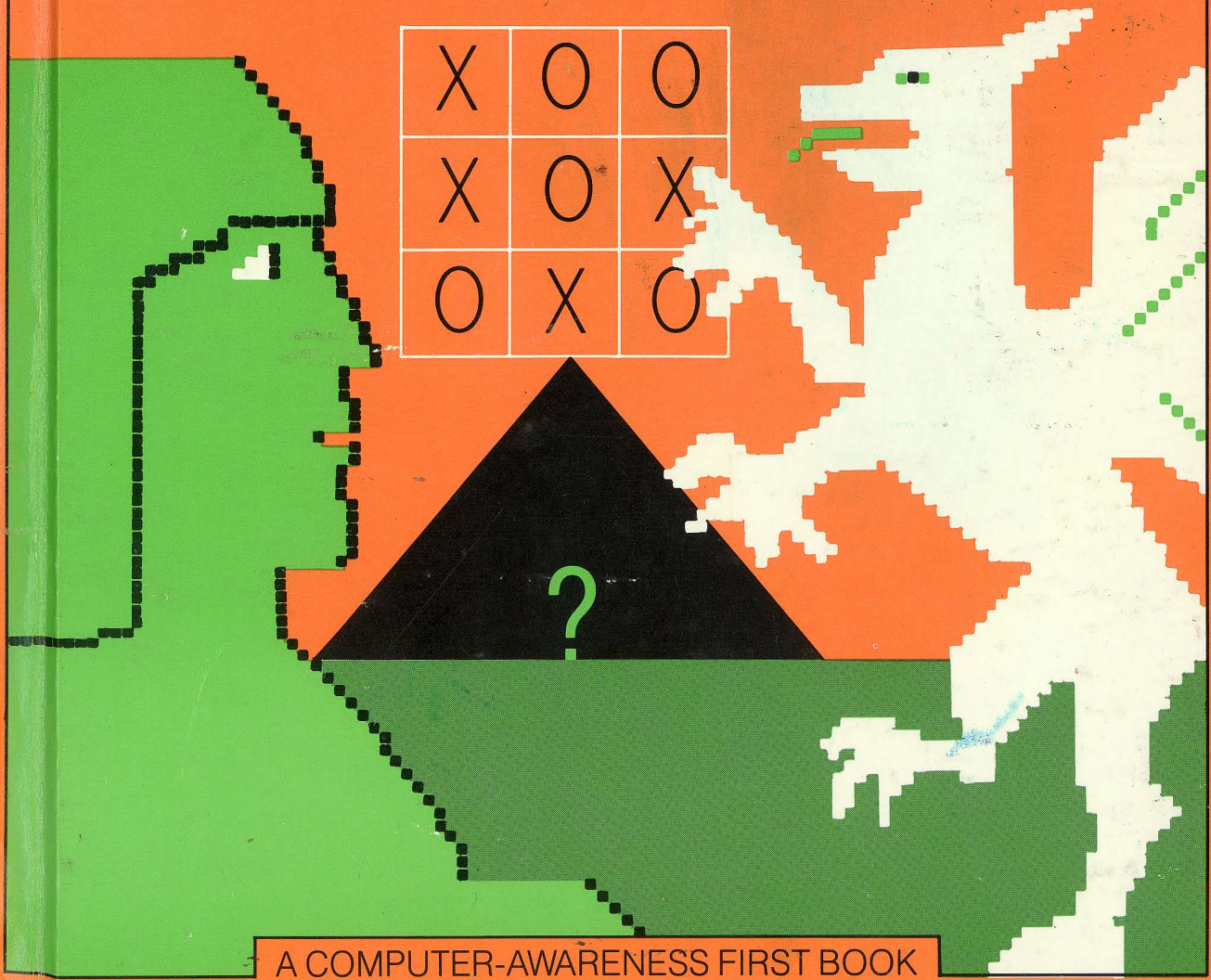
FRED D'IGNAZIO

# Invent Your Own Computer Games




A COMPUTER-AWARENESS FIRST BOOK

FRED D'IGNAZIO

# INVENT YOUR OWN COMPUTER GAMES

A Computer-Awareness
First Book

Consulting Editor
Melvin Berger

Franklin Watts
New York | London | Toronto
Sydney | 1983

Library of Congress Cataloging in Publication Data

D'Ignazio, Fred
Invent your own computer games.

(A Computer-awareness first book)
Bibliography: p.
Includes index.
Summary: Explains how to use a personal computer to create original sports, board, strategy, word, number, and adventure games, with instructions for programming and playing eleven simple computer games.
1. Computer games—Juvenile literature.
2. Basic (Computer program language)
—Juvenile literature.
[1. Computer games. 2. Games.
3. Basic (Computer program language)
4. Programming (Computers)
5. Minicomputers—Programming]
I. Title. II. Series.
GV1469.2.D54 1983 001.64'2 83-10387
ISBN 0-531-04637-0



10 9

# CONTENTS

# INVENT YOUR OWN COMPUTER GAMES

For Felicity, Hope,
Hugh, and Charles

# CHAPTER ONE

## BUILD A GAME ARCADE AT HOME

Do you spend most of your allowance on electronic games at the local arcade? Or on new game cartridges for your home video system?

If you answered yes, here's a suggestion: Why not invent your own games?

You will need two things to get started. First, you will need a personal computer—not a video-game machine, but a real computer with a keyboard full of letters and numbers.

The second thing you will need is an introduction to game programming. That's what this book is for. An introduction to game programming will give you the pointers you need to build a simple game of your own.

Video games are "black boxes." You never see how they work. You just plug in a game cartridge and begin playing.

Personal computers are different. You can still plug in "canned" games, but you can also create your own. You create a game by writing a *program.* A program is a list of instructions to the computer in a language it can understand. When you type **RUN**, the computer will obey the instructions, or *commands,* in order, one at a time.

## FINDING A PERSONAL COMPUTER

Maybe you already have a personal computer. If not, there is a good chance that your school has one—or more.

No? Well, don't despair. Some personal computers are now cheaper than video-game machines. One company, for example, offers a personal computer for less than $60.

Video-game systems range from $50 to $200. And that's without expensive game cartridges. Each cartridge will cost you an additional $15 to $50. But if you make up games on your own, you won't need cartridges.

Maybe you have a birthday coming up or a paper route. You can probably get help from your parents if you remind them that personal computers aren't toys. By learning to program, you are acquiring a valuable skill. One day you might be able to turn that skill into a high-paying career.

In the meantime, you don't have to program just games. You can use the computer to help you with your schoolwork and maybe even write programs to help your family.

## DIFFERENT VERSIONS OF BASIC

In this book, you'll learn to invent games of your own using a computer language called *BASIC*. BASIC is the most popular language for small, personal computers.

All the games in this book were written on an Atari 800 computer, using the Atari company's version of BASIC. However, with a few exceptions, only those commands that are common to almost all popular computers were used. This means that all the programs in this book should run on all the popular computers, with few, if any, changes.

There are lots of popular computers—Apples, Ataris, Com-

modores, Timex/Sinclairs, TIs, Radio Shacks, and so on. Each of these computers has its own version of BASIC. Most versions are extremely similar, but they are not exactly alike.

If a certain command will not work on your computer, check the BASIC manual that came with the computer. Most likely, you will find the same command, but it will look slightly different from the one that appears in this book. If it does look different, you should change the command so that it is exactly like the one in your BASIC manual. Then test your program with the new version of the command.

One command in this book that may give you trouble is the assignment command. We will explore this command in the next chapter. For now, just remember this: If the assignment commands in the game programs do not work, try putting the word **LET** before them.

## CREATING YOUR OWN GAMES

Inventing your own games will make you aware of how much effort it takes to create commercial games like those stored on the microchips inside game cartridges and in arcade games. A good commercial game might take a programmer weeks or even months to write and consists of *thousands* of computer commands.

But our games will be short and simple. You should be able to type most of them into the computer in less than half an hour. Most of the game programs will be no more than twenty or thirty lines long. Each line will have only one or a few computer commands.

If you type in the programs in this book, you'll have ten computer games of your own. And these games will multiply like rabbits. Each time you play a game, you'll think of new features you can add to make it even more challenging and exciting. You'll also find yourself thinking up totally new games.

Before you know it, you'll be the "manager" of your own game arcade. If you get tired of the games in your arcade, you can just invent new ones. And it won't cost you a dime.

## GRAPHICS AND SOUND

As you learn more about your particular computer, you might want to experiment with creating special effects. Perhaps you can program the computer to make monster faces on the video screen, for example, and to make horrible, scary sounds.

Graphics and sound commands are fun to use, but they are usually different for each kind of personal computer. For example, graphics and sound commands that work on an Atari computer would be different from the commands that work on an Apple, a Radio Shack, a Texas Instruments, or a Commodore computer. At the end of this book, in the "For Further Reading" section, you will find listed some books that explain how to create graphics and sound commands on different computers.

# CHAPTER TWO

## GUESS THE NUMBER

To play the games in this book, you will need to do two things. First, you will need to type the commands, exactly as shown, into the computer. Second, to get the computer to play the game, you will have to type in **RUN** at the end of your list of instructions.

When you enter a program into a computer, you must type in line numbers before your commands. Then, when you finish writing the line, you press the **ENTER** (or, on some computers, the **RETURN**) key. This stores the line in the computer's memory. In fact, after typing any command or entering any message into the computer, you should press the **ENTER** key.

The first game program is short, simple, and easy to type into the computer. But it will teach you a lot.

Here it is:

```
50  REM *** GUESS THE NUMBER
60  DIM A$(1)
70  DIM NM$(15)
100 PRINT "WHAT IS YOUR NAME";:INPUT NM$
110 LET N=INT (RND(1)*20)+1
120 PRINT
130 PRINT NM$;", I'M THINKING OF A"
135 PRINT "NUMBER BETWEEN 1 AND 20."
```

```
138 PRINT
140 PRINT "WHAT IS MY NUMBER";:INPUT G
150 IF G<>N THEN 300
160 PRINT "          "
170 PRINT "HURRAY,   ";NM$;"!"
180 PRINT "YOU GUESSED MY NUMBER."
190 FOR T=1 TO 300:NEXT T
200 PRINT
210 PRINT NM$;", DO YOU WANT TO"
220 PRINT "PLAY AGAIN";:INPUT A$
230 IF A$="Y" THEN 110
240 IF A$<>"N" THEN 200
250 PRINT :END
300 PRINT
310 IF G>N THEN 350
320 PRINT "SORRY, ";NM$;".   TOO LITTLE!"
330 GOTO 138
350 PRINT "SORRY, ";NM$;".   TOO BIG!"
360 GOTO 138
```

This will be the biggest chapter in the book. In it, we'll introduce most of the commands you'll need to write game programs of your own. Later chapters will be shorter because you'll already know most of the commands. When you speak, for example, you use the same words in different ways to make new sentences. In the same way, when you write new programs, you will use the same commands in different ways to make new games.

