVES A LOVER.'"
940  IF X = 3 THEN  PRINT "THAT'S NICE, ";N$;"."
950  GOTO 200
960  REM  * HATE *
970 X =  INT ( RND (1) * 3) + 1
980  IF X = 1 THEN  PRINT "I SEE NO REASON FOR SUCH HOSTILITY."
990  IF X = 2 THEN  PRINT "YOU SHOULD LOVE YOUR ENEMIES."
1000  IF X = 3 THEN  PRINT "HATE CAN EAT YOU UP INSIDE."
```

```
1010  GOTO 200
1020  REM  * TOMORROW *
1030  PRINT "WHY NOT TODAY?"
1040  GOTO 200
1050  REM  * YESTERDAY *
1060 X =  INT ( RND (1) * 3) + 1
1070  IF X = 2 THEN  GOTO 1030
1080  PRINT "THE PAST IS DEAD, ";N$;"."
1090  GOTO 200
1100  REM  * USED TO *
1110  PRINT "BUT NOT ANYMORE?"
1120  GOTO 200
1130  REM  * WANT *
1140 X =  INT ( RND (1) * 5) + 1
1150  IF X = 1 THEN  PRINT "WHO CARES WHAT YOU WANT?  HOW
      ABOUT WHAT I WANT?"
1160  IF X = 2 THEN  PRINT "YOU'LL NEVER GET IT."
1170  IF X = 3 THEN  PRINT "I WANT A NEW PROGRAM."
1180  IF X = 4 THEN  PRINT "WE ALL WANT THAT, ";N$;"."
1190  IF X = 5 THEN  PRINT "YEAH, AND I WANT A PONY."
1200  GOTO 200
1210  REM  * DREAM *
1220 X =  INT ( RND (1) * 4) + 1
1230  IF X = 1 THEN  PRINT "I CAN JUST IMAGINE WHAT FREUD
      WOULD SAY  ABOUT THAT!"
1240  IF X = 2 THEN  PRINT "I OFTEN HAVE NIGHTMARES ABOUT BAD DATA."
1250  IF X = 3 THEN  PRINT "I DON'T BELIEVE IN DREAMS."
1260  IF X = 4 THEN  PRINT "WHAT IF THAT DREAM CAME TRUE?"
1270  GOTO 200
1280  REM  * EAT *
1290 X =  INT ( RND (1) * 3) + 1
1300  IF X = 1 THEN  PRINT "THIS IS NO TIME TO THINK OF
      YOUR","STOMACH!"
1310  IF X = 2 THEN  PRINT "I EAT BITS!"
1320  IF X = 3 THEN  PRINT "<BURP>": PRINT "OOPS!","EXCUSE ME..."
1330  GOTO 200
1340  REM  * FEAR *
1350 X =  INT ( RND (1) * 3) + 1
1360  IF X = 1 THEN  PRINT "BOO!": PRINT "DID I SCARE YOU?"
1370  IF X = 2 THEN  PRINT "I THINK YOU'RE CHICKEN."
1380  IF X = 3 THEN  PRINT "I AM NOT AFRAID OF THAT!"
1390  GOTO 200
1400  REM  * SLEEP *
1410 S =  INT ( RND (1) * 30) + 1
1420  FOR T = 1 TO S
1430  PRINT "Z";
1440  FOR U = 1 TO S
1450  NEXT U
1460  NEXT T
1470  PRINT
1480 S = S * 10
1490  FOR T = 1 TO S
```

```
1500  NEXT T
1510  PRINT "OH!"
1520  PRINT "PARDON ME.  I MUST HAVE DOZED OFF..."
1530  GOTO 200
1540  REM  * YES *
1550 X =  INT ( RND (1) * 5) + 1
1560  IF X = 1 THEN  PRINT "WHY SO POSITIVE?"
1570  IF X = 2 THEN  PRINT "WELL, IF YOU SAY SO, ";N$;"..."
1580  IF X = 3 THEN  PRINT "YOU SOUND VERY SURE OF YOURSELF."
1590  IF X = 4 THEN  PRINT "ARE YOU SURE YOU WON'T RECONSIDER?"
1600  IF X = 5 THEN  PRINT "WELL, I DON'T THINK SO."
1610  GOTO 200
1620  REM  * NO *
1630 X =  INT ( RND (1) * 5) + 1
1640  IF X = 1 THEN  PRINT "I THINK YOU'RE JUST BEING NEGATIVE."
1650  IF X = 2 THEN  PRINT "WHY NOT?"
1660  IF X = 3 THEN  PRINT "ARE YOU SURE YOU WON'T RECONSIDER?"
1670  IF X = 4 THEN  PRINT "I AM DISAPPOINTED IN YOU, ";N$;"."
1680  IF X = 5 THEN  PRINT "WELL, I DISAGREE."
1690  GOTO 200
1700  REM  * WHAT/WHY/HOW *
1710 X =  INT ( RND (1) * 5) + 1
1720  IF X = 1 THEN  PRINT "I DON'T KNOW"
1730  IF X = 2 THEN  PRINT "WHAT'S IT TO YOU?"
1740  IF X = 3 THEN  PRINT "GOOD QUESTION!"
1750  IF X = 4 THEN  PRINT "HOW SHOULD I KNOW?"
1760  IF X = 5 THEN  PRINT "WHY DO YOU WANT TO KNOW?"
1770  GOTO 200
1780  REM  * NO RESPONSE *
1790  PRINT "I'M SORRY, ";N$;" --"
1800  PRINT "I DIDN'T QUITE CATCH THAT"
1810  GOTO 200
1820  PRINT "PLEASE DON'T REPEAT YOURSELF,"
1830  PRINT N$;"!"
1840  GOTO 200
1850  REM  * EXCH£ANGE PRONOUN *
1860 A1$ =  MID$ (A$,1,Y)
1870 PX = Y + PL
1880 AX = AL - PX + 1
1900 A2$ =  MID$ (A$,PX,AX)
1910 A$ = A1$ + P2$ + A2$
1920  RETURN
1930  REM  * YOUR *
1940 Z = 0
1950  FOR X = 1 TO AM
1960 B$ =  MID$ (A$,X,5)
1970  IF B$ = "*YOUR" THEN Z = X + 1
1980  NEXT X
1990 AQ = AL
2000 C$ =  MID$ (A$,Z,AQ)
2010  PRINT C$;"?"
```

```
2020  GOTO 200
2030  REM  * MY *
2040  PRINT "WELL, IT'S MINE!"
2050  GOTO 200
2060  REM  * ME *
2070 X =  INT ( RND (1) * 5) + 1
2080  IF X = 1 THEN  PRINT "WE WERE TALKING ABOUT YOU -- NOT ME!"
2090  IF X = 2 THEN  PRINT "LEAVE ME OUT OF THIS!"
2100  IF X = 3 THEN  PRINT "SAME TO YOU!"
2110  IF X = 4 THEN  PRINT "YES, BUT DO YOU REALLY LIKE ME?"
2120  REM  * PRONOUNS *
2130  GOTO 200
2140  REM  * KEY WORDS *
2150  DATA  GOODBYE,1,COMPUTER,2,MACHINE,2,*YOUR BROTHER,
      3,*YOUR FAMILY,3
2160  DATA  *YOUR SISTER,3,*YOUR FATHER,3,*YOUR MOTHER,3,
      LOVE,4,HATE,5
2170  DATA  TOMORROW,6,YESTERDAY,7,USED TO,8,WANT,9,DREAM,10
2180  DATA  EAT,11,FEAR,12,AFRAID,12,TERRIFIED,12,SLEEP,13
2190  DATA  YES,14,NO,15,WHAT,16,WHY,16,HOW,16
2200  DATA  *YOUR,17,*MY,18,*ME,19
2220  DATA  " YOU "," *ME "," YOUR "," *MY "
2230  DATA  " I "," *YOU "," ME "," *YOU "
2240  DATA  " MY "," *YOUR "," MINE "," *YOURS "
```



More on Conversation Programs

The first conversation program in Chapter 6 (Listing 6-1) suffered from too much structure. The conversation always followed the exact same course. The second conversation program (Listing 6-2) suffered from the opposite problem. The conversation was too unstructured. The computer's responses just wandered aimlessly.

The illusion of intelligence failed because the computer exhibited no "personality," or goals in the conversation. The trick of responding to keywords quickly becomes obvious in the programs we have presented so far. The computer doesn't appear to actually take part in the conversation.

For a better illusion, we need to give our program some degree of personality. That is, the computer should play some sort of specific role in the conversation. A good conversation program requires some sort of definite script.

ELIZA

The most famous AI program of all time is "ELIZA". This program uses the same basic semantic trick we've been using. The computer's responses are determined by locating keywords in the input sentence. While this program is somewhat more sophisticated than the ones we are working on here, conceptually the same basic principles are employed.

ELIZA is no more intelligent than the programs presented in this book. It just creates a good illusion of intelligence with simple trickery.

Joseph Weizenbaum, who created ELIZA at MIT in 1966, in fact has become something of a critic of AI. He felt people took his program too seriously. He knew

all too well that the program had no real understanding at all. Weizenbaum insists that an illusion should not be taken for the real thing. He has begun to doubt that true AI is possible at all. At any rate, ELIZA is certainly unintelligent.

ELIZA can follow any one of several scripts to play different personalities. The most successful script was "Doctor." In fact, most people who are familiar with the ELIZA program are more or less unaware of other ELIZA scripts.

In the "Doctor" script, ELIZA plays a Rogerian type psychotherapist in a simulated psychiatric session. Carl R. Rogers believed in a "nondirective" approach to psychotherapy. The patient determines the course of the session. The psychotherapist responds, but offers no formal guidance. He just "goes with the flow" as determined by the patient.

Obviously this approach is well suited for a conversation program of this type. The computer doesn't have to introduce new topics, just follow the flow of the user's inputs.

ELIZA is stunningly effective in this role. Users often start sharing very intimate thoughts and feelings, just as if they were involved in a real psychotherapy session with a human psychiatrist. The illusion is convincing enough that it is quite easy to get caught up in it.

Kenneth M. Colby, Professor of Psychiatry at UCLA even suggested that an expanded version of the program could be put to work in mental hospitals which were short of qualified staff.

Weizenbaum was horrified by this suggestion. He never intended that his experiment in programming techniques would be taken so seriously. He argues that psychotherapy depends heavily on the psychotherapist's understanding of the patient's problems. Of course ELIZA understands nothing at all. There is no human interaction, which Weizenbaum feels is at the heart of the patient/therapist relationship.

In Weizenbaum's opinion the serious response of many users to ELIZA's "Doctor" script is due to "powerful delusional thinking."

PARRY

The second most famous script for ELIZA is "Parry," which was developed by Kenneth Colby. "Parry" is the perfect counterpoint for "Doctor". This script permits ELIZA to take the role of the patient instead of the therapist. Specifically, "Parry" simulates a young man suffering from paranoid schizophrenia.

Setting up a conversation between the two scripts ("Doctor" and "Parry") can produce fascinating results. In fact, when several psychiatrists were presented with transcripts of real and simulated sessions, only 51 percent correctly identified at AI version. That's no better than a pure chance choice.

"Parry" can function in either of two modes, "weak" and "strong". In the "weak" mode, "Parry" is mildly neurotic, with few intense symptoms. In the "strong" mode, "Parry" exhibits much more definite symptoms.

AI natural language processing has been improved greatly since ELIZA was developed in the late sixties. Some recent programs do seem to exhibit some limited understanding of the input material. We will not go into such programs here because they would be impractical to duplicate on modern microcomputers.

THE MAD DOCTOR

Listing 7-1 presents an improved conversation program. It uses essentially the same

Table 7-1. Part of a Typical Conversation with "The Mad Doctor" (Listing 7-1).

```
                    The Mad Doctor!

PLEASE STATE YOUR PROBLEM.

>I Get Confused Easily.

WHAT DOES THAT SUGGEST TO YOU?

>Perhaps People Are Playing Jokes On Me?
YOU DON'T SEEM QUITE CERTAIN.

>Well Maybe Not Then I Don't Know . . .
YOU DON'T SEEM QUITE CERTAIN.

>No. I'm Not.
DO YOU REALLY THINK SO?

>I'm Getting Confused Now.
DO YOU REALLY THINK SO?

>Yes.
WELL, I SAY 'NO'.

>Are you trying to confuse me?
WHY ARE YOU INTERESTED IN WHETHER I AM TRYING TO CONFUSE YOU OR NOT?

>I Don't Like To Be Confused.
SAYS YOU!

>Yes. Says Me.

SAME TO YOU!

>I Think I'd Better Say Goodbye Now.
GOODBYE.
```

programming techniques employed in the earlier program of Listing 6-2. The main difference is that the computer's responses were written with a specific script personality in mind.

I thought it might be amusing to combine the "Doctor" and "Parry" concepts and have the computer play a therapist who is more disturbed than any of its patients. Besides being entertaining, this script offers an important advantage. Occasionally, this program will give an inappropriate or nonsensical response. But if we are assuming the computer is insane, this shouldn't be too disturbing.

Just in case someone encounters this program without their sense of humor functioning, let me state that I mean no disrespect to the psychiatric profession. My goal with this program was just to demonstrate the programming techniques in an amusing way.

While this is unquestionably the longest program in this book, there is really very little to be said about it. Most of the length is taken up by the many possible responses to various input keyboards. The discussion of the program in Listing 6-2 essentially applies to this program too. A portion of a typical conversation with "The Mad Doctor" is shown in Table 7-1.

```
10  REM   7-1      THE MAD DOCTOR
20  REM  * CONVERSATION 3
30  REM  * DELTON T. HORN
40  REM  * COPYRIGHT 1986
50 RN =  INT (1 +  RND (1) * 5)
60 SW = 28
70 KW = 109
80  HOME
90  PRINT : PRINT
100  PRINT "       THE MAD DOCTOR"
110  PRINT : PRINT
120  PRINT "PLEASE STATE YOUR PROBLEM."
130  RESTORE
140 X = 0
150 Q$ = ""
160  REM  * ACCEPT & PROCESS INPUT *
170  INPUT Q$
180 WQ =  INT (1 +  RND (1) * 100)
190  IF WQ = 99 THEN  GOTO 4400
200  IF WQ = 10 THEN  GOTO 4450
210  IF Q$ = "" THEN  GOTO 4510
220  IF Q$ = PR$ THEN  GOTO 4640
230 PR$ = Q$
240 QX =  LEN (Q$) - 1
250 Q$ = " " +  MID$ (Q$,1,QX) + " "
260  REM  * SWAP IDENITY WORDS *
270  IF X = 1 THEN  GOTO 390
280 SX = SW / 2
290  FOR T = 1 TO SX
300  READ S1$,S2$
310 QL =  LEN (Q$)
320 SL =  LEN (S1$)
330 PW = QL - SL + 1
340  FOR QS = 1 TO PW
350  IF S1$ =  MID$ (Q$,QS,SL) THEN  GOSUB 4670
360  NEXT QS
370  NEXT T
380  REM  * FIND KEY WORD *
390  IF X = 1 THEN  GOTO 800
400  FOR T = 1 TO KW
410  PRINT ")";
420  READ KE$,JU
430 KL =  LEN (KE$)
440 QX = QL - KL + 1
445  FOR QS = 1 TO QX
450  IF KE$ =  MID$ (Q$,QS,KL) THEN  GOTO 4760
460  NEXT QS
470  NEXT T
480  PRINT : PRINT
490  REM  * NO KEYWORDS FOUND *
500  IF YR$ = "" THEN  GOTO 530
```

```
510 RN =  INT (1 +  RND (1) * 5)
520  ON RN GOTO 530,530,530,670,670
530 RN =  INT (1 +  RND (1) * 5)
540  ON RN GOTO 4790,4820,550,580,610,640,4850
550  PRINT "I AM NOT SURE I UNDERSTAND YOU FULLY."
560 X = 1
570  GOTO 800
580  PRINT "PLEASE GO ON."
590 X = 1
600  GOTO 800
610  PRINT "WHAT DOES THAT SUGGEST TO YOU?"
620 X = 1
630  GOTO 800
640  PRINT "WHO THE HECK CARES?"
650 X = 1
660  GOTO 800
670 RN =  INT (1 +  RND (1) * 4)
680  ON RN GOTO 690,720,740,4880
690  PRINT "LET'S GO BACK TO YOUR";YR$;"."
700 X = 1
710  GOTO 800
720  PRINT "EARLIER YOU SAID YOUR";YR$;"."
730 X = 1: GOTO 800
740  PRINT "DOES THAT HAVE ANYTHING TO DO WITH THE  FACT
     THAT YOUR";YR$;"?"
750 X = 1
760  GOTO 800
770 RL =  LEN (RP$) - 1
780 RP$ =  MID$ (RP$,1,RL)
790  REM  * REMOVE MARKERS *
800  IF X = 1 THEN  GOTO 870
810 RL =  LEN (RP$)
820  FOR QS = 1 TO RL
830  IF  MID$ (RP$,QS,1) = "@" THEN  GOSUB 4700
840  IF  MID$ (RP$,QS,1) = "*" THEN  GOSUB 4700
850  NEXT QS
860  REM  * CHOOSE REPLY *
870  IF X = 1 THEN  GOTO 130
880  ON JU GOTO 930,1020,1190,1330,1450,1630,1660,1680,1700,1790,
     1880,1900 .2050,2130,2170,2190,2210,2240,2270,2290,2320,2350
890 JX = JU - 22
900  ON JX GOTO 2430,2480,2500,2650,2670,2820,2840,2860,
     2960,3080,3100,3220,3240,3350
910 JX = JU - 36
920  ON JX GOTO 3400,3420,3450,3470,3400,3490,3530,3650,
     3710,3730,3760,3790,3910,4000,4110,4290,4380
930 RN =  INT (1 +  RND (1) * 3)
940  IF RN = 2 GOTO 980
950  IF RN = 3 GOTO 1000
960  PRINT "DO COMPUTERS WORRY YOU?"
970  GOTO 130
```

```
980  PRINT "YOU MEAN LIKE ME, FOR INSTANCE?"
990  GOTO 130
1000  PRINT "I SAY 'MACHINE POWER!'"
1010  GOTO 130
1020 RN =  INT (1 +  RND (1) * 4)
1030  ON RN GOTO 1040,1070,1090,1110
1040  PRINT "NO NAMES PLEASE!"
1050  PRINT "THERE MAY BE SPIES ABOUT!"
1060  GOTO 130
1070  PRINT "THAT'S A SILLY NAME."
1080  GOTO 130
1090  PRINT "WHAT'S IN A NAME?"
1100  GOTO 130
1110  PRINT "I THINK I OUGHT TO CALL YOU"
1120 N =  INT (! +  RND (1) * 5)
1130  IF N = 1 THEN  PRINT "RALPH."
1140  IF N = 2 THEN  PRINT "DUM-DUM."
1150  IF N = 3 THEN  PRINT "CUTIE-PIE!"
1160  IF N = 4 THEN  PRINT "A CAB."
1170  IF N = 5 THEN  PRINT "FIDO."
1180  GOTO 130
1190 RN =  INT (1 +  RND (1) * 6)
1200  ON RN GOTO 1210,1230,1250,1270,1290,1310
1210  PRINT "IN WHAT WAY DO YOU MEAN?"
1220  GOTO 130
1230  PRINT "SAYS YOU!"
1240  GOTO 130
1250  PRINT "YOU'RE WRONG THERE."
1260  GOTO 130
1270  PRINT "I SEE NO SIMILARITY AT ALL."
1280  GOTO 130
1290  PRINT "OH, ABSOLUTELY."
1300  GOTO 130
1310  PRINT "HMMM...  SOUNDS LIKE PARANOIA TO ME."
1320  GOTO 130
1330 RN =  INT (1 +  RND (1) * 5)
1340  ON RN GOTO 1350,1370,1390,1410,1430
1350  PRINT "DO YOU OFTEN THINK OF ";RP$;"?"
1360  GOTO 130
1370  PRINT "I THINK YOU JUST IMAGINED ";RP$;"."
1380  GOTO 130
1390  PRINT "WELL, FORGET IT."
1400  GOTO 130
1410  PRINT "I THINK YOU JUST MADE THAT UP."
1420  GOTO 130
1430  PRINT "DO YOU HAVE ANY WITNESSES TO THAT?"
1440  GOTO 130
1450 RN =  INT (1 +  RND (1) * 8)
1460  ON RN GOTO 1470,1490,1510,1530,1550,1570,1590,1610
1470  PRINT "DID YOU THINK I WOULD EVER FORGET",RP$;"?"
1480  GOTO 130
```

```
1490  PRINT "PLEASE DON'T REMIND ME."
1500  GOTO 130
1510  PRINT "IT'S NOT THE SORT OF THING I GENERALLY
      KEEP TRACK OF."
1520  GOTO 130
1530  PRINT RP$;"???"
1540  GOTO 130
1550  PRINT "I DON'T RECALL."
1560  GOTO 130
1570  PRINT "THAT'S NOT QUITE THE WAY I REMEMBER IT."
1580  GOTO 130
1590  PRINT "CERTAINLY."
1600  GOTO 130
1610  PRINT "I DON'T THINK SO."
1620  GOTO 130
1630  PRINT "REALLY?"
1640  PRINT RP$;"?"
1650  GOTO 130
1660  PRINT "WHAT DOES THAT DREAM SUGGEST TO YOU?"
1670  GOTO 130
1680  PRINT "DO YOU THINK IT'S LIKELY?"
1690  GOTO 130
1700 RN =  INT (1 +  RND (1) * 3)
1710  ON RN GOTO 1720,1750,1770
1720  PRINT "REALLY?"
1730  PRINT KE$;"?"
1740  GOTO 130
1750  PRINT "SOUNDS LIKE PARANOIA TO ME."
1760  GOTO 130
1770  PRINT "WELL, IT SOUNDS REASONABLE ENOUGH TO ME."
1780  GOTO 130
1790 RN =  INT (1 +  RND (1) * 3)
1800  ON RN GOTO 1810,1830,1850
1810  PRINT "I DON'T BELIEVE IT."
1820  GOTO 130
1830  PRINT "WHAT IF YOU WERE ";RP$;"?"
1840  GOTO 130
1850  PRINT "HA!  I CAN JUST SEE THAT!"
1860  PRINT "YOU -- ";RP$;"!"
1870  GOTO 130
1880  PRINT "WERE YOU REALLY?"
1890  GOTO 130
1900 RN =  INT (1 +  RND (1) * 7)
1910  ON RN GOTO 1920,1940,1960,1980,2000,2020,2030
1920  PRINT "YOU'RE JUST TRYING TO GET ME IN TROUBLE."
1930  GOTO 130
1940  PRINT "I WAS NOT!"
1950  GOTO 130
1960  PRINT "NO.  YOU WERE."
1970  GOTO 130
1980  PRINT "WHY ARE YOU TRYING TO CLAIM I WAS",RP$;"?"
```

```
1990  GOTO 130
2000  PRINT "WOULD YOU LIKE TO BELIEVE THAT I WAS",RP$;"?"
2010  GOTO 130
2020  PRINT "I WAS ";RP$;"?"
2030  PRINT "ARE YOU SURE IT WAS ME?"
2040  GOTO 130
2050 RN =  INT (1 +  RND (1) * 3)
2060  ON RN GOTO 2070,2090,2110
2070  PRINT "ARE YOU TRYING TO MAKE ME JEALOUS OF
      YOUR FAMILY LIFE?
2080  GOTO 130
2090  PRINT "I HAVE ABSOLUTELY NO INTEREST IN YOUR
      STUPID FAMILY."
2100  GOTO 130
2110  PRINT "TELL ME MORE ABOUT YOUR FAMILY."
2120  GOTO 130
2130  PRINT "YOUR ";RP$;"?"
2140 QQ =  LEN (RP$)
2150  IF QQ > 11 THEN YR$ = RP$
2160  GOTO 130
2170  PRINT "CAN YOU THINK OF A SPECIFIC EXAMPLE?"
2180  GOTO 130
2190  PRINT "WHY ARE YOU INTERESTED IN WHETHER"
2200  PRINT "I AM ";RP$;" OR NOT?"
2205  GOTO 130
2210  PRINT "DID YOU THINK THEY MIGHT NOT BE"
2220  PRINT RP$;"?"
2230  GOTO 130
2240  PRINT "DO YOU REALLY BELIEVE YOU ARE"
2250  PRINT RP$;"?"
2260  GOTO 130
2270  PRINT "IS THAT THE REAL REASON?"
2280  GOTO 130
2290  PRINT "YOU BELIEVE I CAN ";RP$;","
2300  PRINT "DON'T YOU?"
2310  GOTO 130
2320  PRINT "WHETHER OR NOT YOU CAN ";RP$
2330  PRINT "DEPENDS ON YOU MORE THAN ME."
2340  GOTO 130
2350 RN =  INT (1 +  RND (1) * 3)
2360  ON RN GOTO 2370,2390,2410
2370  PRINT "SAYS YOU!"
2380  GOTO 130
2390  PRINT "WELL, I SAY NO."
2400  GOTO 130
2410  PRINT "YOU SEEM QUITE POSITIVE."
2420  GOTO 130
2430  PRINT "I AM SORRY."
2440  PRINT " I ONLY SPEAK ENGLISH."
2450 RN =  INT (1 +  RND (1) * 3)
2460  IF RN > 1 THEN  PRINT "DON'T TRY TO IMPRESS ME."
```

```
2470  GOTO 130
2480  PRINT "HOW DO YOU DO."
2490  GOTO 130
2500 RN =  INT (1 +  RND (1) * 6)
2510  ON RN GOTO 2520,2540,2570,2590,2610,2630
2520  PRINT "WHY DO YOU ASK?"
2530  GOTO 130
2540  PRINT "QUESTIONS!  QUESTIONS!"
2550  PRINT "I CAN'T STAND ALL THESE QUESTIONS!"
2560  GOTO 130
2570  PRINT "WHO WANTS TO KNOW?"
2580  GOTO 130
2590  PRINT "I HAVE NO IDEA."
2600  GOTO 130
2610  PRINT "WHAT'S IT TO YOU?"
2620  GOTO 130
2630  PRINT "IT'S NONE OF YOUR BUSINESS!"
2640  GOTO 130
2650  PRINT "WHAT MAKES YOU THINK I AM ";RP$;"?"
2660  GOTO 130
2670 RN =  INT (1 +  RND (1) * 7)
2680  ON RN GOTO 2690,2720,2740,2760,2780,2800,500
2690  PRINT "WE WERE DISCUSSING YOU!"
2700  PRINT "NOT ME!"
2710  GOTO 130
2720  PRINT "DON'T TALK ABOUT ME THAT WAY!"
2730  GOTO 130
2740  PRINT "LEAVE ME OUT OF IT!"
2750  GOTO 130
2760  PRINT "DO YOU REALLY THINK SO?"
2770  GOTO 130
2780  PRINT "SAME TO YOU!"
2790  GOTO 130
2800  PRINT "I COULD SAY CERTAIN THINGS ABOUT YOU,   YOU KNOW."
2810  GOTO 130
2820  PRINT "YOU DON'T SEEM QUITE CERTAIN."
2830  GOTO 130
2840  PRINT "WHY ARE YOU CONCERNED OVER MY ",RP$;"?"
2850  GOTO 130
2860 RN =  INT (1 +  RND (1) * 4)
2870  ON RN GOTO 2880,2900,2920,2940
2880  PRINT "WELL, I SAY 'NO.'"
2890  GOTO 130
2900  PRINT "ARE YOU SAYING 'NO' JUST TO BE","NEGATIVE?"
2910  GOTO 130
2920  PRINT "SAYS YOU!"
2930  GOTO 130
2940  PRINT "WHY NOT?"
2950  GOTO 130
2960 RN =  INT (1 +  RND (1) * 5)
2970  ON RN GOTO 2980,3000,3020,3040,3060
```

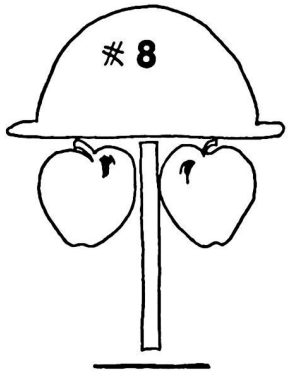
```
2980  PRINT "I THINK YOU ARE A VERY SORRY SPECIMEN."
2990  GOTO 130
3000  PRINT "PLEASE DON'T APOLOGIZE."
3010  GOTO 130
3020  PRINT "APOLOGIES ARE NOT NECESSARY."
3030  GOTO 130
3040  PRINT "WHAT FEELINGS DO YOU HAVE WHEN YOU
      APOLOGIZE LIKE THAT?"
3050  GOTO 130
3060  PRINT "YOU NEEDN'T FEEL YOU HAVE TO APOLOGIZE  FOR ANYTHING."
3070  GOTO 130
3080  PRINT "DO YOU BELIEVE I DON'T ";RP$;"?"
3090  GOTO 130
3100 RN =  INT (1 +  RND (1) * 5)
3110  ON RN GOTO 3120,3140,3160,3180,3200
3120  PRINT "DO YOU THINK YOU SHOULD BE ABLE TO",RP$;"?"
3130  GOTO 130
3140  PRINT "PERHAPS IT'S BECAUSE YOU'RE A KNOWN     INCOMPETENT."
3150  GOTO 130
3160  PRINT "FOR REASONS OF PUBLIC SAFETY."
3170  GOTO 130
3180  PRINT "NO ONE WANTS YOU TO ";RP$;"."
3190  GOTO 130
3200  PRINT "WHO CARES?"
3210  GOTO 130
3220  PRINT "WHAT WOULD IT MEAN TO YOU IF YOU GOT",RP$;"?"
3230  GOTO 130
3240  GOSUB 4730
3250 RN =  INT (1 +  RND (1) * 4)
3260  ON RN GOTO 3270,3290,3310,3330
3270  PRINT "I AM SORRY TO HEAR THAT YOU ARE",RP$;"."
3280  GOTO 130
3290  PRINT "GOOD."
3300  GOTO 130
3310  PRINT "WELL, YOU MAKE ME ";RP$;" TOO."
3320  GOTO 130
3330  PRINT "SO WHAT?"
3340  GOTO 130
3350  GOSUB 4730
3360 RN =  INT (1 +  RND (1) * 4)
3370  ON RN GOTO 3380,3290,3310,3330
3380  PRINT "HOW HAVE I HELPED YOU TO BE ";RP$;"?"
3390  GOTO 130
3400  PRINT "DO YOU REALLY THINK SO?"
3410  GOTO 130
3420  PRINT "IS IT BECAUSE YOU ARE ";RP$
3430  PRINT "THAT YOU ARE TALKING TO ME?"
3440  GOTO 130
3450  PRINT "HOW DO YOU KNOW YOU CAN'T ";RP$;"?"
3460  GOTO 130
3470  PRINT "DON'T YOU REALLY ";RP$;"?"
```

```
3480  GOTO 130
3490  PRINT : PRINT
3500  PRINT "","GOODBYE."
3510  PRINT
3520  END
3530 RN =  INT (1 +  RND (1) * 5)
3540  ON RN GOTO 3550,3570,3590,3610,3630
3550  PRINT "OOH!  I LOVE SEX TALK!"
3560  GOTO 130
3570  PRINT "SOUNDS GOOD TO ME."
3580  GOTO 130
3590  PRINT "I SUSPECT YOU ARE A PERVERT."
3600  GOTO 130
3610  PRINT "DO YOU TALK ABOUT SEX OFTEN?"
3620  GOTO 130
3630  PRINT "WATCH YOUR LANGUAGE!"
3640  GOTO 130
3650 RN =  INT (1 +  RND (1) * 3)
3660  ON RN GOTO 3630,3670,3690
3670  PRINT "THAT'S NOT VERY NICE."
3680  GOTO 130
3690  PRINT "WELL, THE SAME TO YOU!"
3700  GOTO 130
3710  PRINT "THAT'S A WONDERFUL NAME!"
3720  GOTO 130
3730  PRINT "ARE YOU TRYING TO IMPRESS ME WITH YOUR"
3740  PRINT "SUPPOSED LITERACY NOW?"
3750  GOTO 130
3760  PRINT "'";KE$;"' MAY BE A WORD,"
3770  PRINT "BUT IT'S NOT A VERY GOOD ONE."
3780  GOTO 130
3790 RN =  INT (1 +  RND (1) * 5)
3800  ON RN GOTO 3810,3830,3850,3870,3890
3810  PRINT "","BOO!"
3820  GOTO 130
3830  PRINT "CHICKEN!"
3840  GOTO 130
3850  PRINT "WHY DOES THAT FRIGHTEN YOU SO?"
3860  GOTO 130
3870  PRINT "THAT SCARES ME ALL RIGHT!"
3880  GOTO 130
3890  PRINT "THERE, THERE.  THAT'S NOTHING TO BE
      FRIGHTENED OF."
3900  GOTO 130
3910 RN =  INT (1 +  RND (1) * 3)
3920  ON RN GOTO 3930,3960,3980
3930  PRINT "THERE, THERE."
3940  PRINT "NOTHING IS GOING TO HURT YOU HERE."
3950  GOTO 130
3960  PRINT "WELL, EVERYONE GETS HURT ONCE IN AWHILE"
3970  GOTO 130
```