In this game, the computer thinks of a number, and you have to guess it. Let's pretend that you and the computer are playing the game. We'll say your name is Marvin.

## SAMPLE GAME

```
Computer: READY
     You: RUN
```

```
Computer: WHAT IS YOUR NAME?
     You: MARVIN
Computer: I'M THINKING OF A NUMBER BETWEEN
          1 AND 20. WHAT IS MY NUMBER?
     You: 10
Computer: SORRY, MARVIN. TOO LITTLE! WHAT IS
          MY NUMBER?
     You: 17
Computer: SORRY, MARVIN. TOO BIG! WHAT IS
          MY NUMBER?
     You: 15
Computer: HURRAY, MARVIN! YOU GUESSED
          MY NUMBER.
Computer: DO YOU WANT TO PLAY AGAIN?
     You: NO
Computer: READY
```

## THE PROGRAM

In the program for this game, you met the following commands:

| | | |
|---|---|---|
| REM | INPUT | FOR-NEXT |
| DIM | LET | GOTO |
| PRINT | IF-THEN | END |

Let's look at them one at a time.

You already met the RUN command.

You type **RUN** to get the computer to play the game. When you type **RUN**, the computer obeys the commands in the program, one at a time.

The commands in the program, as you will note, are all on line numbers. Whenever you first type in commands, you place them on line numbers that increase each time by ten. (The commands appear on Lines 10, 20, 30, 40, and so on.) This will let you later insert new commands in between the old ones. For example,

if you have commands on Lines 20 and 30, you can later fit a new command in between them by labeling it Line 25.

The first command is REM, on Line 50:

```
50 REM ***GUESS THE NUMBER
```

Why is **REM** on Line 50? With **REM** on Line 50, you can add up to fifty new commands (0–49) at the beginning of your program. It's nice to have all of these unused line numbers, in case you need them.

**REM** stands for "remark." The computer ignores the REM command. Like a string tied to your finger, it is there to remind *you* what your program is or what it is supposed to do. For example, programmers frequently use the REM command at the beginning of a program to give the program a title.

The second command is the DIM command, on Lines 60 and 70:

```
60 DIM A$(1)
70 DIM NM$(15)
```

DIM is an "assignment" command and stands for "dimension." The computer's memory is filled with thousands of cubbyholes in which you can store commands and information. Before you can store any letters in a cubbyhole, however, you need to give the cubbyhole a name and tell the computer how much space you will need.

That's what the DIM command is for. The DIM commands above let you reserve two cubbyholes. A reserved cubbyhole is called a *variable*. One variable is named A$, and another is named NM$. A$ will hold the answers you give the computer. NM$ will hold your name.

The numbers in parentheses tell the computer how much space you are reserving. You want A$ to hold only one (1) letter. You want NM$ to be able to hold up to fifteen (15) letters.

Right now, A$ and NM$ are empty. You will fill them later.

Why is there a dollar sign ($), called a "string," in each of these names? Because these variables, or cubbyholes, are *string variables.* Strings are letters and punctuation marks, such as:

A L p Z r M # * !

Strings can also be numbers, but they must be numbers used for messages (like I AM 12 YEARS OLD.), not for arithmetic operations (such as 12 + 12 = 24). For example, you could dimension a string variable AG$ and store the number 12 in it:

```
DIM AG$
AG$="12"
```

How can you create cubbyholes for numbers used in arithmetic? To do that, you create a new kind of variable—the *numeric variable.* Numeric variables do not have a dollar sign in the name. Also, you do not need to dimension them to create them. You can create them in the assignment command. For example, the command AG=12 creates the numeric variable A and stores the number 12 in it. Now you can do your arithmetic.

Back to our game now. On Line 100, you see the command PRINT. You use the PRINT command to get the computer to display a message on the video screen. In the program the message is enclosed in double quotes (" "). The message on Line 100 of this program is "WHAT IS YOUR NAME."

There is a second command on Line 100: INPUT. You use the INPUT command to get the computer to accept information you will type in later. In this case, the information will be your name. The computer will store your name in the variable NM$.

You can paste together messages and commands by putting a colon (:) or a semicolon (;) between them. You use the colon to

paste *commands* together, for example, PRINT :PRINT (to get two empty lines on the TV screen). You use the semicolon to paste two *messages* (or strings) together. For example, if you type PRINT "HI "; "MARVIN", the computer will print the message HI MARVIN on the TV screen.

The semicolon and colon on Line 100 cause the computer to print an automatic question mark right after the question:

```
WHAT IS YOUR NAME?
```

The computer lets you type in your name directly following the question:

```
WHAT IS YOUR NAME? MARVIN
```

If you had not pasted the commands together, they would have looked like this:

```
100 PRINT "WHAT IS YOUR NAME"
105 INPUT NM$
```

When the program ran, it would have looked like this:

```
WHAT IS YOUR NAME
?MARVIN
```

Though this is not wrong, it does look a little odd, so we did it the other way.

## IS IT GOBBLEDYGOOK?

The next command, which appears on Line 110, is a little like a can of concentrated frozen orange juice or a box of laundry detergent. It packs a lot of programming power into only one line.

The chief command on Line 110 is LET. The LET command "lets" you make a copy of some piece of information to put into a numeric computer variable (cubbyhole) called N. Remember, you don't have to reserve space for N since it is a numeric variable.

But what information are you copying? To us it looks like gobbledygook:

```
INT(RND(1)*20)+1
```

What do the various letters and symbols in this line mean to the computer?

In this program, you have set upper and lower limits for the random number (RND) the computer selects. The upper limit is 20. The lower limit is 1. In this command, the computer is told to choose only numbers between 1 and 20.

If you had wanted the computer to think of a number between 10 and 1,000, you would have typed:

```
INT(RND(1)*991)+10
```

To get the number 991 in the command, you take the highest number that you want the computer to think of, 1000. Then, from that number, you subtract the lowest number, 10. This gives you 990. Then you add 1 and get 991.

Why do all of this juggling just to get a number between 10 and 1,000? Here is the explanation: The function RND(1) makes the computer think of a number greater than or equal to 0 and less than 1. We multiply that number by 991 to get a number greater than or equal to 0 and less than 991. The INT function changes this number into a whole number by dropping all numbers to the right of the decimal point. After that, we have a whole number somewhere between 0 and 990. We add 10 to get a number between 10 and 1,000: i.e., 0+10=10 or 990+10=1,000.

This may seem a complicated way to arrive at a number to you, but to a computer it is perfectly logical! And now, as an

exercise, try to figure out how the command would look if you wanted the computer to think of numbers between 50 and 500.[1]

Let's turn our attention now to the little symbols in this command.

In the BASIC language, whenever you want to multiply two numbers, you use the asterisk (*). For example, if you typed

```
PRINT 5*5
```

the computer would take the 5, multiply it by 5, and print the answer: 25.

In BASIC, whenever you want to add two numbers, you use the plus sign (+). For example, if you typed

```
PRINT 5+5
```

the computer would take the 5, add 5 to it, and print the answer: 10.

In BASIC, parentheses ( ) are used in three different ways. First, they act as "windows" into strings and arrays (variables that store a list of numbers). For example, if you created a string with the name "MARVIN," you could peek through the window into the string and see just the letter V with the commands:

```
10 DIM NM$(6)
20 NM$="MARVIN"
30 PRINT NM$(4,4)
```

After you entered these commands into the computer and typed **RUN**, the computer would print a V on the TV screen. On Line 30, you asked it to look at *only* the fourth letter in the string. (The two 4's mean: "Look at part of the string beginning with the

[1]The command would look like this: `INT(RND(1)*451)+50`.

fourth letter and ending with the fourth letter.") The fourth letter was a V.

The second way to use parentheses is to tell the computer what arithmetic to do first, second, third, fourth, and so on. For example, if you wanted the computer to add two numbers, then divide the sum, then multiply the quotient with another number, you would use parentheses. It might look like this:

```
PRINT ((12+12)/2)*5
```

A computer solves a problem like this from the inside out. That is, it digs down to the deepest level of parentheses and works its way back out, doing the arithmetic as it goes.

There are two sets of parentheses surrounding the addition problem (12+12), so it does that operation first, and it gets 24. There is only one set of parentheses around the division problem (24/2), so it does that next, and it gets 12. And there are no parentheses around the multiplication problem (12*5), so it does that last, and it gets 60.

The third way to use parentheses is to "feed" numbers to *functions*. Functions are like invisible helper programs. They are written in the computer's machine language of 1's (ones) and 0's (zeros). You can't look at a function on the inside, but you can feed it numbers and put it to work.

The RND function calculates random numbers. To make the function work, you must feed it either a 1 or a 0 enclosed in parentheses: **RND(1)** or **RND(0)**.

## BUILDING A RANDOM-NUMBER COMMAND

In our random-number command, a 1 was fed to the RND function: **RND(1)**. But the RND function by itself will always pick a number that is less than 1 and greater than or equal to 0. For example, it will pick numbers such as:

```
0.40321   0.00112   0.99179   0.0
```

In your program, you want a whole number between 1 and 20. You get this by including things along with the RND function. First, you need to multiply the random number by 20: **RND(1)*20**. But just multiplying the random number by 20 will give you numbers between 0 and 19, including decimal numbers such as 12.50436 and 15.66122!

So the next step is to use another function, the INT function. The INT function creates integers, or whole numbers. Whole numbers don't have fractions (like 5/6) or decimal places (like 4.03798). They are "counting" numbers, like 1, 2, 3, and 4.

When you use the INT function to create a random number that is a whole number, it looks like this:

```
INT(RND(1)*20)
```

This will give you a whole number between 0 and 19. But, in this program, you need a number between 1 and 20. The way to do that is *add 1*. The final random-number command looks like this:

```
INT(RND(1)*20)+1
```

## MATCHING NUMBERS

The next new command is on Line 150. This is the IF-THEN command. You use the IF-THEN command when you want the computer to test something, then make a decision based on what it discovers.

Line 150 looks like this:

```
150 IF G<>N THEN 300
```

On Line 150, you are making the computer check to see if your

guess (stored in G) is equal to the number the computer picked (N). The <> symbols mean "is not equal to." If the two numbers are not equal, then the computer jumps to a new line number—Line 300.

On Line 300 is a PRINT command without a message. This tells the computer to print a blank line.

On Line 310 is another IF-THEN command. IF your guess (G) is greater than (>) the computer's number (N) THEN the computer jumps to Line 350. On Line 350, the computer is told to print out the message **SORRY, MARVIN. TOO BIG!**

The next line (360) tells the computer to **GOTO 138**. This means it should jump backward ("go to") Line 138 and pick up the action there.

Before we examine this, let's do some backpedaling of our own.

Take another look at Line 310. What happens if your guess (G) is less than (<) the computer's number (N)? Then the computer ignores the jump command **(THEN 350)** and moves on instead to Line 320. It prints out the message **(SORRY, MARVIN. TOO LITTLE!)** and then goes to Line 330. Line 330 tells the computer to go to Line 138.

Let's backpedal some more. Take another look at Line 150. What happens if your guess (G) *equals* the computer's number (N)? This means that you got the right answer! The computer ignores the jump command **(THEN 300)** and instead goes to the command on the next line.

The computer prints a blank line, as instructed. This makes it easier for you to read the next message on the screen, which is **HURRAY, MARVIN! YOU GUESSED MY NUMBER.**

## THE COMPUTER RACETRACK

If you guess the right number, the computer counts to 300. This slows down the action for a moment and lets you savor your victory.

You get the computer to count to 300 with a FOR-NEXT command. The FOR-NEXT command puts the computer into a *loop.* A loop is like a racetrack. The computer runs round and round the "racetrack" doing only those commands sandwiched in between the FOR (the starting line) and the NEXT (the finish line).

On Line 190, what commands are there between the FOR and the NEXT?

None. So all the computer does is run around the track three hundred times.

How do you get the computer to do a full three hundred laps? You put in a "lap counter" as part of the FOR command:

```
FOR T=1 TO 300
```

The lap counter, T, is increased by one (1) each time the computer does a lap. It starts with the number 1 and ends at the number 300—after the computer has finished its three hundredth lap.

### DO YOU WANT TO PLAY AGAIN?

Look at Lines 210 and 220. The computer asks you if you want to play again.

The computer accepts your answer and stores it in A$. Since A$ is only big enough to store a single letter, the computer stores only the first letter of your answer. If you answered **YES, I SURE WOULD!** the only letter stored in A$ would be a Y. Or if you answered **NO WAY! I'M FINISHED,** the only letter stored in A$ would be the N from NO.

On Lines 230 and 240, the computer tests your answer. If you answered yes, the computer will find a Y in A$ and hop back to Line 110, select a new random number, and begin the game again.

If you answered no, the computer will ignore the jump command **(THEN 200)** on Line 240 and go to the command on Line

250. Line 250 has two commands: **PRINT** and **END**. The PRINT command tells the computer to print one last blank line. This is to separate the last message in the program from the computer's READY message. The computer prints **READY** when it is finished running your program. The END command tells the computer to stop running the program.

But what if you didn't answer either yes or no? What if you typed **MAYBE** or **PHOOEY!**? That would be an error, since you were supposed to answer yes or no, and Line 240 is ready for you. It bounces you back to Line 200, and the computer asks you the same question again. And it will keep asking this question until you get it right (or pull out the plug).

## BELLS AND WHISTLES

Now type in the program, and when you're sure you've typed it in *exactly* as shown, type in **RUN**. Try the game out first on yourself, then on your family and friends.

What sorts of bells and whistles can you add to this program? One thing you can do is to change the upper and lower limits of the number the computer guesses. Right now, the upper limit is 20, and the lower limit is 1. Look at Line 110. How would you change it to make the computer think of a number between 1 and 100? HINT: To change the RND command, look back at the examples given on pages 11 and 12.

Whenever you wish to change a command in a program already written, type the line number of the command, and then press the ENTER or RETURN key. This erases that line from your program. Next, type in the same line number (in this case, Line 110) plus the command *with your changes.*

# CHAPTER THREE

## THE "BUBBLE BRAIN" JOKE

Do you know a good joke, puzzle, or riddle? Then teach it to your computer!

A computer can deliver a joke with a straight face and perfect timing. And once it learns the joke, it will never forget it.

In the next four chapters, we'll teach the computer a variety of jokes and riddles. Then you can teach it some of your own. After a while, your computer will be quite a funny guy (or girl)! And then, if you need a good joke for a party or for the holidays, you can just ask your computer.

### EMPTY, EMPTY, EMPTY!

This is a trick joke. Its success depends on the person recognizing that the letters MT sound like the word "empty."

Here's how the program looks when you type **RUN**:

```
Computer: PUT YOUR FINGER TO YOUR HEAD.

          HOLD IT THERE.

          USE YOUR OTHER HAND TO ANSWER THIS QUES-
          TION:

          WHAT IS THE ABBREVIATION
          FOR MOUNTAIN?
```

You: MT
Computer: SAY THE LETTERS OUT LOUD
THREE TIMES.
You: MT . . . MT . . . MT
(Meanwhile, you are pointing to your head.)
Computer: HA! HA! HA!

YOU'RE RIGHT. YOUR HEAD IS EMPTY!!

**THE "BUBBLE BRAIN" PROGRAM**

```
 50  REM ***THE BUBBLE BRAIN JOKE
 60  DIM A$(2)
 90  PRINT
100  PRINT "PUT YOUR FINGER TO YOUR HEAD."
105  PRINT
107  FOR T=1 TO 1000:NEXT T
108  PRINT "HOLD IT THERE."
110  PRINT
111  FOR T=1 TO 1000:NEXT T
112  PRINT "USE YOUR OTHER HAND TO ANSWER"
113  PRINT "THIS QUESTION:"
115  PRINT
117  FOR T=1 TO 1000:NEXT T
120  PRINT "WHAT IS THE ABBREVIATION"
130  PRINT "FOR MOUNTAIN";:INPUT A$
140  IF A$="MT" THEN 200
150  PRINT
160  PRINT "NO  . . .  NO  . . .  NO  . . . "
170  PRINT
172  FOR T=1 TO 1000:NEXT T
175  PRINT "THE ABBREVIATION FOR MOUNTAIN"
180  PRINT "IS 'MT'."
185  PRINT
```

```
187 FOR T=1 TO 1000:NEXT T
190 PRINT "TRY AGAIN."
193 PRINT
194 FOR T=1 TO 1000:NEXT T
195 GOTO 100
200 PRINT
201 FOR T=1 TO 1000:NEXT T
202 PRINT "SAY THE LETTERS OUT LOUD"
203 PRINT "THREE TIMES."
205 PRINT
208 FOR T=1 TO 2000:NEXT T
210 PRINT "HA! HA! HA!"
215 PRINT
220 PRINT "YOU'RE RIGHT. YOUR HEAD IS EMPTY!!"
225 PRINT
240 PRINT
250 END
```

Note that on Lines 202 and 203, the computer tells the person to say the letters MT *out loud* three times. This is what makes the joke work. The person might not recognize the connection between "empty" and MT unless he or she says the letters MT out loud.

Whenever you teach your computer a joke, check to see if it is a "silent joke" or a "talkie." Silent jokes are funny even when they're just written out and not spoken. Talkies must be said aloud, or they won't work.

Also, look at all the FOR-NEXT delay loops we put in (FOR T=1 TO 1000:NEXT T). This gives the person lots of time to read each of the printed messages. Without the delay loops, the computer would go too fast. Remember, the key to a good joke is *timing*. You have to keep tinkering with the FOR-NEXT loops to polish the computer's timing. Don't give up until you get it just right.

Last, look at Lines 140 to 195. On Line 140, the computer

checks to see if you answered its question with MT. If not, it gives you the right answer. Then it lets you try again.

## MORE JOKES

Here are three more jokes you might use in a program:

```
100 PRINT "HOW DOES A GHOST EAT";:INPUT A$
200 PRINT "BY GOBLIN."
```

```
100 PRINT "NAME FOUR THINGS THAT CONTAIN MILK";:IN-
    PUT A$
200 PRINT "BUTTER, CHEESE, AND TWO COWS."
```

```
100 PRINT "WHAT DID PAUL REVERE SAY WHEN HE"
110 PRINT "COMPLETED HIS FAMOUS RIDE";:INPUT A$
200 PRINT "WHOA!"
```

How do you create these joke programs?

First, enter a REM command on a new Line 50. Give the new program a name.

Second, type in a DIM command on Line 60, to name and reserve space for the answer variable (cubbyhole) A$.

Third, type in one of the jokes.

Fourth, between the question (on Line 100) and the answer (on Line 200), you might want to print a message to respond to the person's answer—something like:

```
150 PRINT "SORRY! WRONG ANSWER!"
```

The computer assumes that the person didn't get the right answer. Then have the computer wait a couple of seconds:

```
170 FOR T=1 TO 1000:NEXT T
```

The computer will then tell the punch line of the joke.

# CHAPTER FOUR

## THE BLACK & BLUE JOKE

This next joke tests whether you are awake or not. If you are, you can outwit the computer.

After you type the joke into the computer, try it out yourself a few times. Then turn the computer loose on your family and friends.

Here's what the joke looks like when you type **RUN**:

```
Computer: I BET I CAN MAKE YOU SAY 'BLACK.'

          WHAT COLOR IS THE AMERICAN FLAG?
     You: RED, WHITE, AND BLUE.
Computer: HA! HA! HA!

          DIDN'T I TELL YOU THAT I COULD MAKE YOU SAY
          BLUE?
     You: NO!
Computer: WELL, WHAT COLOR THEN?
     You: YOU SAID BLACK.
Computer: HA! HA! HA!

          I TOLD YOU I COULD MAKE YOU SAY BLACK!
```

Here is what the program looks like:

```
50  REM *** THE BLACK & BLUE JOKE
60  DIM A$(40)
90  PRINT
100 PRINT "I BET I CAN MAKE YOU SAY 'BLACK.' "
110 FOR T=1 TO 1000:NEXT T
120 PRINT
130 PRINT "WHAT COLOR IS THE AMERICAN FLAG";:IN-
    PUT A$
140 PRINT
150 PRINT "HA! HA! HA!"
160 PRINT
170 PRINT "DIDN'T I TELL YOU THAT I COULD"
180 PRINT "MAKE YOU SAY BLUE";:INPUT A$
185 IF LEN(A$)<5 THEN 220
190 FOR I=1 TO LEN(A$)-4
200 IF A$(I,I)="B" AND A$(I+1,I+1)="L" AND
    A$(I+2,I+2)="A" AND A$(I+3,I+3)="C" AND
    A$(I+4,I+4)="K" THEN 250
210 NEXT I
220 PRINT
222 COUNT=COUNT+1
223 IF COUNT>1 THEN 230
224 PRINT "WELL, WHAT COLOR THEN";:INPUT A$
226 GOTO 185
230 PRINT
238 PRINT "YOU WIN! I GOOFED!"
240 GOTO 295
250 PRINT
260 PRINT "HA! HA! HA!"
270 PRINT
280 PRINT "I TOLD YOU I COULD MAKE YOU"
290 PRINT "SAY BLACK!"
295 PRINT
297 END
```

This joke teaches a lot of new programming tricks.