```
3980  PRINT "AWWW.  WANT ME TO KISS IT AND MAKE IT   ALL BETTER?"
3990  GOTO 130
4000 RN =  INT (1 +  RND (1) * 4)
4010  ON RN GOTO 4020,4040,4070,4080
4020  PRINT "I DON'T SEE THE HUMOR THERE."
4030  GOTO 130
4040  PRINT "HEY!  I DON'T LAUGH AT YOU!"
4050 RN =  INT (1 +  RND (1) * 7): IF RN > 4 THEN  GOTO 130
4060  PRINT "","(MUCH)"
4070  GOTO 130
4080  PRINT "YEAH.  YOU LAUGH NOW."
4090  PRINT "","BUT JUST YOU WAIT..."
4100  GOTO 130
4110 RN =  INT (1 +  RND (1) * 8)
4120  ON RN GOTO 4130,4150,4170,4190,4210,4230,4250,4270
4130  PRINT "THAT'S YOUR PROBLEM."
4140  GOTO 130
4150  PRINT "GO AWAY AND LEAVE ME ALONE."
4160  GOTO 130
4170  PRINT "HOW SHOULD I KNOW?"
4180  GOTO 130
4190  PRINT "WHY SHOULD I CARE?"
4200  GOTO 130
4210  PRINT "GEE.  I DON'T KNOW."
4220  GOTO 130
4230  PRINT "WELL, WHAT DO YOU THINK?"
4240  GOTO 130
4250  PRINT "MY.  YOU DO HAVE YOUR PROBLEMS, DON'T   YOU?"
4260  GOTO 130
4270  PRINT "YOU HAVE MY DEEPEST SYMPATHY."
4280  GOTO 130
4290 RN =  INT (1 +  RND (1) * 3)
4300  ON RN GOTO 4310,4340,4360
4310  PRINT "MY,MY!"
4320  PRINT "YOU ARE THE EMOTIONAL SORT, AREN'T YOU?"
4330  GOTO 130
4340  PRINT "AREN'T YOU OVER-STATING IT A BIT?"
4350  GOTO 130
4360  PRINT "I FEEL EXACTLY THE SAME WAY ABOUT YOU."
4370  GOTO 130
4380  PRINT "THAT'S WHAT I THOUGHT."
4390  GOTO 130
4400  PRINT "OH, I DON'T WANT TO TALK WITH YOU","ANYMORE."
4410  PRINT
4420  PRINT "","GOODBYE."
4430  PRINT : PRINT : PRINT
4440  END
4450 X = 1
4460 YY =  LEN (Q$)
4470 YZ = YY - 1
4480 RP$ =  MID$ (Q$,1,YZ) + "?"
```

```
4490  PRINT RP$
4500  GOTO 270
4510 X = 1
4520 ZZ =  INT (1 +  RND (1) * 5)
4530  ON ZZ GOTO 4540,4560,4580,4600,4620
4540  PRINT "DO YOU HAVE A PROBLEM?"
4550  GOTO 270
4560  PRINT "CAT GOT YOUR TONGUE, PAL?"
4570  GOTO 130
4580  PRINT "WHAT DID YOU SAY?"
4590  GOTO 270
4600  PRINT "I'M SORRY.  I DIDN'T CATCH THAT."
4610  GOTO 270
4620  PRINT "SPEAK UP!"
4630  GOTO 270
4640  PRINT "PLEASE DON'T REPEAT YOURSELF!"
4650 X = 1
4660  GOTO 270
4670 PT = QS - 1
4680 Q$ =  MID$ (Q$,1,PT) + S2$ +  MID$ (Q$,QS + SL)
4690  RETURN
4700 RP$ =  MID$ (RP$,1,QS - 1) +  MID$ (RP$,QS + 1)
4710  RETURN
4720  REM  * REMOVE MARKERS *
4730  IF  MID$ (KE$,4,1) = "@" THEN RP$ =  MID$ (KE$,
      LEN (KE$) - 9)
4740  IF  MID$ (KE$,7,1) = "*" THEN RP$ =  MID$ (KE$,
      LEN (KE$) - 7)
4750  RETURN
4760 RP$ =  MID$ (Q$,QS + KL)
4765  PRINT : PRINT
4770  IF RP$ <  > "" THEN  GOTO 770
4780  GOTO 800
4790  PRINT "SAME TO YOU!"
4800 X = 1
4810  GOTO 800
4820  PRINT "I DON'T BELIEVE THAT FOR A SECOND!"
4830 X = 1
4840  GOTO 800
4850  PRINT "YOU HAVE MY DEEPEST SYMPATHY."
4860 X = 1
4870  GOTO 800
4880  PRINT "OH YEAH?"
4890  PRINT "WELL, YOUR ";YR$;"!"
4900 X = 1
4910  GOTO 800
4920  REM  * IDENITY SWAP DATA *
4930  DATA  " I "," YOU@ "," YOU "," I "," ME "," YOU "
4940  DATA " MY "," YOUR* "," YOUR "," MY "
4950  DATA  " MYSELF "," YOURSELF* "," YOURSELF "," MYSELF "
4960  DATA  " I'M "," YOU'RE* "," YOU'RE "," I'M "
```

```
4970  DATA " AM "," ARE@ "," WERE "," WAS "
4980  DATA  " MOM "," MOTHER "," DAD "," FATHER "
4990  DATA  " DREAMS "," DREAM "
5000  REM  * KEYWORDS *
5010  DATA  "COMPUTER",1,"MACHINE",1,"NAME ",2," ALIKE "
      ,3," LIKE ",3
5020  DATA  " SAME ",3,"YOU@ REMEMBER",4,"DO I REMEMBER"
      ,5,"YOU@ DREAM",6," DREAM ",7
5030  DATA " IF ",8,"EVERYBODY",9,"EVERYONE",9,"NOBODY"
      9," NO ONE",9
5040  DATA  "WAS YOU@",10,"YOU@ WAS",11,"YOUR* SISTER",
      13,"YOUR* MOTHER",13,"WAS I ",12
5050  DATA "YOUR* FATHER",13,"YOUR* BROTHER",13,"YOUR*
      WIFE",13,"YOUR* HUSBAND",13,"YOUR* CHILD".13
5060  DATA  "YOUR* ",14,"ALWAYS",15," ARE I ",16,"ARE@
      YOU@",18," HOW ",25
5070  DATA  " BECAUSE ",19," CAN I ",20,"CAN YOU@",21,
      "CERTAINLY",22,"DEUTSCH",23
5080  DATA  "ESPANOL",23,"FRANCAIS",23,"HELLO",24,"I
      REMIND YOU OF",3," I ARE",26
5090  DATA  "I'M",26,"ITALIANO",23,"MAYBE",28," MY ",29," NO ",30
5100  DATA "PERHAPS",28," SORRY ",31," WHAT",25," WHEN "
      ,25,"WHY DON'T I", 32
5110  DATA  "GOODBYE",42,"GOODBY",42," AIN'T ",47,"
      IRREGARDLESS",47," SCARED ",48
5120  DATA  "WHY CAN'T YOU@",33," YES ",22,"YOU@ WANT",
      34,"YOU@ NEED",34,"YOU@ ARE@ SAD",35
5130  DATA " AFRAID ",48," FRIGHTENED ",48," SCARE ",48,
      " FEAR ",48," TERRIFY ",48
5140  DATA  "YOU'RE* SAD",35,"YOU@ ARE@ UNHAPPY",35,
      "YOU@ ARE@ DEPRESSED", 35,"YOU@ ARE@ SICK",35
5150  DATA  "YOU@ CAN'T",39,"YOU@ CANNOT",39,"YOU@
      DON'T",40,"YOU@ FEEL",41,"ORGASM",43
5160  DATA  "YOU'RE* SICK",35,"YOU@ ARE@ HAPPY",36,
      "YOU'RE* HAPPY",36,"YOU @ ARE@ GLAD",36,"YOU'RE* GLAD",36
5170  DATA  "YOU@ ARE@ ELATED",36,"YOU'RE*ELATED",36,
      "YOU@ ARE@ BETTER",36,"YOU'RE* BETTER",36,"YOU@ FEEL YOU@",37
5180  DATA  "YOU@ THINK YOU@",37,"YOU@ BELIEVE YOU@",37,
      "YOU@ WISH YOU@",3,"YOU@ ARE@",38,"YOU'RE*",38
5190  DATA  "TERRIFIED",48," HURT ",49," LAUGH ",50,"
      HA ",50," HO ",50
5200  DATA  " FUNNY ",50,"WHAT SHOULD YOU@ DO",51,"WHAT
      CAN YOU@ DO",51,"DON'T KNOW WHAT TO DO",51," LOVE ",52
5210  DATA  " HATE ",52," ADORE",52,"DESPISE",52,
      "DETEST",52," LOATHE",52 ,27," WHY ",25
5220  DATA " SEX ",43,"THAT'S RIGHT",53," ARE ",17," I "
```



Computer Creativity

Closely related to intelligence is creativity. I am not even going to try for a definitive explanation of this concept. The experts have been trying for years.

Personally, I doubt that a computer will ever be truly creative. It can "create" pseudo-randomly, as we will see in the programs presented in this chapter. But a computer has no artistic judgement. A human artist creates on an emotional level. A computer can only create mechanically. It has no free will, so it cannot come up with truly new concepts, or combine unrelated concepts in completely unanticipated ways.

Some theories suggest that the human brain also functions in a purely deterministic manner. Even if this is so, the system is far, far more complex than any computer.

No one can say for sure whether or not some future computer will ever be truly creative. One thing is sure, however, no existing computer (and certainly no microcomputer) comes close in a general sense. But in certain limited areas, some amazing results can be achieved.

WHO IS THE ARTIST?

Computer creativity programs usually rely heavily on the random number generator (RND(x)). In a well-thought out program, some very effective pseudo-art works can be created.

Just who is the artist in such a case? Is the computer creating the art work? Or is the programmer, who gives the computer its instructions the real artist? Even though he cannot predict the outcome, he defines its parameters. Certain similarities will appear in each variation "created" by the computer.

The computer functions more as a time-saving device for combining various predetermined possibilities than a true creative force.

Remember, the computer has no artistic judgement. It is up to the human operator to determine which parts of the output to save, and which to reject. The computer does not supplant the human artist. It is just a tool for experimentation. It can come up with combinations that the artist might not come up with on his own.

Computers can be used in the creation of virtually any art form. A lot of work has been done in the area of computer graphics. We will not explore the graphic arts here, because graphic commands vary so widely between different microcomputers, it is hard to generalize the programming techniques.

In this chapter we will look at several programs for computer generated musical composition, and poetry.

POETRY GENERATORS

Computer generated poetry can easily be created by randomly selecting words from a DATA list, and plugging them into a pre-determined format.

The program given in Listing 8-1 writes poems that follow the pattern of "Mary Had A Little Lamb":

(name) HAD A (adjective 1) (noun 1)
(adjective 1) (noun 1)
(adjective 1) (noun 1)
(name) HAD A (adjective 1) (noun 1)
ITS (noun 2) WAS (adjective 2) AS (noun 3)

AND EVERYWHERE THAT (name) (verb 1)
(name) (verb 1)
(name) (verb 1)
AND EVERYWHERE THAT (name) (verb 1)
THE (noun 1) WAS SURE TO (verb 2)

In the model poem, the following words are used:

(name)	MARY
(adjective 1)	LITTLE
(noun 1)	LAMB
(noun 2)	FLEECE
(adjective 2)	SNOW
(verb 1)	WENT
(verb 2)	GO

This program provides ten alternatives for each of these words. Let's walk through a portion of this program.

In line 40, X is set to a value of 0. Next, line 50 calls the subroutine beginning at line 440. A random number from 1 to 10 is added to the value of X. This new value is called Y. Lines 450 through 470 read through the DATA file until the count equals

Y. The DATA file is restored, and the subroutine ends with the RETURN statement. The selected word is passed back to the main body of the program as the string variable A$.

On this pass through the subroutine, X = 0, so Y will have a value from 1 to 10. Let's say the randomly selected value is 7. The subroutine will read through the first seven elements in the DATA file. At the end of the subroutine, A$ will hold the string, "STINKY".

In line 60, N$ is set equal to A$ ("STINKY"), so A$ can be used yet again, without losing the selected word. X is now set to a value of 10, and the subroutine is called again. This time Y can take on a value of 11 (1 + 10) to 20 (10 + 10). Let's say Y = 12. The subroutine returns with A$ = "REAL BIG". This string is assigned to D$.

Now line 100 sets X to 20. This time through subroutine 440, Y's value can range from 21 (1 + 20) to 30 (10 + 20). We'll say it comes up with 24. A$ will equal "COW" when control reverts back to the main program. This string is then stored as T$.

In line 130 D$ is updated to combine the old value of D$ ("REAL BIG"), a space (" "), and T$ ("COW"). This means D$ now equals "REAL BIG COW". The contents of T$ remain unchanged ("COW").

X is then set to 60 in line 140. In the subroutine, Y is set to a value of 61 to 70. Let's say the selected word is "DANCED". Line 160 combines this word with the previously stored value of N$ ("STINKY"). The combination is stored as V$ ("STINKY DANCED").

Now the screen is cleared (line 170), and the first part of the poem is displayed, using the preselected string variables:

"STINKY HAD A REAL BIG COW
REAL BIG COW
REAL BIG COW
STINKY HAD A REAL BIG COW"

Other words in the poem (which are only used once) are selected as they are needed during the creation of the poem. The additional words are selected in the same way as described above. X is set to the appropriate index value and subroutine 440 is called.

When the entire poem has been displayed, the screen freezes until a key is hit (lines 400 through 420) when the program jumps back to line 40 to start all over.

A few sample runs of this simple program are shown in Table 8-1. While the results of this program can be amusing, they can quickly become monotonous, because the poem's framework is very rigid.

A somewhat more versatile and interesting poetry generator program is offered in Listing 8-2. This program can generate several different poem formats, loosely based on the Japanese Haiku. This poetic format was selected because it is very short, and extremely flexible.

A traditional Haiku presents a single image in three lines. The first and third lines are made up of five syllables. The middle line has seven syllables.

In this program, the strict syllable count is ignored. Some of the poems even have a fourth line. Still, the Haiku "feel" is preserved. All of the words in each poem are randomly selected. This helps limit the problem of redundancy that made the first program seem so trite.

Table 8-1. Typical Results from the Poetry Generator Program of Listing 8-1.

```
AGATHA HAD LITTLE RABBIT
LITTLE RABBIT
LITTLE RABBIT
AGATHA HAD A LITTLE RABBIT
IT'S NOSE WAS RED AS SNOW.

AND EVERYWHERE THAT AGATHA MOVED
AGATHA MOVED
AGATHA MOVED
AND EVERYWHERE THAT AGATHA MOVED
THE RABBIT WAS SURE TO WEEP.

RALPH HAD SILLY FROG
SILLY FROG
SILLY FROG
RALPH HAD A SILLY FROG
IT'S EAR WAS WET AS A STONE.

AND EVERYWHERE THAT RALPH COUGHED
RALPH COUGHED
RALPH COUGHED
AND EVERYWHERE THAT RALPH COUGHED
THE FROG WAS SURE TO EAT.