First, look at Line 60. We reserve forty spaces for the variable A$—DIM A$(40)—because most computers let you type up to forty letters on a single line. That should be enough space for all your answers.

Second, look at Lines 185 to 210. This is where we test to see if the person answered **BLACK** and fell into the computer's trap.

On Line 185, we use the LEN function. This function computes the length of the answer you've entered into A$. To see why we use the LEN function, let's look further back in the program for a moment.

First, on Line 130, the computer asks the color of the American flag, and you answer.

Does the computer look at this answer? No, it ignores your answer. It assumes that you answered **RED, WHITE, AND BLUE.**

On Lines 150 to 180, the computer tries to trick you. It laughs at you, then asks, **DIDN'T I TELL YOU THAT I COULD MAKE YOU SAY BLUE?**

But of course it didn't tell you this. It really said it could make you say BLACK. It "hopes" you will fall into its trap and answer its question on Lines 170 and 180 with the word BLACK.

On Lines 185 to 210, the computer tests your answer to see if it might be BLACK. It uses the LEN function, on Line 185, to see if your answer to its question on Lines 170 and 180 has five letters. If it does, that means you may have answered BLACK, since the word BLACK has five letters.

But if your answer doesn't have five letters, you probably answered something else, like **NO** or **NOPE** or **NO WAY, JOSE!**Your answer probably wasn't BLACK, so it was not what the computer wanted. You didn't fall into the trap.

What does the computer do next? It jumps to Line 220 and begins setting up a new trap.

Line 220 makes the computer print a blank line. Always have the computer print blank lines after each message on the TV screen. This makes the display nicer looking and easier to read. Otherwise, the messages look all jumbled up.

On Line 222, the computer meets a new variable, COUNT. Since this is the first time the computer has come to Line 222, it does three things. First, it creates a space for COUNT in its memory. Second, it gives COUNT an initial value of 0 (zero). Third, it obeys the command on Line 222 and adds the two numbers, COUNT+1, then stores their sum back in COUNT. COUNT started as 0, so the computer adds 0+1 and gets 1. Then it stores the 1 back in COUNT.

The value in COUNT is important. It tells the computer how many times it has tried to trick you. Right now, COUNT has a value of 1. That means the computer has tried to trick you one time.

As we shall see, the computer will return to Line 222 later in the program. And each time it returns, it will add 1 to COUNT.

On Line 223, the computer tests to see if COUNT is greater than (>) 1. If it is, you must have evaded the computer's trap twice. It wanted you to type the word BLACK, but twice you entered something else. You are too smart. The computer gives up. It jumps to 230 and prints: **YOU WIN! I GOOFED!** The program ends. You outwitted the computer.

But, if back on Line 223, the COUNT is equal to 1, then the computer tries a new trick. It goes to Line 224 and asks: **WELL, WHAT COLOR THEN?** It accepts your answer and then jumps back to Line 185. Line 185 tests to see if there are at least five letters in your answer. That way there is a chance that the word BLACK is somewhere in your answer.

If so, the computer goes to the FOR-NEXT loop on Lines 190 to 210. I is a *loop counter.* We'll come back to this in a second.

Look at what the loop does. On Line 200, the computer tests

to see if your answer has the word BLACK in it. Each time you see A$, you see parentheses with an I inside. The parentheses after A$ let you zoom in on just one letter stored in A$. It's like opening a window into A$. You only want to open the window wide enough to peek at one letter.

Let's say I=1. If we then replace all the I's with 1's and do the arithmetic, the command on Line 200 will look like this:

```
IF A$(1,1)="B" AND A$(2,2)="L" AND A$(3,3)="A" AND
    A$(4,4)="C" AND A$(5,5)="K" THEN 250
```

This means:

If the first letter in A$ =B and
the second letter in A$ = L and
the third letter in A$ = A and
the fourth letter in A$ = C and
the fifth letter in A$ = K

*then* jump to Line 250.

Since the loop counter, I, increases by 1 each time through the loop, the first time through the computer looks for the word BLACK beginning with the *first* letter in A$. The second time through the loop, the computer looks for the word BLACK beginning with the *second* letter in A$. The third time through, it looks at the *third* letter in A$, to see if it begins to spell the word BLACK, and so on.

The computer is very careful. It doesn't miss much. If the word BLACK is somewhere in A$, it will find it.

The computer keeps up its search until it gets to the last five letters in A$. You stop its search by the FOR command on Line 190. This command sets a top value of LEN(A$)−4 on the loop counter.

If, for example, your answer, A$, had twenty-eight (28) letters, LEN(A$)−4 would equal 28−4, or 24. The computer would keep circling through the loop until I equaled 24. Then the computer would check to see if the twenty-fourth letter was B, the twenty-fifth letter was L, the twenty-sixth letter was A, the twenty-seventh letter was C, and the twenty-eighth letter was K.

If the computer finds the word BLACK somewhere in your answer (A$), that means you fell into its trap. It jumps to Line 250, laughs at you, and tells you: I TOLD YOU I COULD MAKE YOU SAY BLACK. Then the program ends.

But what if you were too tricky and avoided saying the word BLACK?

Then the computer finishes the loop and moves on to Line 222. It adds 1 to COUNT. If you have outwitted the computer twice, COUNT will be equal to 2. The computer jumps to Line 230, admits defeat, and quits trying to trick you.

# CHAPTER FIVE

## KNOCK-KNOCK JOKES

Your computer can become an expert at knock-knock jokes. You can teach the computer your favorite jokes using the program in this chapter.

The computer begins telling jokes as soon as you type RUN:

```
Computer: KNOCK, KNOCK?
     You: WHO'S THERE?
Computer: ADOLF
     You: ADOLF WHO?
Computer: ADOLF BALL HIT ME.
          DAT'S WHY I TALK DIS WAY.

Computer: KNOCK, KNOCK?
     You: WHO'S THERE?
Computer: ROBIN
     You: ROBIN WHO?
Computer: ROBIN YOU!
          SO HAND OVER THE MONEY!
```

The program that follows contains ten knock-knock jokes, just like the ones above. If you have a strong stomach, you can run the program and listen to all ten.

The program looks like this:

```
50    REM *** KNOCK-KNOCK JOKES
60    DIM A$(30)
100   FOR I=1 TO 10
110   PRINT
120   PRINT "KNOCK, KNOCK";:INPUT A$
130   PRINT
140   READ A$
150   PRINT A$
155   PRINT
160   INPUT A$
165   PRINT
170   READ A$:PRINT A$
180   READ A$:PRINT A$
190   NEXT I
200   PRINT
210   PRINT "*** THAT'S ALL, FOLKS! ***"
220   PRINT
230   END
1000  DATA ADOLF,ADOLF BALL HIT ME.,DAT'S WHY I
      TALK DIS WAY.
1010  DATA DEBBIE,DEB-BIE STUNG,ME.
1020  DATA ROBIN,ROBIN YOU!,SO HAND OVER THE
      MONEY!
1030  DATA WENDY,WENDY TODAY,CLOUDY TOMOR-
      ROW.
1040  DATA HARRY,HARRY UP AND,OPEN THIS DOOR!
1050  DATA SARAH,SARAH DOCTOR,IN THE HOUSE?
1060  DATA CHESTER,CHESTER MINUTE AND,I'LL TRY
      TO FIND OUT.
1070  DATA MINNIE,MINNIE BRAVE HEARTS LIE,ASLEEP
      IN THE DEEP.
1080  DATA BOO,DON'T CRY.,IT'S ONLY A JOKE.
1090  DATA AMOS,AMOS-QUITO JUST,BIT ME!
```

This program teaches you an important new command: READ.

The READ command is similar to the INPUT command. When the computer sees the INPUT command, it prints a question mark (?) and waits for you to type something on its keyboard. Then it takes the information or message you type in and stores it in a variable (a memory cubbyhole).

When the computer sees a READ command, it looks for another command, the DATA command. Then it takes one unit of information and stores it in the variable A$. Usually, you put DATA commands at the end of the program, after all the other commands. DATA commands hold information. In this program, the information is knock-knock jokes.

You can add a new unit of information in two ways. First, you can write a new DATA command on a new line. Or, you can place a comma after the old unit of information, then write the new unit of information right after it. For example, you can get the computer to read the following two pieces of information this way:

```
10 READ A$:PRINT A$
20 READ A$:PRINT A$
30 DATA ADOLF BALL HIT ME.
40 DATA DAT'S WHY I TALK DIS WAY.
```

or this way:

```
10 READ A$:PRINT A$
20 READ A$:PRINT A$
30 DATA ADOLF BALL HIT ME., DAT'S WHY I TALK DIS
   WAY.
```

In either case, the computer reads and prints out only one line of a knock-knock joke at a time.

## MORE KNOCK-KNOCK JOKES

It's simple to teach the computer some new knock-knock jokes. Just add the jokes at the tail end of the program, beginning with Line 1100. Remember to begin each new line with a line number and the DATA command.

Also, you will need to change the FOR-NEXT loop so it will read your new jokes. Look at Line 100. The loop is set to 10, since the computer knows ten knock-knock jokes. When you teach the computer new jokes, count up how many there are, and add this number to the number 10. For example, if you add six new jokes, add 6 to 10. You get 16. Change the FOR command on Line 100 to: FOR I=1 TO 16. Then the computer will read all sixteen jokes.

Now here are six more knock-knock jokes you can use:

```
1100 DATA GORILLA,GORILLA MY DREAMS,I LOVE
     YOU!
1110 DATA CARMEN,CARMEN,GET IT!
1120 DATA SHARON,SHARON,SHARE ALIKE.
1130 DATA CELIA,CELIA,LATER.
1140 DATA NOAH,NOAH BODY KNOWS,THE TROU-
     BLE I'VE SEEN.
1150 DATA HURON,HURON TIME,FOR ONCE.
```

# CHAPTER SIX

# THE RIDDLES OF THE SPHINX

The Sphinx was a mythical Greek monster. It had the head of a woman, the paws and body of a lion, the wings of a bird, and the tail of a serpent. It also had a human voice and prowled around the Greek countryside just outside the city of Thebes. Whenever it encountered travelers, it seized them and asked them a riddle. When people couldn't answer the riddle, the Sphinx ate them.

One day, the Sphinx met up with a young man named Oedipus and gave him this riddle:

*What goes on four feet, on two feet, and three,*
*But the more feet it goes on the weaker it be?*

'The answer to your riddle," said Oedipus, "is Man. Man goes on four feet when he crawls as a baby. He goes on two feet as an adult. In old age, he is weak and needs a staff, so he goes on three feet."

Oedipus was right! In a fearful rage, the Sphinx killed itself. Oedipus became a hero and the king of Thebes.

## THE COMPUTER SPHINX

Your computer can become an expert at riddles, just like the Sphinx. But what happens if you miss one of the computer's riddles? Will it eat you?

Not likely.

The Computer Sphinx program looks like this when you type RUN:

```
Computer: HI! I'M THE SPHINX.
          I WILL TELL YOU SOME RIDDLES.
          YOU TRY TO GUESS THE ANSWERS.

          WHAT QUESTION CAN NEVER
          BE ANSWERED 'YES?'
     You: WILL YOU MARRY ME?
Computer: NOPE.
          THE QUESTION IS:
          ARE YOU ASLEEP?

Computer: WHAT DID ONE EYE SAY
          TO THE OTHER?
     You: YOU LOOK TERRIBLE.
Computer: NOPE.
          SOMETHING BETWEEN US SMELLS.
```

Here is the Computer Sphinx program:

```
 50   REM *** THE COMPUTER SPHINX
 60   DIM A$(40)
 70   PRINT
 80   PRINT "HI! I'M THE SPHINX."
 82   PRINT
 83   FOR T=1 TO 1000:NEXT T
 85   PRINT "I WILL TELL YOU SOME RIDDLES."
 90   PRINT
 92   FOR T=1 TO 1000:NEXT T
 95   PRINT "YOU TRY TO GUESS THE ANSWERS."
 97   PRINT
 98   FOR T=1 TO 1000:NEXT T
100   FOR I=1 TO 16
```

```
110  PRINT
120  READ A$:PRINT A$
130  READ A$:PRINT A$;:INPUT A$
140  PRINT
145  FOR T=1 TO 1000:NEXT T
147  PRINT "NOPE."
148  PRINT
150  READ A$:PRINT A$
160  READ A$:PRINT A$
170  FOR T=1 TO 1000:NEXT T
180  NEXT I
190  PRINT
200  PRINT " *** THAT'S ALL, FOLKS! ***"
210  PRINT
220  PRINT
230  END
1000 DATA WHAT QUESTION CAN NEVER, BE AN-
     SWERED 'YES'
1010 DATA THE QUESTION IS:, ARE YOU ASLEEP?
1020 DATA WHY IS A DOG BITING ITS TAIL,LIKE A
     GOOD MANAGER
1030 DATA BECAUSE IT MAKES, BOTH ENDS MEET.
1040 DATA THE WOODS ARE TWO MILES ACROSS.,
     HOW FAR CAN YOU GO INTO THEM
1050 DATA ONE MILE. AFTER THAT,YOU ARE GOING
     OUT.
1060 DATA WHAT DID ONE STRAWBERRY SAY,TO THE
     OTHER
1070 DATA IF YOU WEREN'T SO FRESH,WE WOULDN'T
     BE IN THIS JAM.
1080 DATA WHAT DID ONE EYE SAY,TO THE OTHER
1090 DATA SOMETHING BETWEEN US SMELLS!,
1100 DATA WHAT HAS EIGHTEEN LEGS,AND CATCHES
     FLIES
1110 DATA A BASEBALL TEAM,
```

```
1120 DATA WHAT SHOULD A MAN KNOW,BEFORE
     TRYING TO TEACH A DOG
1130 DATA MORE THAN THE DOG,
1140 DATA WHAT DO YOU GET IF YOU POUR,WARM
     WATER DOWN A RABBIT HOLE
1150 DATA HOT CROSS BUNNIES,
1160 DATA WHAT IS YELLOW AND,ALWAYS POINTS
     NORTH
1170 DATA A MAGNETIZED BANANA,
1180 DATA WHAT STARTS WITH T,ENDS WITH T AND IS
     FULL OF T
1190 DATA A TEAPOT,
1200 DATA WHERE WAS THE DECLARATION OF,INDE-
     PENDENCE SIGNED
1210 DATA AT THE BOTTOM,
1220 DATA WHAT SQUEALS LOUDER THAN,A PIG
     CAUGHT UNDER A FENCE
1230 DATA TWO PIGS,
1240 DATA WHAT EIGHT-LETTER WORD,HAS ALL THE
     LETTERS IN IT
1250 DATA ALPHABET,
1260 DATA WHAT WORD WILL ALL PEOPLE,PRO-
     NOUNCE WRONG
1270 DATA WRONG,
1280 DATA WHERE WAS HENRY VIII,CROWNED
1290 DATA ON HIS HEAD,
1300 DATA HOW MUCH DIRT IS IN A HOLE 8 FEET,
     LONG AND 8 FEET WIDE AND 8 FEET DEEP
1310 DATA NONE.,A HOLE IS EMPTY.
```

This program is really quite short. It only looks long because the Computer Sphinx knows sixteen riddles. The DATA statements with the riddles take up more than half the space.

Also, this program is simple. It is very similar to the Knock-Knock Joke program you just learned about.

## MORE RIDDLES

You can teach your computer many riddles. Just add new ones at the tail end of this program, beginning with Line 1320. Remember to begin each new riddle with a line number and a DATA command. Also, remember to add up all the riddles and change the FOR command on Line 100 from 16 to the new number of riddles.

For example, if you wanted to add six new jokes, you would change the FOR command on Line 100 to 100 FOR I=1 TO 22. And you would add the following DATA commands:

```
1320 DATA WHAT KIND OF SHOES,ARE MADE OUT OF
     BANANA SKINS
1330 DATA SLIPPERS,
1340 DATA WHAT IS A,TWIP
1350 DATA IT IS WHEN A WABBIT,WIDES ON A TWAIN
1360 DATA WHAT IS GREEN AND CRAWLY,AND HAS
     100 LEGS
1370 DATA A CENTIPICKLE,
1380 DATA WHAT GOES THROUGH A DOOR,BUT NEV-
     ER GOES IN OR OUT
1390 DATA A KEYHOLE,
1400 DATA WHAT IS BLACK AND WRINKLED,AND
     MAKES PIT STOPS
1410 DATA A RACING,PRUNE
1420 DATA WHAT IS GRAY INSIDE,AND CLEAR OUT-
     SIDE
1430 DATA AN ELEPHANT,IN A BAGGIE
```

Type in these new commands, and your computer will know twenty-two riddles. Then teach the computer some favorite riddles of your own.

# CHAPTER SEVEN

## THE FORTUNE-TELLER

It is the middle of the night. Outside, the wind is howling. Tree branches, like spidery fingers, are scratching against your bedroom window.

You sit up in your bed. You just heard a noise—a strange, eerie noise.

You get out of bed and creep silently out of your room. The noises are coming from the living room. You walk down the stairs slowly, so the steps won't creak and wake everyone up.

You peek into the living room. That's where you keep your computer. All the lights are off. The room is dark, except for the computer's video screen. It is blazing.

The strange noises have stopped. But why is the computer on? Did you forget to turn it off? You creep over and look closely at the screen.

On a hunch, you type **RUN**. This is what you see:

```
HELLO! I AM GRZYKIEL.
I LIVE INSIDE THIS COMPUTER.
```

You gasp. Is this for real? You pinch yourself to see if you are dreaming. The pinch hurts!

You sink into the chair in front of the computer and stare at the screen. Grzykiel keeps talking:

I AM 4000 YEARS OLD.
I HAVE DISCOVERED THE SECRET OF FORTUNE-TELLING.
I CAN SEE THE FUTURE.
WHAT IS YOUR NAME?

Should you tell Grzykiel your name? Your hand is shaking.

You can see the computer's keyboard in the flickering light cast by the screen. Slowly you type in your name: **MARVIN**. You push the RETURN button.

Grzykiel answers:

MARVIN, DO YOU WANT ME TO TELL
YOUR FORTUNE?

You're scared silly. You really ought to wake your parents and show them this.

But you're fascinated, too. Can Grzykiel really tell your fortune? Why not give him—or it—a chance. You type yes, then press the RETURN button. Grzykiel quickly answers:

ASK ME ANY QUESTION.
(IT MUST HAVE A YES OR NO ANSWER.)
WHAT DO YOU WANT TO KNOW?

You think for a moment. What do you most want to know? Ah, ha! You have it. You type:

WILL I EVER BREAK A HUNDRED
THOUSAND ON THE *DINOSAUR*
*WARS* GAME AT THE ARCADE?

You hit RETURN and laugh a little—nervously. Grzykiel storms back:

I MUST HAVE ABSOLUTE SILENCE!

You choke on your laughter. Gryzkiel continues:

I AM GAZING AT MY CRYSTAL BALL
GAZING . . .
GAZING . . .
GAZING . . .

You get impatient. You press your nose against the screen. "C'mon," you whisper.

Gryzkiel must have heard you. He types:

I SEE YOUR FUTURE!
YOUR QUESTION WAS:
WILL I EVER BREAK A HUNDRED
THOUSAND ON THE *DINOSAUR*
*WARS* GAME AT THE ARCADE?
THE ANSWER IS: *** YES! ***

Gryzkiel continues typing. He says he's ready to gaze into his crystal ball again and uncover a little more of your future.

But you aren't there to see this part. You're on your way upstairs to wake up your whole family to tell them the good news.