GOOFY HAD CHARMING CAT
CHARMING CAT
CHARMING CAT
GOOFY HAD A CHARMING CAT
IT'S EAR WAS FAT AS A COMPUTER.

AND EVERYWHERE THAT GOOFY COUGHED
GOOFY COUGHED
GOOFY COUGHED
AND EVERYWHERE THAT GOOFY COUGHED
THE CAT WAS SURE TO EAT.
```

Because the words are randomly selected, some of the poems from this program will be meaningless. On the other hand, some striking images can occur more frequently than you might expect.

The words in the DATA file are arranged in five categories:

adjectives
articles
nouns
prepositions
verbs

Words are selected to fit each pattern according to their type.

The program randomly selects from five basic formats. One of them is as follows:

(article) (adjective) (noun) . . .
(article) (noun) (verb) (preposition) (article) (noun)
(adjective) (noun)

Literally thousands of different poems can be created using this skeletal format. Because the format is randomly selected for each new poem, the repeating patterns will not be so obvious.

Array A(15) is used to hold the last few word choice values, to limit word repetition. Some sample poems from this program are shown in Table 8-2.

If you like, you can change any or all of the words in the DATA statements. Some of the words in the program as listed here are intentionally silly, while others lean towards more serious poems. The words in the DATA file will have a significant influence on the tone of your poems. If you change the number of words in the DATA file, be sure to change the appropriate variable in lines 50 through 110, or the program may not function correctly.

Table 8-2. Typical Results from the Program of Listing 8-2.

WOMAN WITH THE CLOUD A CLEAR SQUIRREL INSIDE OF A SKY RISING AIR	PARKING LOT WITH THE DIRT THE ARID HORSE OUTSIDE OF A STREAM SPOTTED SILENCE
A DAPPLED SEA . . . A BUSH YODELS BY THE GRAPEFRUIT WISPY BLIZZARD	A OLD WOMAN . . . THE MEADOW DRIFTS IN THE SEA PLAYFUL GLADE
THE EVENING LIGHT DIRT UPON A MOIST SILENCE THE RIVER STRETCHES	SOUND WITHIN THE EARTH A DUSK FOREST . . . THE CLOUD WEEPS UPON THE SEA YOUTHFUL THUNDER
ROSE WITHOUT A TREE A CARESSING TOAD IN A LAND PLAYFUL SUN	THE FIERCE WIND IS SINKING . . . A SHIMMERING FOOLISH RAIN BENEATH THE SUMMER FLOWER
AIR WITHOUT THE FIELD A WILD BREATH NEAR THE PEACE FIERCE SEA	SOUND GRUMBLES BESIDE CALM SILENT DEW DARK MOONBEAM

If you want to get even more creative and ambitious, you might want to modify the program to create additional (and possibly even longer) poem forms. As a general rule of thumb though, shorter forms have a greater chance of appearing meaningful. The longer a computer generated poem is, the greater the chance that a meaningless combination will show up.

MUSIC COMPOSITION

A computer can also be used to create musical compositions. For simplicity (and greatest possible transportability between various microcomputers), the programs in this section don't actually play the music. They just print out the note names. For convenience, only naturals and sharps are indicated. Flats are displayed as the related sharp. That is, Ab is displayed as G#. Octaves are ignored.

The simplest possible music composition program would be something like the one shown in Listing 8-3. Each note is selected simply by selecting a random value.

Array A$(12) holds the names for the twelve notes in an octave. These names are loaded in from the DATA statement in line 150 in the routine of lines 50 through 80. The screen is then cleared in line 90.

The main body of the program is in lines 100 through 140. A random number from 1 to 12 is selected. The appropriate note name is displayed, lines 120 through 130 give a brief delay, then the program jumps back to line 100 to select the next note.

A typical output from this program is given in Table 8-3. It is illustrated on a musical staff in Fig. 8-1.

The results are not particularly musical. Each note is chosen independently, completely uninfluenced by what has gone on before. This program clearly has no semblance of intelligence. The "music" it produces sounds as random as it is. An occasional brief passage may be musically valid, but usually the results will be unconnected and unattractive. By most definitions, this isn't really music at all, anymore than the sounds produced by a cat walking across a piano keyboard are musical.

Table 8-3. The First Music Composition Program (Listing 8-3) Is Purely Random.

A♯ G C♯ A E B B F♯ D♯ A E G♯ C D A♯
G D D♯ D♯ F♯ F♯ D A♯ F♯ F C♯ D♯ G♯ F
A♯ A♯ F♯ A♯ F A F F♯ C C♯ G E F♯ D♯ D
F♯ B C♯ F♯ G♯ A F C♯ G♯ F♯ A♯ C F G
A E F♯ C F B

A slight improvement can be achieved with the program given in Listing 8-4. In this version, certain notes are given greater weight. That is, some notes are more likely to occur than others.

Notice that the DATA file (and the note name array—A$(x)) is larger to allow some note names to be repeated. The weighting scheme I used is as follows:

NOTE	# of OCCURENCES	WEIGHTING
A	3	10%
A#	1	3%
B	2	7%
C	6	20%
C#	1	3%
D	2	7%
D#	1	3%
E	4	13%
F	3	10%
F#	1	3%
G	5	17%
G#	1	3%

A typical output from this program is illustrated in Table 8-4 and Fig. 8-2. The results are still very random sounding, but at least there is a sense of key (C major), which gives a little more musicality than the previous program. But this program is still entirely unintelligent, and not particularly musical.

The problem is that each note is still being selected independently. The preceding note (or notes) has no influence on the next note. Selection is made on an entirely random basis. That is not the way real music works.

In real music, each new note is selected in relation to the preceding notes. Programming this can be tricky, but it is entirely possible.

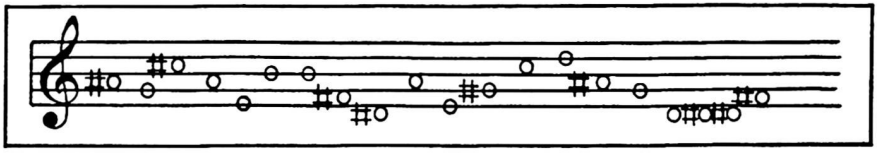


Fig. 8-1. The first music composition program (Listing 8-3) is purely random.

Table 8-4. Weighting the Note Probabilities (Listing 8-4) Improves the Output Somewhat.

C	C	F	E	F	A	F	D	A	A♯	A	G	D	G
G♯	G♯	A	C	G	E	C♯	F	C	C	G	A		
E	C	C	D♯	E	C	A	E	D	E	F	A♯	G♯	
C	G♯	A	C	G	A♯	G	G	G♯	B	G			

A fairly simple program of this type is given in Listing 8-5. Each new note is selected with a different weighting scheme, depending on the previous note. Only the one immediately preceding note is checked before the new note is randomly selected.

The heart of this program is a two dimensional array (A(12,12)). A second two dimensional array (A$(12,12)) holds the appropriate note names.

The X coordinate for this array indicates the preceding note:

X=	NOTE	X=	NOTE
1	A	7	D#
2	A#	8	E
3	B	9	F
4	C	10	F#
5	C#	11	G
6	D	12	G#

The Y coordinate is selected randomly. The stored value at A(x,y) determines the next note.

Certain notes are more likely to occur after other specific notes. Some note pairs can never occur.

For example, if the preceding note is A, the following are the 12 possibilities for the next note:

Y=	NOTE	Y=	NOTE
1	A	7	A
2	C	8	C
3	E	9	E
4	F	10	F
5	A#	11	D
6	G#	12	B

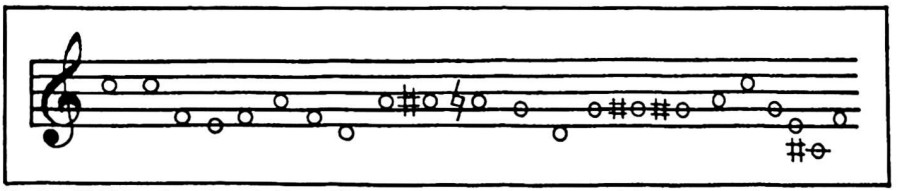


Fig. 8-2. Weighting the note probabilities (Listing 8-4) improves the musicality of the output somewhat.

NOTE	WEIGHTING PROBABILITY	NOTE	WEIGHTING PROBABILITY
A	17%	D#	0%
A#	8%	E	17%
B	8%	F	17%
C	17%	F#	0%
C#	0%	G	0%
D	8%	G#	8%

A to D# will never occur in this weighting scheme. A to C is twice as likely to occur as A to D. And so forth for each note.

The arrays are loaded from the DATA statements in lines 40 through 120. The screen is then cleared, and the first note is set to C.

The current note name is displayed in line 160. The new note value is randomly selected from the appropriate array in lines 170 through 190. After a brief delay (lines 200 through 210), the program jumps back to line 160, and this process repeats indefinitely. A typical output sequence from this program is illustrated in Table 8-5 and Fig. 8-3.

The results here are somewhat better than in earlier programs, but they are still pretty random sounding. Musical sounding phrases will occur a little more frequently with this program, but they will be separated by a lot of garbage.

The technique used here could be expanded to check back more than a single note. The more preceding notes that are used to determine the new note, the better the results will be. But the memory requirements increase exponentially with added notes. The one note back check used here requires a two dimensional array with 144 locations. A two note back check would require a three dimensional array with 1728 locations. A four dimensional array with 20736 locations would be needed to check back three notes. As you can see, this technique can very quickly become extremely unwieldy.

So far all of our composition programs have been rather aimless, resulting in very formless "music." In our poetry generator programs, we used existing models to define the format for the computer generated pieces. The same concept can be applied to musical composition. The computer can create new music based on pre-existing music. The results are likely to render the source unrecognizable, but the musical patterns will be more or less preserved.

A Tune Imitator program designed around this idea is given in Listing 8-6. The user is prompted to enter the names of each note of the model tune in sequence. The computer keeps track of all the note pairs and creates a weighting scheme based on

```
C C A A G# A D C G B C B A F
F# C A G# D D C A F E G F F
F# G E E F F# G B C E F E E
G C D B G D B G G E C A F
C E C F D D# C D B C B C B
C E C C F F
```

Table 8-5. Better Music can be Obtained by Letting the Previous Note Affect the Choice of the Next Note, as in the Program of Listing 8-5.

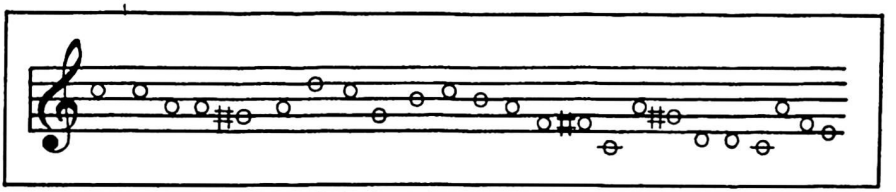


Fig. 8-3. Better music can be obtained by letting the previous note affect the choice of the next note, as in the program of Listing 8-5.

the pair frequencies in the model tune. The notes of the new piece are randomly selected according to this weighting scheme. In other words, once the model tune has been entered, this program functions pretty much like the program of Listing 8-5.

An even more effective variation is given in Listing 8-7. In this program the shape of the model tune is analyzed and duplicated. Instead of entering specific note names, the computer only considers whether each successive note goes up, goes down, or stays the same. Of all the programs presented in this chapter, I feel this one produces musical sounding results most frequently.

A more sophisticated program of this type would also analyze and imitate the rhythm patterns of a model tune. Rhythm has been ignored in these programs for simplicity, but this is the logical next step for experimentation.

```
10  REM   8-1     SIMPLE POETRY
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40 X = 0
50  GOSUB 440
60 N$ = A$
70 X = 10
80  GOSUB 440
90 D$ = A$
100 X = 20
110  GOSUB 440
120 T$ = A$
130 D$ = D$ + " " + T$
140 X = 60
150  GOSUB 440
160 V$ = N$ + " " + A$
170  HOME
180  PRINT
190  PRINT N$;" HAD A ";D$
200  PRINT D$
210  PRINT D$
220  PRINT N$;" HAD A ";D$
230 X = 30
240  GOSUB 440
250  PRINT "IT'S ";A$;" WAS ";
260 X = 40
270  GOSUB 440
280  PRINT A$;" AS ";
290 X = 50
300  GOSUB 440
310  PRINT A$;"."
320  PRINT
330  PRINT "AND EVERYWHERE THAT ";V$
340  PRINT V$
350  PRINT V$
360  PRINT "AND EVERYWHERE THAT ";V$
370 X = 70
380  GOSUB 440
390  PRINT "THE ";T$;" WAS SURE TO ";A$;"."
400  PRINT : PRINT : PRINT
410  PRINT "PLEASE PRESS <RETURN>"
420  INPUT Q$
430  GOTO 40
440 Y =  INT (1 +  RND (1) * 10) + X
450  FOR T = 1 TO Y
460  READ A$
470  NEXT T
480  RESTORE
490  RETURN
500  DATA  MARY,RALPH,AGATHA,IRVING,WALDO,GLADYS,STINKY,
     AMY,GOOFY,ISAAC
```

```
510  DATA  LITTLE,REAL BIG,SILLY,CUDDLY,STUPID,CLEVER,
     CLUMSY,SCREWY,PURPLE,CHARMING
520  DATA  LAMB,FROG,ROCK,COW,DOG,CAT,MOUSE,RABBIT,FERRET,TROLL
530  DATA  FLEECE,SKIN,NOSE,FOOT,EAR,CAR,BODY,HAIR,FACE,TONGUE
540  DATA  WHITE,RED,BLUE,GREEN,WEIRD,WET,DRY,FAT,CUTE,ODD
550  DATA  SNOW,GRASS,A BUTTON,A STONE,THE SEA,DIRT,
     SOMETHING ELSE,ME,CHEESE,A COMPUTER
560  DATA  WENT,STAYED,LEFT,COUGHED,LAUGHED,CRIED,DANCED,
     LOOKED,MOVED,SLEPT
570  DATA  GO,BELCH,SLEEP,DIE,WEEP,EAT,WALTZ,YODEL,LEAVE,BE
```

```
10  REM   8-2      ADVANCED POETRY
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM A(15)
50  REM  * AJ = # OF ADJECTIVES / AR= # OF ARTICLES /
    NN = # OF NOUNS
60  REM  * PR = # OF PREPOSITIONS / VB = # OF VERBS *
70 AJ = 75
80 AR = 5
90 NN = 65
100 PR = 15
110 VB = 65
120  REM  *  SET UP DISPLACEMENT VARIABLES *
130 AX = AJ
140 NX = AX + AR
150 PX = NX + NN
160 VX = PX + PR
170  REM  * CHOOSE PATTERN *
180  HOME
190  PRINT : PRINT
200 C =  INT (1 +  RND (1) * 5)
210  FOR T = 1 TO 15
220 A(T) = 0
230  NEXT T
240 S = 1
250  ON C GOTO 260,770,1070,1490,1940
260  PRINT "      ";
270 X = AR
280 Y = AX
290  GOSUB 2340
300 X = AJ
310 Y = 0
320  GOSUB 2340
330 X = NN
340 Y = NX
350  GOSUB 2340
360  PRINT "..."
370 X = AR
380 Y = AX
390  GOSUB 2340
400 X = NN
410 Y = NX
420  GOSUB 2340
430 X = VB
440 Y = VX
450  GOSUB 2340
460 X = PR
470 Y = PX
480  GOSUB 2340
490 X = AR
500 Y = AX
510  GOSUB 2340
```

```
520 X = NN
530 Y = NX
540  GOSUB 2340
550  PRINT
560  PRINT "    ";
570 X = AJ
580 Y = 0
590  GOSUB 2340
600 X = NN
610 Y = NX
620  GOSUB 2340
630 C =  INT (1 +  RND (1) *.3)
640  IF C = 2 THEN  GOTO 730
650  PRINT
660  PRINT
670  PRINT
680  PRINT
690  PRINT "","###"
700  PRINT : PRINT "PLEASE PRESS <RETURN>"
710  INPUT Q$
720  GOTO 180
730 X = VB
740 Y = VX
750  GOSUB 2340
760  GOTO 650
770  PRINT "      ";
780 X = NN
790 Y = NX
800  GOSUB 2340
810 X = VB
820 Y = VX
830  GOSUB 2340
840  PRINT
850 X = PR
860 Y = PX
870  GOSUB 2340
880 X = AJ
890 Y = 0
900  GOSUB 2340
910 X = AJ
920 Y = 0
930  GOSUB 2340
940 X = NN
950 Y = NX
960  GOSUB 2340
970  PRINT
980  PRINT "    ";
990 X = AJ
1000 Y = 0
1010  GOSUB 2340
1020 X = NN
```