## THE PROGRAM

```
50  REM *** FORTUNE TELLER
60  DIM NM$(30)
70  DIM A$(1)
```

```
80  DIM F$(80)
100 PRINT :PRINT :PRINT
110 PRINT "HELLO! I AM GRZYKIEL!"
120 PRINT
125 FOR T=1 TO 1000:NEXT T
126 PRINT "I LIVE INSIDE THIS COMPUTER."
127 PRINT
128 FOR T=1 TO 1000:NEXT T
130 PRINT "I AM 4000 YEARS OLD."
134 PRINT
138 FOR T=1 TO 1000:NEXT T
140 PRINT "I HAVE DISCOVERED THE SECRET"
150 PRINT "OF FORTUNE TELLING."
155 PRINT
160 FOR T=1 TO 1000:NEXT T
170 PRINT "I CAN SEE THE FUTURE!"
175 PRINT
176 FOR T=1 TO 1000:NEXT T
180 PRINT "WHAT IS YOUR NAME";:INPUT NM$
190 PRINT
200 PRINT NM$;", DO YOU WANT ME TO TELL"
210 PRINT "YOUR FORTUNE";:INPUT A$
220 PRINT
230 IF A$="N" THEN 530
240 IF A$<>"Y" THEN 200
250 PRINT "ASK ME ANY QUESTION."
260 PRINT "(IT MUST HAVE A YES OR NO ANSWER.)"
270 PRINT
275 FOR T=1 TO 800:NEXT T
280 PRINT "WHAT DO YOU WANT TO KNOW";:INPUT F$
290 PRINT
300 PRINT "I MUST HAVE ABSOLUTE SILENCE!"
305 PRINT
306 FOR T=1 TO 800:NEXT T
```

```
310 PRINT "I AM GAZING AT MY CRYSTAL BALL."
320 FOR I=1 TO 3
330 PRINT
340 FOR T=1 TO 800:NEXT T
350 PRINT "        GAZING  . . . "
360 NEXT I
370 PRINT
375 FOR T=1 TO 1000:NEXT T
380 PRINT "I SEE YOUR FUTURE!"
390 PRINT
395 FOR T=1 TO 1000:NEXT T
400 PRINT "YOUR QUESTION WAS:"
410 PRINT
420 PRINT F$
430 FOR T=1 TO 1000:NEXT T
440 PRINT
445 PRINT "THE ANSWER IS:";
450 IF INT(RND(1)*2)+1=1 THEN 480
455 FOR T=1 TO 1000:NEXT T
460 PRINT " *** YES!! ***"
470 GOTO 500
480 FOR T=1 TO 1000:NEXT T
485 PRINT " *** NO! ***"
500 PRINT
510 FOR T=1 TO 1000:NEXT T
520 GOTO 190
530 PRINT :PRINT :PRINT
540 END
```

The Gryzkiel program is long, but it's incredibly simple. It is mostly a bunch of PRINT commands. The PRINT commands display Gryzkiel's messages on the TV screen. Between the PRINT commands are FOR-NEXT loops, which make the computer wait between messages.

The heart of the program is the RND (random number) function on Line 450. The function picks a number by chance—either a 1 or a 2. If it picks a 2, Gryzkiel answers your question with a yes. If it picks a 1, Gryzkiel answers no. It's as if the fortunes were picked out of a hat. They are completely unrelated to the questions.

## IT WAS A DARK AND STORMY NIGHT

Is there really a wizard living inside your computer? No.

Does the Gryzkiel program really wake you up on a dark and stormy night and make strange, eerie noises? No.

The little story at the beginning of this chapter was just that—a story. It wasn't part of your program.

Then why was it there? It was there to show you the power of the imagination. The Grzykiel program doesn't do very fancy things. But if you get people to use their imaginations before you let them use it, you will have them believing there really is something eerie going on inside your computer.

Or at least they'll have fun pretending.

Don't tell people that the program just tosses a coin and comes up with their future. Instead, surround it with magic. Tell people you have captured a four-thousand-year-old wizard. The wizard can tell the future. Perhaps you can heighten the illusion by adding visual effects, like bright flashes on the video screen, and sound effects, like blowing wind and explosions.

Remember, your programs can be extremely simple but still have a powerful effect on another person. All you have to do is trigger that person's imagination.

# CHAPTER EIGHT

# DRAGON WORLD

You are a space vagabond. You make a living scavenging things on remote planets, then selling them at interstellar flea markets. At a bar one night you hear about Dragon World, a planet in a faraway star system.

The planet is controlled by dragons. The dragons have the ability to teleport. They can go anywhere, just by thinking it. They use their special power to teleport to rich planets in the star system and rob them of their jewels and precious metals.

Dragon World is honeycombed with underground mazes, caves, and secret passages. It is in this netherworld that the dragons live, surrounded by their stolen treasure.

You fly your tiny, beat-up spaceship to Dragon World and leave it hidden in the swamp. You then begin your trek through the jungle searching for dragon caves.

You come into a clearing. At the far end of the clearing are two enormous caves. You are ready to plunge into the caves. But which one should you enter?

## THE GAME

You type **RUN**, and the game begins:

```
Computer: WHAT IS YOUR NAME?
     You: MARVIN
Computer: WELCOME TO DRAGON WORLD, MARVIN!

          YOU ARE STANDING IN FRONT OF TWO DRAGONS'
          CAVES. IN EACH CAVE IS A FABULOUS TREASURE.

          ONE OF THE DRAGONS IS FRIENDLY. IT WILL BE
          HAPPY TO SHARE ITS TREASURE.

          THE OTHER DRAGON IS HUNGRY AND EVIL. WHEN
          YOU ENTER THE CAVE, IT WILL EAT YOU.

          THE TWO DRAGONS LIKE TO TRADE CAVES. YOU
          NEVER KNOW WHICH CAVE THE MEAN DRAGON IS
          IN.

          WHICH CAVE DO YOU CHOOSE (1 OR 2)?
     You: 1
Computer: YOU ENTER CAVE #1.

          THE CAVE IS DARK AND SMELLY.

          A HUGE, UGLY DRAGON LEAPS OUT AT YOU AND . . .
     You: AND WHAT? AND WHAT?
          [You are sweating. You grip the computer in fear.]
Computer: AND GIVES YOU ITS TREASURE.
          [You collapse on the floor under the computer.]
          WANT TO TRY AGAIN?
```

## THE PROGRAM

You have learned all the commands you need to create the Dragon World game.

The Dragon World program looks like this:

```
50  REM *** DRAGON WORLD
60  DIM NM$(15)
```

```
70  DIM A$(1)
100 PRINT
110 PRINT "WHAT IS YOUR NAME";:INPUT NM$
120 PRINT
125 PRINT
130 PRINT "WELCOME TO DRAGON WORLD, "
    ;NM$;"!"
140 PRINT
150 PRINT
155 FOR T=1 TO 1000:NEXT T
160 PRINT "YOU ARE STANDING IN FRONT OF TWO"
170 PRINT "DRAGONS' CAVES. IN EACH CAVE"
180 PRINT "THERE IS A FABULOUS TREASURE."
190 PRINT
195 FOR T=1 TO 1000:NEXT T
200 PRINT "ONE OF THE DRAGONS IS FRIENDLY."
210 PRINT "IT WILL BE HAPPY TO SHARE ITS"
220 PRINT "TREASURE."
230 PRINT
240 FOR T=1 TO 1000:NEXT T
250 PRINT "THE OTHER DRAGON IS HUNGRY AND"
260 PRINT "EVIL. WHEN YOU ENTER THE CAVE,"
270 PRINT "IT WILL EAT YOU."
280 PRINT
290 FOR T=1 TO 1000:NEXT T
300 PRINT "THE TWO DRAGONS LIKE TO TRADE
    CAVES."
310 PRINT "YOU NEVER KNOW WHICH CAVE THE"
320 PRINT "MEAN DRAGON IS IN."
330 PRINT
340 FOR T=1 TO 1000:NEXT T
350 PRINT "WHICH CAVE DO YOU CHOOSE (1 OR
    2)";:INPUT A
360 PRINT
```

```
365 IF A<>1 AND A<>2 THEN 350
370 FOR T=1 TO 1000:NEXT T
380 PRINT "YOU ENTER CAVE # ";A;"."
390 PRINT "THE CAVE IS DARK AND SMELLY."
400 PRINT "A HUGE, UGLY DRAGON LEAPS OUT AT"
410 PRINT "YOU AND . . . "
420 PRINT
430 FOR T=1 TO 1000:NEXT T
440 IF (INT(RND(1)*2)+1)=A THEN 500
460 PRINT "GIVES YOU ITS TREASURE!"
470 GOTO 510
500 PRINT "EATS YOU IN ONE BIG BITE!"
510 PRINT
520 FOR T=1 TO 1000:NEXT T
530 PRINT "WANT TO TRY AGAIN, ";NM$;:INPUT A$
540 PRINT
550 IF A$="Y" THEN 350
560 IF A$<>"N" THEN 510
570 PRINT :END
```

Line 50 is the program title. On Lines 60 and 70, you reserve space in the computer's memory for two variables—NM$ for your name and A$ for answers.

As you can see, most of the commands in Dragon World are PRINT commands. After each PRINT message from the computer, there is an empty PRINT command followed by a FOR-NEXT loop. (See, for example, Lines 150 and 155.) This is how you make the computer print a blank line and pause while it counts to a thousand. The pause gives you time to read the computer's messages.

Now look at Line 365. Line 365 is an *error trap.* It tests to make sure you have entered the right answer—a 1 for cave #1 or a 2 for cave #2—into the computer. If you didn't give the right answer, the computer bounces back to Line 350 and prints the

question WHICH CAVE DO YOU CHOOSE (1 OR 2)? again. It keeps printing the question until you answer with a 1 or a 2.

Remember the RND function? You use it to make the computer think up a random number. We use the RND function on Line 440 as part of an IF-THEN command.

Let's say that you chose cave #2. That means the computer stored the number 2 in the variable named A (see Line 350).

Let's also say that, on Line 440, the computer obeys the RND function and selects the number 2.

Next the computer puts these values into the command on Line 440:

```
IF 2=2 THEN 500
```

Since 2 does equal 2, the computer jumps to Line 500.

Uh-oh. Too bad. You picked the cave with the mean dragon. It eats you in one bite!

But what would have happened if you had selected cave #1? Then Line 440 would have looked like this:

```
IF 2=1 THEN 500
```

Since 2 does not equal 1, the computer ignores the jump command (THEN 500). Instead, it goes to the command on the next line.

Hurray! You picked the cave with the friendly dragon. It gives you its treasure.

## FIVE EVIL DRAGONS

Dragon World is a very short program. It has only one villain—an evil dragon. You can make the program much longer and much more exciting. For example, you can change the two dragon caves into six. A dragon would live inside each cave, but only

one dragon would be friendly. The other five would be mean and evil. You can make each of the dragons different. The hero would have to battle or outwit each dragon differently.