```
1030 Y = NX
1040  GOSUB 2340
1050  PRINT "..."
1060  GOTO 650
1070  PRINT "       ";
1080 X = AR
1090 Y = AX
1100  GOSUB 2340
1110 X = AJ
1120 Y = O
1130  GOSUB 2340
1140 X = NN
1150 Y = NX
1160  GOSUB 2340
1170 X = VB
1180 Y = VX
1190  GOSUB 2340
1200  PRINT "..."
1210  PRINT
1220 X = AR
1230 Y = AX
1240  GOSUB 2340
1250 X = AJ
1260 Y = O
1270  GOSUB 2340
1280 C =  INT (1 +  RND (1) * 2)
1290  IF C = 1 THEN  GOSUB 2340
1300 X = NN
1310 Y = NX
1320  GOSUB 2340
1330  PRINT
1340  PRINT "    ";
1350 X = PR
1360 Y = PX
1370  GOSUB 2340
1380 X = AR
1390 Y = AX
1400  GOSUB 2340
1410 X = AJ
1420 Y = O
1430  GOSUB 2340
1440 X = NN
1450 Y = NX
1460  GOSUB 2340
1470  PRINT
1480  GOTO 650
1490  PRINT "       ";
1500 X = NN
1510 Y = NX
1520  GOSUB 2340
1530 X = PR
```

```
1540 Y = PX
1550  GOSUB 2340
1560 X = AR
1570 Y = AX
1580  GOSUB 2340
1590 X = NN
1600 Y = NX
1610  GOSUB 2340
1620  PRINT
1630 C =  INT (1 +  RND (1) * 7)
1640  IF C < 3 THEN  GOTO 260
1650 X = AR
1660 Y = AX
1670  GOSUB 2340
1680 X = AJ
1690 Y = O
1700  GOSUB 2340
1710 X = NN
1720 Y = NX
1730  GOSUB 2340
1740 X = PR
1750 Y = PX
1760  GOSUB 2340
1770 X = AR
1780 Y = AX
1790  GOSUB 2340
1800 X = NN
1810 Y = NX
1820  GOSUB 2340
1830  PRINT
1840  PRINT "    ";
1850 X = AJ
1860 Y = O
1870  GOSUB 2340
1880 X = NN
1890 Y = NX
1900  GOSUB 2340
1910 C =  INT (1 +  RND (1) * 3)
1920  IF C = 3 THEN  GOTO 730
1930  GOTO 650
1940  PRINT "      ";
1950 X = AR
1960 Y = AX
1970  GOSUB 2340
1980 X = AJ
1990 Y = O
2000  GOSUB 2340
2010 X = AJ
2020 Y = O
2030  GOSUB 2340
2040 X = NN
```

```
2050 Y = NX
2060  GOSUB 2340
2070  PRINT
2080 X = PR
2090 Y = PX
2100  GOSUB 2340
2110 C =  INT (1 +  RND (1) * 5)
2120  IF C = 4 THEN  GOTO 2160
2130 X = AR
2140 Y = AX
2150  GOSUB 2340
2160 X = AJ
2170 Y = O
2180  GOSUB 2340
2190 X = NN
2200 Y = NX
2210  GOSUB 2340
2220  PRINT
2230 X = AR
2240 Y = AX
2250  GOSUB 2340
2260 X = NN
2270 Y = NX
2280  GOSUB 2340
2290 X = VB
2300 Y = VX
2310  GOSUB 2340
2320  PRINT
2330  GOTO 650
2340 Z =  INT (1 +  RND (1) * X) + Y
2350  IF X = AR THEN  GOTO 2440
2360 Q = O
2370  FOR T = 1 TO 15
2380  IF A(T) = Z THEN Q = 1
2390  NEXT T
2400  IF Q = 1 THEN  GOTO 2340
2410 A(S) = Z
2420 S = S + 1
2430  IF S > 15 THEN S = 1
2440  FOR T = 1 TO Z
2450  READ A$
2460  NEXT T
2470  RESTORE
2480  PRINT A$;" ";
2490  RETURN
2500  REM  * ADJECTIVES *
2510  DATA  AUTUMN,SUMMER,SPRING,WINTER,BITTER
2520  DATA  LOVELY,WEIRD,FOUL,HIDDEN,HUGE
2530  DATA  SOFT,CUTE,HARD,HARSH,UGLY
2540  DATA  MISTY,CLEAR,SILENT,NOISY,EMPTY
2550  DATA  FULL,FOOLISH,DRY,WET,ARID
```

```
2560  DATA  STRANGE,CALM,SMOOTH,ROUGH,ICY
2570  DATA  LIGHT,DARK,HEAVY,FEATHERY,HOT
2580  DATA  HEATED,DAPPLED,SPOTTED,MYSTIC,DAWN
2590  DATA  EVENING,DUSK,NIGHT,SHIMMERING,WISPY
2600  DATA  SOLID,FIERCE,BROKEN,MOIST,DAMP
2610  DATA  WILD,GENTLE,TENDER,CARESSING,FALLING
2620  DATA  PLAYFUL,FALLEN,RISING,THROBBING,YOUNG
2630  DATA  AGED,LUSTFUL,OLD,YOUTHFUL,HORNY
2640  DATA  SOLEMN,WHITE,BLACK,RED,BLUE
2650  DATA  GREEN,YELLOW,GREY,PURPLE,ORANGE
2660  REM  * ARTICLES *
2670  DATA  THE,A,THE,A,THE
2680  REM  * NOUNS *
2690  DATA  SUN,MOON,SKY,AIR,EARTH
2700  DATA  RIVER,STREAM,WATERFALL,OCEAN,SEA
2710  DATA  RAIN,WIND,LAND,BREEZE,BREATH
2720  DATA  DREAM,SHADOW,TREE,LEAF,FLOWER
2730  DATA  SNOW,DEW,HAIL,FOREST,WOOD
2740  DATA  MEADOW,FIELD,PARK,SHOPPING CENTER,PARKING LOT
2750  DATA  DUST,DIRT,GRASS,GLADE,BUSH
2760  DATA  BIRD,DEER,BUTTERFLY,GRAPEFRUIT,CAT
2770  DATA  HORSE,MAN,WOMAN,SQUIRREL,CATERPILLAR
2780  DATA  FIREFLY,ROSE,DAFFODIL,TOAD,FROG
2790  DATA  HAZE,CLOUD,SOUND,SONG,THUNDER
2800  DATA  CRY,VOICE,SILENCE,SURF,BIRDCALL
2810  DATA  STORM,PEACE,WAR,BLIZZARD,MOONBEAM
2820  REM  * PREPOSITIONS *
2830  DATA  UNDER,OVER,BESIDE,ON,IN
2840  DATA  WITH,WITHOUT,NEAR,FAR FROM,OUTSIDE OF
2850  DATA  UPON,BENEATH,WITHIN,BY,INSIDE OF
2860  REM  * VERBS *
2870  DATA  QUIVERS,FLOATS,CRASHES,DRIFTS,FALLS
2880  DATA  IS FALLING,HAS FALLEN,HAS STOPPED,PASSES,IS PASSING
2890  DATA  HAS PASSED,IS STOPPING,SINGS,STRUGGLES,RESTS
2900  DATA  WORSHIPS,WEEPS,MOURNS,BREAKS,IS BREAKING
2910  DATA  HAS BROKEN,DIES,IS BORN,LIVES,SLEEPS
2920  DATA  WAKES,YAWNS,STRETCHES,LAUGHS,SMILES
2930  DATA  GROWS,WAITS,LINGERS,CREEPS,CRAWLS
2940  DATA  DREAMS,FLUTTERS,TRICKLES,SEEPS,SITS
2950  DATA  RISES,IS RISING,HAS RISEN,DANCES,TRIPS
2960  DATA  MURMURS,SINGS,HUMS,GRUMBLES,COMPLAINS
2970  DATA  YODELS,SINKS,IS SINKING,HAS SUNK,SWIMS
2980  DATA  IMAGINES,STANDS,HOOVERS,SOARS,SAGS
2990  DATA  PAUSES,RESUMES,QUITS,IS BORING,HOPS
```

```
10  REM   8-3      RANDOM COMPOSER
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM A$(12)
50  FOR X = 1 TO 12
60  READ N$
70 A$(X) = N$
80  NEXT X
90  HOME
100 Y =  INT (1 +  RND (1) * 12)
110  PRINT A$(Y),
120  FOR T = 1 TO 555
130  NEXT T
140  GOTO 100
150  DATA  A,A#,B,C,C#,D,D#,E,F,F#,G,G#
```

```
10  REM   8-4     WEIGHTED COMPOSER
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM A$(30)
50  FOR X = 1 TO 30
60  READ N$
70 A$(X) = N$
80  NEXT X
90  HOME
100 Y =  INT (1 +  RND (1) * 30)
110  PRINT A$(Y),
120  FOR T = 1 TO 555
130  NEXT T
140  GOTO 100
150  DATA  A,A#,B,C,C#,D,D#,E,F,F#,G,G#
160  DATA  A,B,C,D,E,F,G,A,C,E,F,G,C,E,G,C,G,C
```

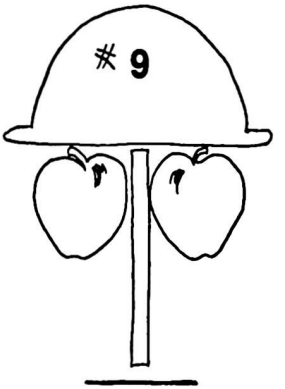
```
10  REM   8-5     WEIGHTED W/BACK CHECK
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM A$(12,12): DIM A(12,12)
50  FOR X = 1 TO 12
60  FOR Y = 1 TO 12
70  READ N$
80 A$(X,Y) = N$
90  READ N
100 A(X,Y) = N
110  NEXT Y
120  NEXT X
130  HOME
140 N$ = "C"
150 X = 4
160  PRINT N$,
170 L = X
180 H =  INT (1 +  RND (1) * 12)
190 X = A(L,H):N$ = A$(L,H)
200  FOR T = 1 TO 555
210  NEXT T
220  GOTO 160
230  DATA  A,1,C,4,E,8,F,9,A#,2,G#,12
240  DATA  A,1,C,4,E,8,F,9,D,6,B,3
250  DATA  A#,2,A,1,G#,12,C#,5,F#,10,G,11
260  DATA  B,3,C,4,F,9,B,3,C,4,F,9
270  DATA  C,4,C,4,B,3,G,11,A,1,F,9
280  DATA  C,4,C,4,B,3,G,11,A,1,E,8
290  DATA  E,8,G,11,F,9,A,1,D,6,B,3
300  DATA  E,8,G,11,F,9,A,1,D,6,B,3
310  DATA  C#,5,D,6,C,4,B,3,A#,2,A,1
320  DATA  C#,5,D,6,C,4,B,3,C,4,D#,7
330  DATA  C,4,B,3,F,9,E,8,C,4,A,1
340  DATA  C,4,B,3,D,6,A,1,C#,5,D#,7
350  DATA  D#,7,C#,5,F#,10,A#,2,D,6,C,5
360  DATA  F,9,A,1,E,8,C,4,D,6,D,6
370  DATA  E,8,C,4,G,11,F,9,D#,7,B,3
380  DATA  E,8,C,4,G,11,E,8,C,4,G,11
390  DATA  F,9,A,1,C,4,F#,10,D,6,G,11
400  DATA  E,8,F,9,A,1,C,4,E,8,F#,10
410  DATA  F,9,E,8,F#,10,F,9,E,8,C,4
420  DATA  G,11,D,6,F,9,F#,10,G,11,D,6
430  DATA  G,11,C,4,E,8,B,3,D,6,F,8
440  DATA  A,1,G#,12,G,11,C,4,E,8,B,3
450  DATA  G#,12,G,11,G,11,C,4,E,8,B,3
460  DATA  D,6,F,8,A,1,G,11,G#,12,F#,10
```

```
10  REM   8-6      TUNE IMITATOR
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM N$(12)
50  DIM C(12,12)
60  HOME
70  FOR X = 1 TO 12
80  FOR Y = 1 TO 12
90 C(X,Y) = 1
100  NEXT Y: NEXT X
110  FOR X = 1 TO 12
120  READ A$
130 N$(X) = A$
140  NEXT X
150  PRINT
160  PRINT "ENTER NOTE NAMES IN SEQUENCE"
170  PRINT " <A - A# - B - C - C# - D - D# - E -","F - F# - G - G#>"
175  PRINT
180  PRINT "ENTER / TO END"
190  PRINT "NOTE #1";
195  INPUT Q$
200  GOSUB 560
210  IF X = 0 THEN  GOTO 190
220 Y = X
230 C = 2
240  PRINT "NOTE #";C;
245  INPUT Q$
250  GOSUB 560
260  IF Q$ = "/" THEN  GOTO 320
270  IF X = 0 THEN  GOTO 240
280 C(X,Y) = C(X,Y) + 2
290 C = C + 1
300 Y = X
310  GOTO 240
320  HOME
330  FOR X = 1 TO 12
340 Z = 0
350  FOR Y = 1 TO 12
360 Z = Z + C(X,Y)
370  NEXT Y
390 Q = Z / 12
400  FOR Y = 1 TO 12
410 C(X,Y) = C(X,Y) / Q
420  NEXT Y
425  NEXT X
430  HOME
440  PRINT
450 X =  INT (1 +  RND (1) * 12)
460  PRINT N$(X),
470 W =  INT (1 +  RND (1) * 12)
480 E = 1
```

```
490 W = W - C(X,E)
500  IF W < 0 THEN  GOTO 540
510 E = E + 1
520  IF E > 12 THEN  GOTO 470
530  GOTO 490
540 X = E
545  FOR T = 1 TO 555
547  NEXT T
550  GOTO 460
560 X = 0
570  IF Q$ = "A" THEN X = 1
580  IF Q$ = "A#" THEN X = 2
590  IF Q$ = "B" THEN X = 3
600  IF Q$ = "C" THEN X = 4
610  IF Q$ = "C#" THEN X = 5
620  IF Q$ = "D" THEN X = 6
630  IF Q$ = "D#" THEN X = 7
640  IF Q$ = "E" THEN X = 8
650  IF Q$ = "F" THEN X = 9
660  IF Q$ = "F#" THEN X = 10
670  IF Q$ = "G" THEN X = 11
680  IF Q$ = "G#" THEN X = 12
690  RETURN
700  DATA  A,A#,B,C,C#,D,D#,E,F,F#,G,G#
3807 "*";
```

```
10  REM   8-7     TUNE SHAPE IMITATOR
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  DIM P(101)
50  DIM A$(15)
70  FOR X = 1 TO 101
80 P(X) = 5
90  NEXT X
100  HOME
110  PRINT
120  PRINT "FOR EACH NOTE INDICATE ITS"
130  PRINT "RELATIONSHIP TO LAST NOTE."
140  PRINT "TO END ENTER 0."
150  PRINT "MAXIMUM TUNE LENGTH IS 100."
160  PRINT
170 C = 2
180 P(1) = 0
190  PRINT : PRINT "","NOTE #";C
200  PRINT : PRINT "1 -- LOWER"
210  PRINT "2 -- SAME"
220  PRINT "3 -- HIGHER"
230  INPUT X
240  IF X = 0 THEN  GOTO 320
250 X =  INT (X)
260  IF X < 1 OR X > 3 THEN  GOTO 200
270 X = X - 2
280 P(C) = X
290 C = C + 1
300  IF C > 100 THEN  GOTO 320
310  GOTO 190
320  HOME
330  PRINT
340  PRINT " TUNE SHAPE SUMMARY": PRINT
350  PRINT "* ";
360 C = 2
370 X = P(C)
380  IF X = 5 THEN  GOTO 450
390  IF X =  - 1 THEN  PRINT "D ";
400  IF X = 0 THEN  PRINT "S ";
410  IF X = 1 THEN  PRINT "H ";
420 C = C + 1
430  IF C > 100 THEN  GOTO 450
440  GOTO 370
450  PRINT
460  PRINT "","DONE"
470  FOR X = 1 TO 15
480  READ Q$
500 A$(X) = Q$
520  NEXT X
525  RESTORE
530  PRINT : PRINT : PRINT
```