Think of each dragon battle as a scene in a movie. You can create these scenes on different lines in your program, then make the computer jump to them. Let's examine the following series of commands:

```
440 IF (INT(RND(1)*6)+1)<>A THEN 455
445 PRINT "GIVES YOU ITS TREASURE!"
450 GOTO 460
455 ON INT(RND(1)*5)+1 GOSUB 500,600,
    700,800,900
460 PRINT
470 FOR T=1 TO 1000:NEXT T
480 PRINT "WANT TO TRY AGAIN, " ;NM$;:INPUT A$
485 PRINT
490 IF A$="Y" THEN 350
492 IF A$<>"N" THEN 460
495 PRINT :END
500 REM *** MEAN DRAGON #1
                          . . .

590 RETURN
600 REM *** MEAN DRAGON #2
                          . . .

690 RETURN
```

In this program, Line 440 checks to see if the number it has picked is equal to the number you picked. If it is, that means you picked the friendly dragon, and you get the treasure.

However, if the numbers are not equal, it means you picked

one of the evil dragons' caves. On Line 455, the computer finds out which cave:

```
455 ON INT(RND(1)*5)+1 GOSUB 500,600,700,800,900
```

The first thing the computer does on Line 455 is think of a number between 1 and 5: INT(RND(1)*5)+1. The number is the key to where the computer goes next—Line 500, Line 600, Line 700, Line 800, or Line 900.

If the number is 1, the computer jumps to the *first* line number—Line 500. From Lines 500 to 589, you can use PRINT commands to create the first evil dragon.

If the number is 2, the computer jumps to the *second* line number—Line 600. From Lines 600 to 689, you can use PRINT commands to create the second evil dragon.

And so on, all the way up to the number 5. Based on which number the computer picks, you send the player/hero to dragon #1 (Lines 500–590), dragon #2 (Lines 600–690), dragon #3 (Lines 700–790), dragon #4 (Lines 800–890), or dragon #5 (Lines 900–990). In each of those places you create a different dragon.

## THE BOOMERANG COMMAND

Look again at Line 455. Notice that the jump command is different. Instead of GOTO, it is GOSUB.

GOSUB is a "boomerang" command. GOTO simply makes the computer jump to another part of the program. Then the program proceeds normally.

GOSUB is different. It sends the computer to a line number, then waits for the computer to come back again.

What makes the computer come back? The RETURN command.

You can use the GOSUB command to divide big game programs into little "helper" programs. The helper programs are called *subroutines*. The GOSUB command tells the computer to obey one of these subroutines, then return when it's finished.

Programmers use subroutines to handle lots of details that the main program shouldn't be bothered with. Subroutines are like servants to a powerful king. The king orders the servants to take care of their various jobs, then report back when they are done.

Each subroutine here creates a different dragon. When you are done with a particular dragon, you just enter a line number and the RETURN command. This causes the computer to bounce back to the main program. The next command it obeys is the one immediately after the GOSUB command.

We will use subroutines in all the remaining game programs. They make writing such programs much easier. You can take a complicated game and divide it into lots of simple, manageable parts. Subroutines can be created to take care of each part. The main program controls all the subroutines and makes them work together, just like a stagecoach driver reining in a team of powerful horses.

# CHAPTER NINE

# SECRET AGENT

While I was growing up, I often dreamed I was a secret agent. I was always inventing secret codes and thinking up places to stash important stuff I didn't want anyone else to see. Sometimes I pretended I was Blackbeard the Pirate and buried my treasure in metal file-card boxes all around the backyard. I kept the directions to my buried boxes in my top right-hand desk drawer. They were safe from snoops because they were all in code—code only I could decipher.

Only problem was, sometimes I'd forget the code and not be able to figure it out either!

Even when I remembered how a code worked, it was still a hassle. I spent hours converting things written in English into my code, then hours more translating them back again.

What I needed was a secret code machine.

## BUILD A SECRET CODE MACHINE

Now, at long last, I have a secret code machine. I no longer have to worry about anyone snooping in my bedroom, of course. But maybe you do. Or maybe you would like to send secret messages to a friend with a computer. You can both program your computers for coding and decoding, then pass written messages back and forth.

Here's how it works. After you enter the program into your computer and type **RUN**, the screen shows:

```
I AM AN ELECTRONIC CODING MACHINE

I OFFER THREE KINDS OF CODES:

1. THE OB CODE
2. SCRAMBLED LETTERS
3. UPSIDE-DOWN AND BACKWARD

WHICH CODE (1, 2, OR 3)?
```

You type 1 (for the Ob Code). The coding machine answers:

```
ENTER THE MESSAGE YOU WANT CODED.
IT MUST BE NO MORE THAN 5 LINES.
```

The machine types **LINE** and the line number (#1, 2, 3, 4, or 5), and you enter the lines in your message. Remember to press the ENTER or RETURN key after you type in each line:

| The Computer Types | You Type |
|---|---|
| LINE #1 | THE KEYS TO MY ROBOT |
| LINE #2 | ARE BURIED BEHIND |
| LINE #3 | THE PINE TREE NEXT |
| LINE #4 | TO THE WOOD PILE |
| LINE #5 | UNDER A BIG ROCK. |

The machine then types:

```
WHIRRRR!! NOW CODING!!!
```

A few seconds later it types:

DO YOU WANT TO SEE YOUR ORIGINAL MESSAGE?

You can look at your message if you type **YES**.

After printing the original message, the machine checks to see if you want to see the coded message. If you type **YES**, the machine types:

THEB KEB YB S TOB MYB ROB BOB T
AB REB BUB RIB EB D BEB HIB ND
THEB PIB NEB TREB EB NEB XT
TOB THEB WOB OB D PIB LEB
UB NDEB R AB BIB G ROB CK.

That's what your message looks like in the Ob Code.

You can get the machine to code the message in the two other codes, too. When it asks if you want to see a new code, you type the numbers of the other two codes.

The machine whirrs again, then prints the message in the Scrambled Letters Code:

UIF!LFZT!UP!NZ!SPCPU
BSF!CVSJFE!CF!JOE
UIF!QJOF!USFF!OFYU
UP!UIF!XPPE!QJMF
VOEFS!B!CJH!SPDL/

The machine does some more quick fiddling with your message and comes up with the Upside-Down and Backward Code:

.KCOR GIB A REDNU
ELIP DOOW EHT OT
TXEN EERT ENIP EHT
DNIHEB DEIRUB ERA
TOBOR YM OT SYEK EHT

## THE PROGRAM

The Secret Agent coding machine is the longest program so far. Here is what the main program looks like:

```
 50  REM *** SECRET AGENT
 55  DIM A$(1)
 60  DIM L1$(36)
 65  DIM L2$(36)
 70  DIM M1$(180)
 80  DIM M2$(180)
 90  FOR I=0 TO 9
 92  M1$(I*18+1,I*18+18)="                  "
 93  M2$(I*18+1,I*18+18)="                  "
 94  NEXT I
 95  C1=-1
100  PRINT
110  PRINT "I AM AN ELECTRONIC CODING MACHINE."
120  PRINT
123  PRINT "I OFFER THREE TYPES OF CODES:"
125  PRINT
127  PRINT "1. THE OB CODE"
129  PRINT "2. SCRAMBLED LETTERS"
131  PRINT "3. UPSIDE-DOWN & BACKWARD"
133  PRINT
135  PRINT "WHICH CODE (1, 2, OR 3)";:INPUT CD
137  PRINT
138  PRINT "ENTER THE MESSAGE YOU WANT CODED."
140  PRINT
150  PRINT "IT MUST BE NO MORE THAN 5 LINES."
155  IF CD<>1 THEN 160
157  PRINT
158  PRINT "KEEP YOUR MESSAGE LINES SHORT!"
159  PRINT
160  FOR I=0 TO 4
170  PRINT
```

```
180 PRINT "LINE #";I+1;:INPUT L1$
182 IF L1$="" THEN I=5:GOTO 200
185 M1$(I*36+1, I*36+36)=L1$
190 C1=C1+1
200 NEXT I
210 PRINT :PRINT :PRINT "WHIRRR!! NOW CODING!!!" :PRINT
216 ON CD GOSUB 1010, 2010, 3010
220 PRINT :PRINT "DO YOU WANT TO SEE YOUR"
240 PRINT "ORIGINAL MESSAGE";:INPUT A$
250 IF A$="N" THEN 300
260 IF A$<>"Y" THEN 220
265 PRINT :PRINT
270 FOR I=0 TO C1
280 PRINT M1$(I*36+1, I*36+36)
290 NEXT I
300 PRINT :PRINT "DO YOU WANT TO SEE THE"
320 PRINT "CODED MESSAGE";:INPUT A$
330 IF A$="N" THEN 400
340 IF A$<>"Y" THEN 300
345 PRINT :PRINT
350 FOR I=0 TO C1
360 PRINT M2$(I*36+1, I*36+36)
370 NEXT I
400 PRINT :PRINT :PRINT "NEW CODE (1, 2, 3, OR 0
    FOR NO)";:INPUT CD
404 IF CD=0 THEN 410
406 GOTO 210
410 PRINT :PRINT :PRINT "DO YOU WANT TO CODE A
    NEW MESSAGE";:INPUT A$
420 IF A$="N" THEN PRINT :PRINT :END
430 IF A$<>"Y" THEN 410
440 GOTO 90
```

Most of the program is pretty simple. But there are some new tricks you should pay special attention to.

For example, look at the variables (memory cubbyholes) on Lines 60 to 80. The short variables, L1$ and L2$, hold only one message line at a time. L1$ holds a line from the original message. L2$ holds the same line after it has been coded (into code #1, code #2, or code #3).

The M1$ and M2$ variables, however, store the *entire* message. M1$ holds the message in its original form. M2$ holds the message after it has been coded.

When you turn your computer on, M1$ and M2$ will have random charges of electricity stored in them. These take the form of letters, numbers, and symbols—all in a meaningless jumble. (In computer jargon, this is known as "garbage.") You want M1$ and M2$ to store only your message, so you need to get rid of the garbage. The FOR-NEXT loop on Lines 90 to 94 fills M1$ and M2$ with empty spaces.

When it comes time to store your messages in M1$ and M2$, the FOR-NEXT loop on Lines 160 to 200 does the job. On Line 185, each line of the original message, L1$, is tacked onto the end of the message in M1$. As the computer gets new message lines, M1$ keeps growing.

## CODING EXPERTS

Do you remember the "Dragon World" chapter? In Dragon World, you learned about subroutines. Subroutines are little helper programs that assist the main program. The subroutine does its job, then returns control to the main program. We use subroutines in the Secret Agent program as coding experts.