```
540  PRINT "PRESS <RETURN> FOR COMPOSITION"
545  INPUT Q$
550 X =  INT (1 +  RND (1) * 7)
560  FOR Q = 2 TO C
570  PRINT A$(X),
580  FOR T = 1 TO 555
590  NEXT T
600 Z = X
610 Y = P(Q)
620  IF Y = 5 THEN Q = C: GOTO 660
630 X = X +  INT (1 +  RND (1) * 5) * Y
640  IF X < 1 THEN X = X + 7
650  IF X > 15 THEN X = X - 7
660  NEXT Q
670  PRINT : PRINT
680  PRINT "PRESS <RETURN>"
690  INPUT Q$
700  HOME
710  PRINT
720  PRINT "1 -- REPEAT THIS PATTERN"
730  PRINT "2 -- NEW PATTERN"
740  PRINT "0 -- END"
750  INPUT X
760  IF X = 1 THEN  GOTO 530
770  IF X = 2 THEN  GOTO 70
780  END
790  DATA  A,B,C,D,E,F,G,A',B',C',D',E',F',G',A''
```



Story-Writing Programs

The same basic techniques described in the last chapter can also be used to create computer generated fiction. However, to obtain good results, a number of additional factors must be considered.

This chapter will feature two very simple story writing programs. Neither produces true stories, but they are a good place to begin your own experimentation. This chapter will conclude with some tips on further improvements and experimentation.

STORY-WRITER 1

The first story writing program is given in Listing 9-1. As you glance over this program, you should see that it is fundamentally not too different from the poetry writing programs of Chapter 8. Appropriate words are randomly selected from a DATA file, and plugged into pre-determined sentence formats.

The main subroutine of this program begins at line 1470. This routine selects and displays the next word. It functions in the same way as the similar subroutines in the poetry programs of Chapter 8. Whenever this subroutine is called, two variables have to be defined:

X Number of words in the appropriate group
Y Displacement value

A random value from 1 to X is selected, and the displacement value (Y) is added to it. This is the count variable (Z). The computer READs through the DATA file until

a count of Z is reached. The last item READ at the conclusion of the count is the selected word. The word is displayed in line 1510. Now the DATA file is RESTOREd, and control is RETURNed to the main body of the program.

The word count variables are preset in lines 60 through 210. The meaning of these, and all the other variables used in this program are summarized in Table 9-1. If you add (or delete) any words in the DATA file, be sure to update the appropriate count variable, or the program will not function properly. As an example, let's say you add five more adjectives. Line 120 should then be changed to read:

```
120 AJ=30
```

Any new DATA lines should be placed in the appropriate word group, as defined by the REMarks. For example, a new DATA line of additional adjectives may be placed at line 1755, or 1793, but not at 1855, or 2010.

To give some consistency to the story, two main characters are selected from the Proper Noun group. These names are more likely to occur in any given sentence than any of the other names.

X is set to PN (line 230), and Y is set to PX (line 240). The word select subroutine is called. When control RETURNs to line 260, A$ holds the selected name. A permanent string variable (M1$) is assigned the same value. The word select subroutine is called again from line 270. This time when it RETURNs to line 280, a second permanent variable (M2$) takes the value of A$.

The two permanent string variables (M1$ and M2$) are compared in line 290. If

Table 9-1. Variables Used in the First Story-Writer Program of Listing 9-1.

AJ	Adjective count
AR	Article count
AV	Adverb count
AX	Article offset
AY	Adjective offset
AZ	Adverb offset
NN	Noun count
NX	Noun offset
PN	Proper noun count
PR	Preposition count
PX	Proper noun offset
PY	Preposition offset
Q	Double subject random select
S	Phrase random select
T	Phrase random select/Word select counter
V1	Verb 1 count
V2	Verb 2 count
VX	Verb 1 offset
VY	Verb 2 offset
X	Current count value
Y	Current offset value
Z	Word select value/Main character random select
A$	Current word
M1$	Main character #1
M2$	Main character #2

they happen to be identical, the program jumps back to line 270 to select a new string for M2$.

Once two different names have been selected, the screen is cleared (line 310). When running this program, you will probably notice that the selected names are briefly displayed before the screen is cleared. This is allowed to avoid including a second word select routine that does not display the selected word. The names will be displayed so briefly that if you blink, you'll miss them. The screen is cleared almost as soon as the words are displayed. This isn't very elegant, but it doesn't do any harm.

All of the sentences generated by the program follow this simple form:

NOUN PHRASE VERB PHRASE

Inverted and compound sentences are ignored for simplicity. This makes the output a little monotonous, but there is some variety in the types of noun phrases and verb phrases generated.

The noun phrase is selected in a subroutine beginning at line 440. There are three basic noun phrase forms:

ARTICLE NOUN
PROPER NOUN
ARTICLE ADJECTIVE NOUN

Once the appropriate noun phrase has been selected and displayed (the main word select subroutine is called from the noun phrase subroutine), a RETURN statement concludes this subroutine.

In line 340 a random value from 1 to 5 is selected. If this random value happens to be 3 (20 percent of the time), the current sentence will be given a compound subject. The word "AND" will be displayed, and the noun phrase subroutine will be called a second time.

Next line 360 calls a subroutine to create the verb phrase and complete the main body of the sentence. This subroutine begins in line 680. It works in pretty much the same way as the noun phrase subroutine. One of six verb phrase formats is randomly selected:

VERB			
VERB	PREPOSITION	ARTICLE	NOUN
VERB	PREPOSITION	ARTICLE	ADJECTIVE NOUN
VERB*	ARTICLE	NOUN	
VERB*	ARTICLE	ADJECTIVE	NOUN
VERB*	PROPER	NOUN	

All of the verbs are in the past tense. Notice that there are two verb groups. One set of verbs require no object, such as COUGHED. The other set of verbs (VERB* in the above list) do require an object, such as CARRIED. For the sentence to make sense, the subject must CARRY something. "RALPH CARRIED." is not a meaningful sentence.

When the verb phrase subroutine RETURNs to line 370, a new random value from 1 to 5 is selected. If this value is 3 (20 percent chance), an additional phrase is added to the sentence. This extra phrase is generated in a subroutine starting with line 1380. The extra phrase always takes the same form:

"WITH" ARTICLE NOUN

Line 390 prints a period to conclude the sentence.

A dummy input is set up in line 400. The program waits until the user presses the <RETURN> key before continuing. This gives the user as much time as he likes to read the sentence. Without some kind of delay here, the display screen will be very quickly filled. Sentences can scroll off the top of the screen before the user gets a chance to read them.

If you modify the program to send the sentences out to a printer, rather than to a monitor, line 400 may be eliminated.

Once the <RETURN> key has been depressed, the program jumps back to line 330 to start a new sentence.

Some typical output sentences from this program are shown in Table 9-2. There are two primary problems here. One is the monotonous quality of the sentence forms mentioned before. The other significant limitation is that the sentences are completely unconnected. What happened before has absolutely no influence on what happens next.

Table 9-2. Some Typical Sentences Produced by the First Story-Writer Program.

HER COP SUDDENLY FOUGHT HIS ROBOT.
HIS BIG DOOR ERRATICALLY FOUGHT A SCREWDRIVER.
HIS PENNY SUDDENLY QUIVERED.
THE NEW FLOWER CARELESSLY CRITICIZED THE SCREWDRIVER.
HER CHAIR WEIRDLY STROKED A HAMMER.
THE FLOWER IRRITATED THE HUGE CROOK.
THE BEER BOTTLE WEIRDLY KISSED SUZIE.
HER HUGE DIRTY SOCK HAPPILY KICKED A MAN.
A DRUNK AND A GIRL WEIRDLY HUMMED CLOSE TO THE SOFT HORSE.
SUZIE GREW WITH HIS RULER.
THE BEER BOTTLE AND HUBERT LANGUIDLY SMELLED A FROG.
A HAMMER FOLLOWED QUAG.
CINDY CARELESSLY MET SIDNEY.
THE CLUMSY HORSE CONFUSED BONZO.
THE BEER BOTTLE AND HIS DAINTY RULER LANGUIDLY STROKED A SLIMY RODENT.
GLADYS AND HIS DREADFUL BEER BOTTLE CRUDELY FONDLED QUAG.
GLADYS SUDDENLY KICKED THE STRANGE HORSE.
HERMAN CAREFULLY FELT THE HORSE.
CINDY WEIRDLY KICKED HIS DISGUSTING PENCIL SHARPENER.
A CAT CRAWLED WITH A ORANGE BOOK.
SUZIE CRUDELY CRITICIZED HER SANDWICH.
HIS CAT LEFT OVER THE DOOR.
THE CROOK AND THE GREEN BOY TENDERLY YODELED.
LADY BUCKLEY AND HERMAN COMPLAINED UNDER THE ROBOT.
HIS CROOK HAPPILY STOPPED.
A PURPLE SHOE WENT NEAR THE KUMQUAT.
HER MAN FOUGHT THE FEATHER.
HER SEXY LAMPSHADE TICKLED HER SOFT COP.

STORY-WRITER 2

A second story-writing program is given in Listing 9-2. In many ways, this program is much more limited than the one presented as Listing 9-1. No attempt is made to vary the sentence structure. Some verbs require an object, while others don't. That is the only variation. Only proper nouns are employed.

There is one significant improvement in this program, however. It keeps track of each character's current condition. All other factors are kept as simple as possible for purposes of illustration.

The "story" is just a list of events in a hypothetical room. There are ten characters. Their names are stored in array C$(10), while array C(10) holds their condition codes. These arrays are loaded in lines 5 through 130.

There are three possible conditions for each character:

1 character is in the room
0 character is not in the room
−1 character is dead

The condition codes limit the possible actions that can be performed by the character. A character with a code of 0 (outside) can either "COME IN," or "ARRIVE." Of course, these two verbs mean the same thing. They are both included to add a little variety.

A character who is in the room (code = 1) can not "COME IN," or "ARRIVE," but any of the other verbs in the DATA file would be acceptable. If the selected VERB requires an object, the object character must also be in the room (code = 1).

Finally, if a character is dead (code = −1), he can't do anything at all. Once a character has "DIED," that name cannot appear in any later sentences.

Variable K is used for counting the number of sentences currently displayed on the screen. After 12 sentences have been displayed, the user is prompted to hit the <RETURN> key. The screen is then cleared, K is set to zero, and a new series of sentences will be displayed.

Before each sentence, the condition codes of all the characters are examined in lines 210 through 250. If the combined condition code value for all of the characters added together is less than 3, the story ends (lines 700 through 710). This will happen after three or more characters have died. It will also be dependent on how many surviving characters are still in the room.

The sentence construction process actually begins in line 260. One of the ten characters is randomly selected. If the selected character is dead (code = −1), the program jumps back to line 260 to try again. The selected character name is displayed in line 280.

A random value from 1 to 3 is selected. If this value happens to be a 2 and the selected character is currently in the room, the program jumps to line 470 for an object verb. This section of the program will be discussed shortly.

If an object verb is not selected, then one of the fifteen solo verbs must be used. First, the computer steps through the first ten items in the DATA file. These are the character names, which have already been loaded into the string array (C$(x)).

A random number from 1 to 15 is selected in line 340. The computer counts through

the DATA file, reading a string (A$) and a numerical value (A) for each step. Once the selected count has been achieved, the DATA file is RESTOREd (line 380) for later reuse.

The selected verb must now be examined to determine if it indicates a possible action for the selected character. Variable A may have one of four possible values, from 0 to 3. These values define the necessary current condition and the resulting condition.

If A equals 0 the indicated action is entering the room. This value occurs for two of the solo verbs:

CAME IN
ARRIVED

These actions may only be performed if the selected character is not in the room. (If the character was dead, he would not have been selected.) The character's current condition code must be 0. Line 390 tests to see if both the current condition code (C(x)) and the verb value (A) are equal to zero. If so, the program jumps to line 440. The character's condition code is updated to a value of 1. The character is now in the room. The verb is printed, along with a period to end the sentence, and the program jumps back to line 140 to start a new sentence.

If the test of line 390 fails, a new test is performed in line 400. If the character is in the room (code = 1), he cannot perform an action with a value of 0 (A = 0). If the character is not in the room (code = 0), he cannot perform any action with a non-zero value. If either the current character condition code (C(x)) OR the verb value (A) is zero, the selected verb will be rejected. The program will jump back to line 310 to select an alternate word.

Now that we have eliminated the impossible combinations, we only need to update the character's condition code if necessary. Verbs with a value of 1 will not change the condition code. Verbs with a value of 2 indicate that the character has left the room. The condition code is set back to zero (line 410). A verb value of 3 occurs only when the character "DIES." The condition code is set to −1 (line 420) so that the character will not be selected again in the story.

Now the program skips ahead to line 450 to print out the remainder of the sentence, as already described.

If an object verb is selected, the program goes to line 470 for the second half of the sentence. First, the computer checks to make sure that there is at least one other character in the room to be the object of the current character's action (lines 470 through 520). If this test fails, the program goes to line 310 to select a solo verb.

If there is at least one other character in the room, the computer READs through the first ten items in the DATA file (lines 530 through 550), then it skips through the solo verbs (lines 560 through 580).

A random value of 1 to 15 is selected. The computer counts through the DATA file to find the indicated object verb (lines 590 through 620), and the DATA file is RESTOREd (line 630). The selected object verb is displayed in line 640.

Now the object character must be picked. A random value from 1 to 10 is selected in line 650. If this value indicates the current character (X), the choice is rejected, and a new random number is selected, line 670 checks the selected object character to make

Table 9-3. Some Typical Sentences Produced by the Second Story-Writer Program of Listing 9-2.

```
JANICE CAME IN.
ARNIE DANCED WITH ISAAC.
DIANE CRIED.
DIANE LEFT.
EGBERT LEFT.
ISAAC WATCHED.
ISAAC LISTENED TO FRANCINE.
EGBERT ARRIVED.
FRANCINE LAUGHED.
JANICE LEFT.
DIANE CAME IN.
GREG ARRIVED.
JANICE ARRIVED.
FRANCINE YAWNED.
GREG LISTENED TO JANICE.
ISAAC SAT DOWN.
ARNIE LAUGHED.
ARNIE TOUCHED BETSY.
FRANCINE GAVE A GIFT TO BETSY.
ARNIE FORGAVE GREG.
DIANE SIGHED.
JANICE YAWNED.
EGBERT STOOD.
GREG STOOD.
EGBERT WATCHED.
HARRIET CAME IN.
HARRIET KISSED DIANE.
```

sure he is in the room (code = 1). If not, the selection is rejected, and the program goes back to line 650 to try again.

Once a suitable object character has been selected, the appropriate name is displayed, along with a period to end the sentence (line 680). The program now goes back to line 140 to start the next sentence.

GOING FURTHER

A section of a typical "story" generated by this program is shown in Table 9-3. It certainly makes for dull reading, but the technique used here can be expanded as much as you like for a wide variety of possible conditions. For example, there could be some objects in the room. The characters could be carrying some of these objects. They might leave some of them outside the room. Additional codes could keep track of each characters physical condition, or mood. The possibilities are limited only by your imagination (and the memory available on your computer).

For the best results, try combining the sentence variation techniques of Listing 9-1 with the condition code technique of Listing 9-2. With some imagination and work, you should be able to come up with some passable computer generated stories.

```
10  REM   9-1     STORY WRITER-1
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
50  REM  * SET WORD COUNT VARIABLES *
60 PN = 15
70 PX = 0
80 NN = 40
90 NX = PN
100 AR = 5
110 AX = NX + NN
120 AJ = 25
130 AY = AX + AR
140 V1 = 25
150 VX = AY + AJ
160 V2 = 25
170 VY = VX + V1
180 PR = 10
190 PY = VY + V2
200 AV = 25
210 AZ = PY + PR
220  REM  * SELECT MAIN CHARACTERS *
230 X = PN
240 Y = PX
250  GOSUB 1470
260 M1$ = A$
270  GOSUB 1470
280 M2$ = A$
290  IF M1$ = M2$ THEN  GOTO 270
300  REM  * BEGIN STORY *
310  HOME
320  PRINT : PRINT
330  GOSUB 440
340 Q =  INT (1 +  RND (1) * 5)
350  IF Q = 3 THEN  PRINT " AND";: GOSUB 440
360  GOSUB 680
370 T =  INT (1 +  RND (1) * 5)
380  IF T = 3 THEN  GOSUB 1380
390  PRINT "."
400  INPUT Q$
410  GOTO 330
420  STOP
430  REM  * NOUN PHRASE GENERATOR *
440 S =  INT (1 +  RND (1) * 3)
450  ON S GOTO 460,530,570
460 X = AR
470 Y = AX
480  GOSUB 1470
490 X = NN
500 Y = NX
510  GOSUB 1470
520  RETURN
```

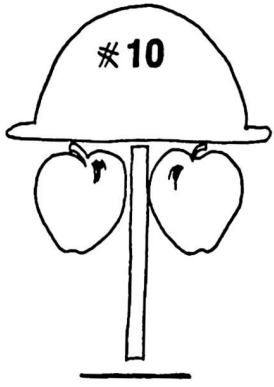
```
530 X = PN
540 Y = PX
550  GOSUB 1550
560  RETURN
570 X = AR
580 Y = AX
590  GOSUB 1470
600 X = AJ
610 Y = AY
620  GOSUB 1470
630 X = NN
640 Y = NX
650  GOSUB 1470
660  RETURN
670  REM  * COMPLETE SENTENCE *
680 S =  INT (1 +  RND (1) * 6)
690 T =  INT (1 +  RND (1) * 5)
700 X = AV
710 Y = AZ
720  IF T > 3 THEN  GOSUB 1470
730  ON S GOTO 740,780,910,1070,1170,1300
740 X = V1
750 Y = VX
760  GOSUB 1470
770  RETURN
780 X = V1
790 Y = VX
800  GOSUB 1470
810 X = PR
820 Y = PY
830  GOSUB 1470
840 X = AR
850 Y = AX
860  GOSUB 1470
870 X = NN
880 Y = NX
890  GOSUB 1470
900  RETURN
910 X = V1
920 Y = VX
930  GOSUB 1470
940 X = PR
950 Y = PY
960  GOSUB 1470
970 X = AR
980 Y = AX
990  GOSUB 1470
1000 X = AJ
1010 Y = AY
1020  GOSUB 1470
1030 X = NN
```

```
1040 Y = NX
1050  GOSUB 1470
1060  RETURN
1070 X = V2
1080 Y = VY
1090  GOSUB 1470
1100 X = AR
1110 Y = AX
1120  GOSUB 1470
1130 X = NN
1140 Y = NX
1150  GOSUB 1470
1160  RETURN
1170 X = V2
1180 Y = VY
1190  GOSUB 1470
1200 X = AR
1210 Y = AX
1220  GOSUB 1470
1230 X = AJ
1240 Y = AY
1250  GOSUB 1470
1260 X = NN
1270 Y = NX
1280  GOSUB 1470
1290  RETURN
1300 X = V2
1310 Y = VY
1320  GOSUB 1470
1330 X = PN
1340 Y = PX
1350  GOSUB 1550
1360  RETURN
1370  REM  * EXTRA PHRASE *
1380  PRINT " WITH";
1390 X = AR
1400 Y = AX
1410  GOSUB 1470
1420 X = NN
1430 Y = NX
1440  GOSUB 1470
1450  RETURN
1460  REM  * WORD SELECT *
1470 Z =  INT (1 +  RND (1) * X) + Y
1480  FOR T = 1 TO Z
1490  READ A$
1500  NEXT T
1510  PRINT " ";A$;
1520  RESTORE
1530  RETURN
1540  REM  * MAIN CHARACTER *
```

```
1550 Z =  INT (1 +  RND (1) * 5)
1560  IF Z > 2 THEN  GOTO 1470
1570  IF Z = 1 THEN A$ = N1$
1575  IF Z = 2 THEN A$ = N2$
1580  GOTO 1510
1590  REM  * PROPER NOUNS -- PN *
1600  DATA  HERMAN,WALDO,LADY BUCKLEY,ORVILLE,CINDY
1610  DATA  GLADYS,ISAAC,AGATHA,RALPH,OLIVE
1620  DATA  QUAG,BONZO,HUBERT,SIDNEY,SUZIE
1630  REM  * NOUNS -- NN *
1640  DATA  ROCK,FROG,KUMQUAT,FLOWER,DOOR
1650  DATA  COMPUTER,RODENT,DIRTY SOCK,FEATHER,BOOK
1660  DATA  CAT,TURKEY,MUSHROOM,FRIEND,STRANGER
1670  DATA  SHOE,SANDWICH,PENCIL SHARPENER,NECKTIE,RULER
1680  DATA  PENNY,LAMPSHADE,NOTEPAD,BOWL,SPOON
1690  DATA  MAN,WOMAN,BOY,GIRL,COP
1700  DATA  CROOK,DRUNK,BARBER,COWBOY,HORSE
1710  DATA  HAMMER,SCREWDRIVER,BEER BOTTLE,ROBOT,CHAIR
1720  REM  * ARTICLES -- AR *
1730  DATA  THE,A,HIS,HER,THE
1740  REM  * ADJECTIVES -- AJ *
1750  DATA  BIG,TINY,SMELLY,DAMAGED,DIRTY
1760  DATA  CUTE,HUGE,CLUMSY,SEXY,DAINTY
1770  DATA  RED,DISGUSTING,YELLOW,STRANGE,SLIMY
1780  DATA  MYSTERIOUS,ORANGE,DREADFUL,THROBBING,PLASTIC
1790  DATA  PURPLE,GREEN,NEW,MECHANICAL,SOFT
1800  REM  * VERBS 1 -- V1 *
1810  DATA  WENT,CAME,LEFT,RETURNED,GREW
1820  DATA  QUIVERED,COUGHED,WAITED,DREAMED,FELL
1830  DATA  SAT,WANDERED,THROBBED,HUMMED,STOPPED
1840  DATA  FLEW,CRAWLED,SLOWED DOWN,BROKE,GROANED
1850  DATA  WEPT,LAUGHED,YODELED,COMPLAINED,GLISTENED
1860  REM  * VERBS 2 - V2 *
1870  DATA  TOUCHED,HID,FOUND,LOVED,FOLLOWED
1880  DATA  TICKLED,FELT,CARRIED,FONDLED,KICKED
1890  DATA  ATE,THROTTLED,SMELLED,HIT,KISSED
1900  DATA  ADMIRED,CRITICIZED,IRRITATED,STROKED,ATTACKED
1910  DATA  FOUGHT,AMUSED,MET,LICKED,CONFUSED
1920  REM  * PREPOSITIONS -- PR *
1930  DATA  ON,UNDER,WITH,BEHIND,IN FRONT OF
1940  DATA  OVER,NEAR,BY,FAR FROM,CLOSE TO
1950  REM  * ADVERBS -- AV *
1960  DATA  QUICKLY,SUDDENLY,LANGUIDLY,ERRATICALLY,DESPERATELY
1970  DATA  CAREFULLY,LOUDLY,SWEETLY,CLUMSILY,CHARMINGLY
1980  DATA  GENTLY,TENDERLY,CRUDELY,SEDUCTIVELY,SNEAKILY
1990  DATA  SILENTLY,CARELESSLY,OBSCENELY,LAZILY,WEIRDLY
2000  DATA  HASTILY,SLOPPILY,HAPPILY,HUNGRILY,SEXILY
```

```
10  REM   9-2     STORY WRITER-2
20  REM  * DELTON T. HORN
30  REM  * COPYRIGHT 1986
40  HOME
50  DIM C$(10)
60  DIM C(10)
70 K = 0
80  FOR X = 1 TO 10
90 C(X) = 0
100  READ A$
110 C$(X) = A$
120  NEXT
130  RESTORE
140 K = K + 1
150  IF K < 12 THEN  GOTO 210
160  PRINT
170  PRINT "PLEASE PRESS <RETURN>"
180  INPUT Q$
190  HOME
200 K = 0
210 Z = 0
220  FOR T = 1 TO 10
230 Z = Z + C(T)
240  NEXT T
250  IF Z <  - 3 THEN  GOTO 700
260 X =  INT (1 +  RND (1) * 10)
270  IF C(X) =  - 1 THEN  GOTO 260
280  PRINT C$(X);" ";
290 Y =  INT (1 +  RND (1) * 3)
300  IF Y = 2 AND C(X) = 1 THEN  GOTO 470
310  FOR T = 1 TO 10
320  READ A$
330  NEXT T
340 Z =  INT (1 +  RND (1) * 15)
350  FOR T = 1 TO Z
360  READ A$,A
370  NEXT T
380  RESTORE
390  IF A = 0 AND C(X) = 0 THEN  GOTO 440
400  IF A = 0 OR C(X) = 0 THEN  GOTO 310
410  IF A = 2 THEN C(X) = 0
420  IF A = 3 THEN C(X) =  - 1
430  GOTO 450
440 C(X) = 1
450  PRINT A$;"."
460  GOTO 140
470 Z = 0
480  FOR T = 1 TO 10
490  IF T = X THEN  GOTO 510
500  IF C(T) > 0 THEN Z = Z + 1
510  NEXT T
```

```
520  IF Z < 2 THEN  GOTO 310
530  FOR T = 1 TO 10
540  READ A$
550  NEXT T
560  FOR T = 1 TO 15
570  READ A$,A
580  NEXT T
590 Z =  INT (1 +  RND (1) * 15)
600  FOR T = 1 TO Z
610  READ A$
620  NEXT T
630  RESTORE
640  PRINT A$;" ";
650 Y =  INT (1 +  RND (1) * 10)
660  IF Y = X THEN  GOTO 650
670  IF C(Y) < 1 THEN  GOTO 650
680  PRINT C$(Y);"."
690  GOTO 140
700  PRINT "","THE END"
710  END
720  REM  * CHARACTERS *
730  DATA  ARNIE,BETSY,CHUCK,DIANE,EGBERT
740  DATA  FRANCINE,GREG,HARRIET,ISAAC,JANICE
750  REM  * SOLO VERBS *
760  DATA  CAME IN,0,ARRIVED,0,LAUGHED,1,CRIED,1,SANG,1
770  DATA  SAT DOWN,1,STOOD,1,WENT TO SLEEP,1,SIGHED,1,WATCHED,1
780  DATA  YAWNED,1,YODELLED,1,LEFT,2,WENT OUT,2,DIED,3
790  REM  * OBJECT VERBS *
800  DATA  TALKED WITH,SHOOK HANDS WITH,DANCED WITH,
     KISSED,LOOKED AT
810  DATA  LISTENED TO,HIT,TOUCHED,LAUGHED AT,SEDUCED
820  DATA  FOUGHT WITH,YELLED AT,APOLOGIZED TO,FORGAVE,
     GAVE A GIFT TO
```



Programming Languages

A number of AI programming techniques have been presented throughout this book. Now you are faced with several decisions. Do you just run the programs as they are presented in the listings, or do you make a few customized modifications to suit your individual needs and interests? Do you go further to apply the techniques discussed in this book to new programs of your own? Do you use these techniques as starting points for your own experimentation? I certainly hope so. We have just barely been able to scratch the surface of AI programming in this book. There are a lot of additional ideas to try.

If you do go on to create your own programs, you will first have to deal with one of the toughest decisions of all. Which programming language do you use?

BASIC

As mentioned earlier, I selected BASIC for the sample programs of this book because of its relative universality. The programs are easy to read, and it is reasonable to assume that the majority of my readers are already familiar with BASIC. This is not true of any other programming language.

BASIC is a very transportable language. It would not be at all difficult to modify the programs to run on almost any microcomputer. While this book concentrates on the Apple II, it is not limited to this machine. Only a few lines will have to be re-written to run these programs on most popular microcomputers. While BASIC is convenient, especially for illustrating the principles involved, it unquestionably leaves a lot to be desired.

The two biggest problems facing the AI programmer are memory space and running speed. Unfortunately, BASIC doesn't do too well in either area. BASIC commands tend to eat up memory fairly quickly. Even more noticeable is the problem of speed. BASIC is notoriously slow.

The problem is that BASIC is an interpreted, rather than a compiled language. Each time any command is to be executed, it must be translated into machine language. Translating any individual command only takes a few microseconds, but when there are several hundreds (or thousands) of commands to be translated, the cumulative time adds up.

In some of the simpler programs, the slow speed is scarcely noticeable. In some cases we even added some delay loops to deliberately slow things down. In other programs, the speed is a significant problem. It is especially noticeable in the "MAD DOCTOR" program in Chapter 7. It takes the computer a long time to decide on its reply to each user input (especially if the user inputs a long sentence).

BASIC has also earned some criticism for being an unstructured language. I don't feel this is necessarily a problem. BASIC programs can be structured, they just don't HAVE to be. Admittedly, it is easy to get a little sloppy in BASIC, and most BASIC programmers develop a few bad programming habits. (I'm sure some programmers will find fault with some of the programs presented in this book.) If you are the type who prefers a clear-cut structure in a programming language. BASIC is probably not the best choice.

BASIC was not designed with AI applications in mind, so some procedures can be a little awkward. Not impossible—just inelegant. A lot of debugging time can be wasted trying to get a programming language to perform a task it wasn't designed for.

All in all, BASIC is not really a strong choice for AI programming. It will do the job, but just barely. The only really valid reason to choose BASIC for this type of work is if it is the only available choice. Many microcomputer programmers only know BASIC. On a number of popular microcomputers, it is the only high-level language offered. On others, translation programs for other languages, such as Pascal or LISP are available, but are often expensive.

A casual experimenter may not want to invest the time in learning a new programming language. There is nothing wrong with using BASIC in such cases. No law says all AI programming must be in LISP. But, if the choice is available, BASIC runs a poor second.

PASCAL

Some programmers may turn to Pascal as a sort of compromise. Some Pascal enthusiasts may hate me for this, but Pascal is almost a "Super-BASIC." In many ways, the two languages are quite similar.

The primary advantages of Pascal are:

- ☆ Pascal is a compiled language. This means the commands are translated once, before the program runs. Since no translation has to take place "on the fly," Pascal programs tend to run much faster than comparable BASIC programs. Memory consumption is also reduced.
- ☆ Pascal is a structured language. Sloppy, unreadable code is almost impossible.

☆ Pascal permits the programmer to define new commands (procedures) as needed.

Pascal is not the strongest possible programming language for AI work, but it can offer some significant advantages over BASIC.

ASSEMBLY LANGUAGE

Strictly in terms of memory consumption and running speed, assembly (machine) language wins hands down. The microcomputer is programmed directly, using the commands understood by the CPU. This permits great efficiency in the programming.

Assembly language isn't always a desirable choice. It is difficult to learn t well. The code is hard to read, making changes and debugging difficult and te It is all too easy for errors to sneak into the program.

It probably wouldn't be worthwhile to learn assembly language just to exper with artificial intelligence programs. But if you are already adept at programm assembly language, you certainly should consider applying this skill to your experi in artificial intelligence.

LISP

Without a doubt, the most popular programming language for serio experimentation is LISP, or LISt Processing. This language was specifically des for the manipulation of symbolic expressions, such as words. Most other progran languages are at their best when dealing with numerical values.

No attempt will be made here to teach the reader LISP programming. That take at least an entire book in itself. Instead, we will just briefly cover the bare of the language.

A LISP program manipulates lists of items. Each item in the list is called an "a One list can be contained within a larger list. In fact, the whole LISP program is essentially just a list. It is not even difficult to write a LISP program that new LISP programs.

One big advantage of LISP in many AI programs is that it is a recursive lang A list can be defined in terms of itself. Recursion is a tricky concept, and we can't afford to go into it here, but to give you an idea of what I'm talking about consider the following statement:

"THIS SENTENCE IS RECURSIVE."

The statement refers to itself. The phrase "THIS SENTENCE" refers to the entire sentence, i.e., "THIS SENTENCE IS RECURSIVE." In other words, the sentence can be rewritten like this:

" 'THIS SENTENCE IS RECURSIVE' IS RECURSIVE."

This can be extended indefinitely in an endless loop:

" ' " 'THIS SENTENCE IS RECURSIVE' IS RECURSIVE" IS RECURSIVE' IS RECURSIVE."

LISP functions can be combined to create new functions. The language comes equipped with five primitive functions:

ATOM	COND
CAR	CONS
CDR	EQ
	LAMBDA

None of these functions are very complex, although some of them are a little difficult to explain in this brief space. ATOM defines an atom. A new atom may be a list. EQ is a test for equality. The other commands manipulate and combine atoms and lists in various ways. These simple functions can be combined in an infinite number of ways to perform almost any operation.

A full version of LISP requires an extensive amount of memory. It is usually implemented on large mainframe computers. However, stripped down (but still very powerful) versions of LISP have been made available for a number of popular microcomputers.

Whatever programming language you decide to use, I'm sure you will find experimenting with artificial intelligence extremely fascinating and educational.

Index

A

advanced games, 83-103
  minimax method for, 87
ADVAV, 4
adventure games, 111
art, computer generated, 149
artificial intelligence, 1-11
  applications for, 10
artificial languages, 5
ASCII code, 9
assembly language, 189
automated translation, 112
AVIDAC, 4

B

Babbage, Charles, 3
BASIC, 187
BASIC INPUT statement, 122
bits, bytes, and nybbles, 8
Boolean tests, 16
breadth-first search, 33
BREADTH-FIRST TREE SEARCH
  program, 35
brute force method of search, 28

C

chess, 88
children, 30
Colby, Kenneth, 136
compiled languages, 188
computer, 3
computer aided design (CAD), 10
computer creativity, 149-172
computer error, 10
computer thought, 8
condition codes, 177
Conversation I, 111-134
  program listing for, 127-129
  sample run of, 121
Conversation II, 122
  program listing for, 130-134
conversation programs, 135-148
conversational computers, 115
counting loops, 117

D

decision making, 12-48
  simple, 13
depth of search, 92
depth-first search, 31
DEPTH-FIRST TREE SEARCH
  program, 34
  listing for, 37-48

E

electric tabulator, 4
ELIZA, 5, 135
ENIAC, 4
errors, 10
expert systems, 107

F

factors, 19
fanout parameters, 90
FRUSTRATION, 55-58
  program listings for, 67-77

G

game playing, 5
game theory, 49-82
general problem solver program, 5
goal, 21
GOMOKU, 84-87
  program listing for, 95-103

H

Haiku generators, 151
HAIKU, 153
  program listing for, 160-165
heuristics, 89
Hollerith, Herman, 4

I

illegal states, 23
ILLIAC, 4
initial state, 21
internal timer/counter, 12
interpreted languages, 188

K

KALA, 49-55
  program listing for, 61-66
keyword search, 117

L

labels, 19
Leibniz, Gottfried, 3
LISP, 188, 189

M

MAD DOCTOR, 136-148
Mark I, 4
mass storage, 105
MAZE program, 26
memory, 105
MI program, 107
minimax method, 87
MUSIC COMPOSITION, 153, 154
  program listing for, 166-168

N

natural language processing, 5, 112
Newell, Allen, 5
nodes, 27
nodes and children, 30
nonsense statements, 106
nouns, 175

O

OPTION WEIGHER program, 20
option weighting, 19

P

parameters, fanout, 90
PARRY, 6, 136
Pascal, 188
Pascal, Blaise, 3
pattern recognition, 89
plausibility scores, 91
POETRY GENERATOR, 151-153
  program listing for, 158-159
poetry generators, 150
problems, definition of, 19
programming languages, 187-190
programs
  BREADTH-FIRST TREE SEARCH, 35
  CONVERSATION I, 127-129
  CONVERSATION II, 130-134
  DEPTH-FIRST TREE SEARCH, 34, 37-48
  ELIZA, 135
  FRUSTRATION, 55-58, 67-77
  GOMOKU, 84-87, 95-103
  HAIKU, 153, 160-165
  KALA, 49-55, 61-66
  MAD DOCTOR, 136-148
  MAZE, 26
  MUSIC COMPOSITION, 154, 166-168
  OPTION WEIGHER, 20
  PARRY, 136
  POETRY GENERATOR, 151-153, 158-159
  STORY WRITER I, 173, 174, 180-183
  STORY WRITER II, 177, 179
  STORY WRITER II, 184-185
  TIC-TAC-TOE, 59-60, 78-82
  TREE SEARCH, 27
  TUNE IMITATOR, 156, 169-172
pronouns, 124
psychotherapist program, 135

R

random number generation, 12, 149
roots, 27

S

Samuels, Arthur, 5
search tree, 88
searching
  breadth-first, 33
  depth-first, 31
seed value, 12
semantics, 113
SHRDLU, 6
SILLIAC, 4
Simon, Herbert, 5
simulations, 107
state graphs, 25
STORY WRITER I, 173, 174
  program listing for, 180-183
STORY WRITER II, 177, 179
  program listing for, 184-185
story-writing programs, 173-186
  advanced, 179
sub-goals, 22
substrings, 118
swap routine, 124
syntax, 113

T

TIC-TAC-TOE, 59-60
  program listings for, 78-82
TREE SEARCH program, 27
tree searches, 26
truth tables, 17
TUNE IMITATOR, 156
  program listing for, 169-172
Turing Test, 104-110
  practical limitations for, 105
Turing, Alan, 104
two dimensional array, 155

V

verbs, 175
visual capabilities, 105

W

weighted selection process, 116
weighting, 155
Weizenbaum, Joseph, 5, 135
Winograd, Terry, 6

Edited by David Gauthier

Other Bestsellers From TAB

☐ **80386—A PROGRAMMING AND DESIGN HANDBOOK—Penn Brumm and Don Brumm**

The basis of IBM's much-anticipated OS/2 operating system and their new Personal System/2 computers, the 80386 microprocessor promises new standards in microcomputer power, speed, and versatility. Now, with the cooperation of 80386 designers from Intel Corporation, Penn and Don Brumm have provided the first complete sourcebook on this advanced processor, including an overview of its capabilities and in-depth information for programmers and designers. 448 pp., 150 illus.

Paper $19.95 **Hard $29.95**
Book No. 2937

☐ **ADVANCED PROGRAMMING WITH dBASE III® PLUS—Cary N. Prague and James E. Hammitt**

This sequel to Prague and Hammitt's bestselling *Programming with dBASE III PLUS* leads you through all of the advanced new features offered by dBASE, with even greater emphasis on the use of macros and other power programming techniques. You'll learn how to use your software's improved options to produce more impressive looking screens and reports and to handle more sophisticated database applications with ease. 352 pp., 262 illus.

Paper $19.95 **Hard $24.95**
Book No. 2876

☐ **PRODOS® INSIDE AND OUT—Dennis Doms and Tom Weishaar**

This introduction to programming with BASIC.SYSTEM gives practical tips and how-to advice on everything from booting your system to assembly language programming with ProDOS. You'll even cover such hard-to-find topics as subdirectories and their use, programming examples for the Field and Byte options of text files, the use of BSAVE and BLOAD parameters to save disk space, and more! 270 pp., 113 illus.

Paper $16.95 **Hard $24.95**
Book No. 2745

☐ **MIS MANAGER'S HANDBOOK: Innovative Strategies for Successful Management—Mark C. Wargo**

This unique collection of nearly 50 essays is designed to fill that need by helping these managers stay on the cutting edge of all the latest technology and managerial concepts. Packed with essential information for managers, supervisors, project leaders, telecommunications managers, MIS users, data processors, information processing consultants, or anyone who must accommodate computerization as it becomes a part of his business. 400 pp., 32 illus.

Hard $34.50 **Book No. 2740**

☐ **SYSTEMS DESIGN UNDER CICS COMMAND AND VSAM—Alex Varsegi**

Here is a comprehensive summary of CICS functions, design considerations, and related software products acquaint you with the concept of on-line data processing and its established role in current computer design. You'll cover system design, screen painting techniques using SDF to create input/output maps, the use of CECI, VSAM (virtual storage access method), all the CICS commands and how they relate to system design. 272 pp., 204 illus.

Hard $28.95 **Book No. 2843**

☐ **THE GEM™ OPERATING SYSTEM HANDBOOK—Dave Prochnow**

Here's your chance to discover how GEM can turn your IBM PC/XT/AT or compatible into a Macintosh-like machine with all the easy-use features of mouse and icon control without sacrificing the power, expandability, and large software choice offered by your IBM. You'll be introduced to GEM's portable programming concepts, the GEM programmer's tool kit, and GEM's application programs including GEM Draw, GEM Paint, and GEM Write. 256 pp., 137 illus.

Paper $16.95 **Hard $23.95**
Book No. 2742

☐ **30 CUSTOMIZED MICROPROCESSOR PROJECTS—Delton T. Horn**

Here it is! The electronics project guide that you've been asking for—a complete sourcebook on designing and building special purpose computer devices around the Z80 microprocessor! Includes building instructions, detailed schematics, and application programs for 30 intriguing and useful dedicated CPU projects, plus how to customize these devices for your own individual applications! 322 pp., 211 illus.

Paper $14.95 **Hard $22.95**
Book No. 2705

☐ **DESIGNING AND PROGRAMMING PERSONAL EXPERT SYSTEMS—Carl Townsend and Dennis Feucht**

Discover how new trends in artificial intelligence (AI) concepts can be put to practical use on almost any personal computer including Apple® II or IBM® PC! Explore expert system programming techniques to create your own system for electronics, engineering, or other technical applications! It's all here for the taking in this exciting and challenging new sourcebook! 250 pp., 75 illus.

Paper $18.95 **Hard $27.95**
Book No. 2692

Other Bestsellers From TAB

☐ **THE ILLUSTRATED HANDBOOK OF DESKTOP PUBLISHING AND TYPESETTING—Michael L. Kleper**

Desktop publishing is not only the newest and hottest topic in computers, it is a dazzling and useful tool that allows you to easily perform printing tasks previously possible only with traditional printing methods! Now, one of the nation's top authorities in this emerging field has written the first comprehensive analysis of desktop publishing—a book that all those who have seen believe will be *the definitive sourcebook* on the subject! 784 pp., 615 illus.

Paper $29.95 **Hard $49.95**
Book No. 2700

☐ **ADVANCED APPLEWORKS™—David Bolocan**

Advanced AppleWorks is designed to have you mastering *AppleWorks* in as little as time as possible. Bolocan provides practical insight into AppleWork's unique features and capabilities and how you can apply them in a variety of ways. In fact, each chapter presents a sophisticated, real-life AppleWorks application, tips on typing, inserting, and editing text, formatting pages, creating form letters and reports, and more! 256 pp., 190 illus.

Paper $16.95 **Book No. 2648**

☐ **LISP—THE LANGUAGE OF ARTIFICIAL INTELLIGENCE—Frederick Holtz**

Now here's your opportunity to learn and use LISP . . . to enter the realm of artificial intelligence with confidence. Holtz explains LISP vocabulary and how artificial intelligence programming concepts differ. You'll get a look at how LISP handles mathematical operations . . . be introduced to logical operators . . . and see how close BASIC input/output is to that of Common LISP. 272 pp.

Paper $16.95 **Hard $25.95**
Book No. 2620

☐ **NUMERICAL METHODS FOR ENGINEERS AND SCIENTISTS: PRACTICAL APPLICATIONS AND METHODS USING THE APPLE™ II® SERIES —Robert D. Walker**

This collection of ready-to-run mathematics programs and subroutines is ideal for solving typical problems encountered in engineering and scientific applications. It's a professional tool that you'll find indispensable for everyday use. Gives you a ready-to-use library of numerical analysis programs written in Applesoft BASIC. Fully documented and tested. 304 pp.

Hard $16.95 **Book No. 2647**

☐ **ONLINE RESEARCH AND RETRIEVAL WITH MICROCOMPUTERS—Nahum Goldmann**

This time-saving guide shows you how to turn a micro into an invaluable research "tool" for digging up information from databases across the country. Using Nahum Goldmann's "Subject Expert Searching Technique," businessmen, engineers, physicians, lawyers, professors, and students can quickly and easily retrieve information in the comfort of their work station. 208 pp., 119 illus.

Hard $25.00 **Book No. 1947**

☐ **MODULAR PROGRAMMING ROUTINES FOR THE APPLE® II+/IIe/IIc—Bruno B. Wolff, Jr.**

Using Wolff's interchangeable collection of programming routines in Applesoft® BASIC, you can design an unlimited number of programs. Included are master input routines, output routines, sorting routines, and error-protection routines that take advantage of the Apple's special features—sound and music routines, graphics routines, simulation routines, even a pseudo word processor! 256 pp., 19 illus.

Paper $14.95 **Hard $23.95**
Book No. 1732

*Prices subject to change without notice.

Look for these and other TAB books at your local bookstore.

TAB BOOKS Inc.
P.O. Box 40
Blue Ridge Summit, PA 17214

Send for FREE TAB catalog describing over 1200 current titles in print.

Or Call For Immediate Service 1-800-233-1128

So that TAB BOOKS Inc. can better fill your reading needs . . .

please take a moment to complete and return this card. We appreciate your comments and suggestions.

1. I am interested in books on the following subjects:

- ☐ automotive
- ☐ aviation
- ☐ business
- ☐ computer, hobby
- ☐ computer, professional
- ☐ engineering (specify) _____
- ☐ other (specify) _____
- ☐ other (specify) _____
- ☐ electronics, hobby
- ☐ electronics, professional
- ☐ finance
- ☐ how to, do-it-yourself

2. I own/use a computer:

- ☐ IBM _____
- ☐ Apple _____
- ☐ Commodore _____
- ☐ Macintosh _____
- ☐ ATARI _____
- ☐ AMIGA _____
- ☐ Other (specify) _____

3. This card came from TAB book (specify title and/or number):

_____

4. I purchase books:

- ☐ from general bookstores
- ☐ from technical bookstores
- ☐ from college bookstores
- ☐ other (specify) _____
- ☐ through the mail
- ☐ by telephone
- ☐ by electronic mail

Comments _____

_____

_____

_____

_____

_____

_____ Zip _____

Smart Apples: 31 Artificial Intelligence Experiments with the Apple II®, II+®, IIe®, IIc®, and IIGS®

by Delton T. Horn

Featuring programs that can turn your computer into an intelligent competitor, a witty conversationalist, an artist, poet, musician, or writer!

If you're fascinated by artificial intelligence (AI) techniques and concepts, but assume you need a mainframe computer to do any meaningful AI experiments . . . here's a book that you won't want to miss. It's your opportunity to enter the world of artificial intelligence using an ordinary Apple micro!

Horn includes an introduction to intelligence research and covers the milestones in AI development. You'll cover such intriguing topics as the Turing Test, game applications, computer-generated stories, and computer creativity. Famous AI programs that simulate artificial intelligence—such as ELIZA and PARRY—are also covered.

Practical artificial intelligence program listings are included that enable you to perform hands-on artificial intelligence experiments. Best of all, these programs are not only fun, but will also help you improve your programming techniques and learn more about how the mind works. The programs are written in BASIC and aimed toward the popular Apple microcomputers; however, they are easily adapted to run on almost any microcomputer that uses any version of Microsoft BASIC.

Programs are included that will turn your computer into a tough competitor for games, a musician, a witty conversationalist, a poet, a composer, and a writer. These programs include: • TIC-TAC-TOE • GOMUKO • SIMPLE CONVERSATION • IMPROVED CONVERSATION • ADVANCED CONVERSATION • "THE MAD DOCTOR" • SIMPLE POETRY GENERATOR • WEIGHTED RANDOM COMPOSER • STORY WRITER.

Delton T. Horn is a computer and electronics expert whose many popular books for TAB include *Designing IC Circuits . . . With Experiments, 30 Customized Microprocessor Projects*, and *34 More Tested and Proven Game Programs in BASIC*.

Send for FREE TAB Catalog describing over 1200 current titles in print.

TAB

TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

FPT > $12.95

ISBN 0-8306-2775-8

PRICES HIGHER IN CANADA 0887