The main program calls the subroutines on Line 216. Using the variable CD, it chooses which subroutine it will call. If you wanted to see code #1, then you typed a 1, and a 1 is stored in the variable CD. Since CD is a 1, the computer jumps to the *first* subroutine. The first subroutine is on Line 1000. It is an expert at the Ob Code:

```
1000 REM *** OB-CODE SUBROUTINE
1010 FOR I=0 TO C1
1012 L1$="                                                    "
1013 L2$="                                                    "
1015 C2=0
1020 L1$=M1$(I*36+1,I*36+36)
1030 FOR J=36 TO 1 STEP -1
1035 IF L1$(J,J)<>" " THEN S=J:J=1
1036 NEXT J
1038 FOR J=1 TO S
1040 C2=C2+1
1042 IF L1$(J,J)="Y" THEN 1047
1045 IF L1$(J,J)<> "A" AND L1$(J,J)<>"E" AND
     L1$(J,J)<>"I" AND L1$(J,J)<>"O" AND
     L1$(J,J)<> "U" THEN 1060
1047 L2$(C2,C2)=L1$(J,J)
1048 L2$(C2+1, C2+1)="B"
1049 C2=C2+2
1055 GOTO 1070
1060 L2$(C2,C2)=L1$(J,J)
1070 NEXT J
1080 M2$(I*36+1,I*36+36)=L2$
1090 NEXT I
1100 RETURN
```

If you wanted to see code #2, then you stored a 2 in the variable CD. On Line 216, the computer jumps to the *second* subroutine, which begins on Line 2000. The second subroutine is an expert at the Scrambled Letters Code:

```
2000 REM *** SCRAMBLE-CODE SUBROUTINE
2010 FOR I=0 TO C1
2012 L1$="                                                    "
2013 L2$="                                                    "
```

```
2020 L1$=M1$(I*36+1,I*36+36)
2030 FOR J=36 TO 1 STEP -1
2035 IF L1$(J,J)<>" " THEN S=J:J=1
2036 NEXT J
2040 FOR J=1 TO S
2050 L2$(J,J)=CHR$(ASC(L1$(J,J))+1)
2060 NEXT J
2070 M2$(I*36+1,I*36+36)=L2$
2080 NEXT I
2090 RETURN
```

If you wanted to see code #3, then you stored a 3 in the variable CD. On Line 216 the computer jumps to the *third* subroutine, beginning on Line 3000. The third subroutine is an expert at the Upside-Down and Backward Code:

```
3000 REM *** UPSIDE-DOWN AND
3005 REM *** BACKWARD CODE
3006 REM *** SUBROUTINE
3010 FOR I=C1 TO 0 STEP -1
3012 L1$="                                                    "
3013 L2$="                                                    "
3015 C2=0
3020 L1$=M1$(I*36+1,I*36+36)
3030 FOR J=36 TO 1 STEP -1
3035 IF L1$(J,J)<>" " THEN S=J:J=1
3036 NEXT J
3038 FOR J=S TO 1 STEP -1
3040 C2=C2+1
3060 L2$(C2,C2)=L1$(J,J)
3070 NEXT J
3080 M2$((C1-I)*36)+1,((C1-I)*36)+36)=L2$
3090 NEXT I
3100 RETURN
```

## GETTING RID OF EMPTY SPACES

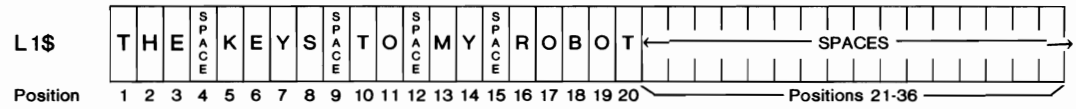
Look back at the Ob Code subroutine on Line 1030. Notice anything unusual? Right, the FOR-NEXT loop is running *backward*. It looks like this:

```
FOR J=36 TO 1 STEP-1
```

We'll return to this backward-counting loop in just a moment. First let's look at how the Ob Code subroutine gets started.

The Ob-Code subroutine, beginning on Line 1000, converts your message from English into the Ob Code. Using the FOR-NEXT loop, on Lines 1010 to 1090, it codes the message, one line at a time.

On Line 1020, a copy of the first line of your message is sent from the variable (cubbyhole) M1$ to the variable L1$. When you created L1$ (with the DIM command on Line 60), you reserved space for a message of up to thirty-six letters. But the message line that comes from M1$ might not be thirty-six letters long. It might be only twenty letters long, as in the example on page 52. What are in the remaining sixteen positions? Spaces—blanks:




You don't want those blanks turned into secret code. You want only your message coded. But how do you stop the computer from coding all thirty-six positions in L1$?

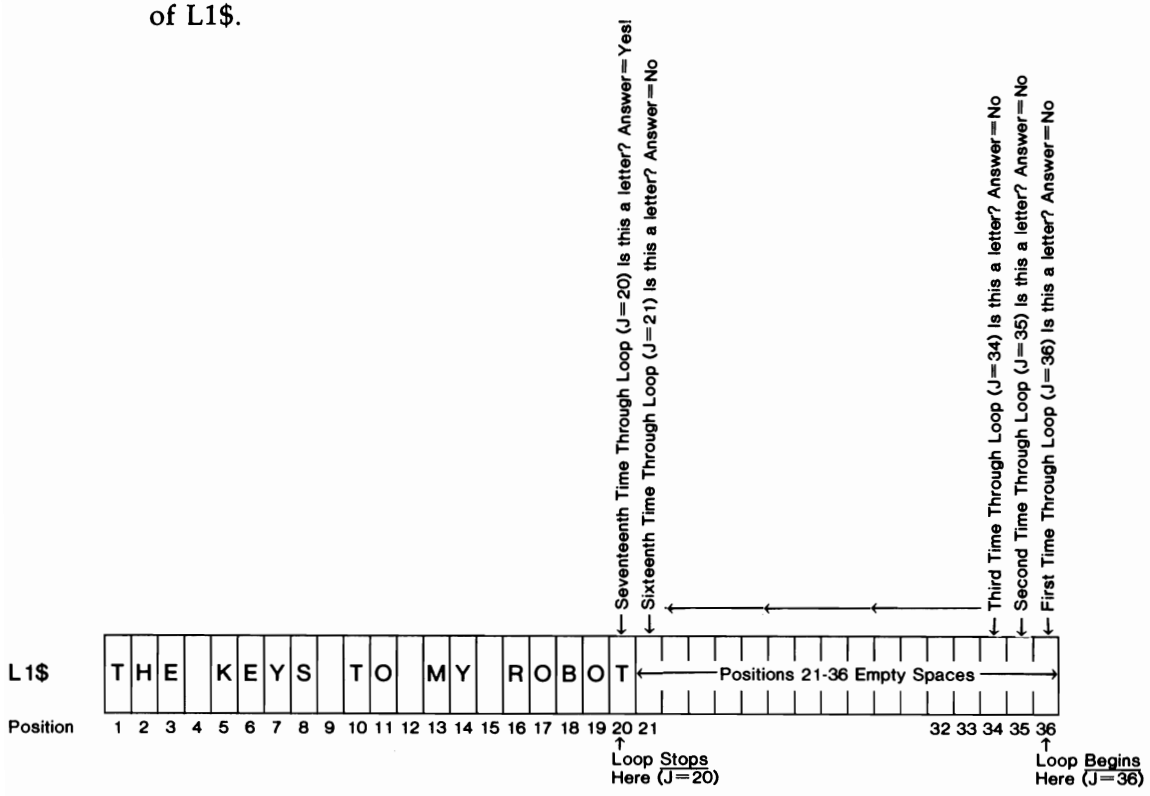
Use a backward-counting loop. Here's how the loop works.

Look at the FOR command on Line 1030. The computer counts backward, one number at a time, because a key word in that command, STEP, is set to −1. STEP tells the computer which way to count and by how much. If STEP = +1, then the computer

counts forward, one number at a time (for example, 1, 2, 3, and so on). If STEP = −12, the computer counts backward, twelve numbers at a time (for example, 60, 48, 36, 24, 12, 0).

In the FOR-NEXT loop on lines 1030 to 1036, STEP = −1, so the computer counts backward, one step at a time. The first time through the loop, the loop counter, J, is equal to 36. The next time through the loop, J=35. The next time, J=34, and so on, until finally J=1.

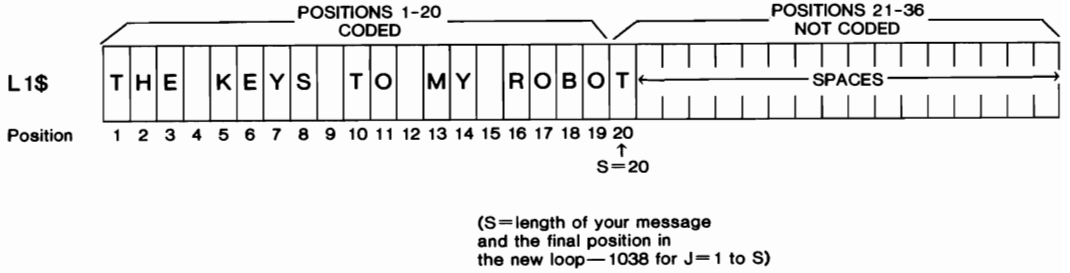
The computer begins the loop with J=36. It uses J to point to a position in L1$. Your message line is stored in L1$. When the loop begins, J is pointing to the last position in L1$. If there are spaces at the end of your message line, they will be at the tail end of L1$.



On Line 1035, the computer looks for those blanks. It begins with the last position in L1$ (position 36) and works backward toward the first position. L1$(J,J) is a "window" into L1$. It lets the computer look at only one position in L1$. When J=36, the computer looks at the thirty-sixth position. When J=35, the computer looks at the thirty-fifth position.

If the computer finds a blank in a certain position, it keeps counting backward so it can look for blanks in positions closer to the front of L1$. But the first time the computer finds a letter—a position in L1$ with something in it—it stops counting. It has found the end of your message.

What does it do next? It stops the loop. The length of your message is equal to J. The computer makes a copy of this number and stores it in S. Now it is ready to go on to the next loop (Lines 1038 to 1070) and code up your message. S is the final position in the new loop (look at Line 1038). That means the computer will only code the positions in L1$ that contain your message. It will not code any of the positions at the tail end of L1$ that have spaces:




## THE KANGAROO LOOP

In the Ob Code subroutine, the STEP −1 made the computer go backward one step at a time. In the next chapter, you will see a STEP 3 command. The STEP 3 command turns the computer into

a kangaroo. The computer doesn't walk forward, slowly and calmly, one step at a time, around the FOR-NEXT loop. Instead it takes giant steps. It hops forward *three* steps at a time.

## LIST OF VARIABLES

When you write a long program, it is good to divide it into a main program and lots of subroutines. It is also good to keep track of your variables—the little pigeonholes in the computer's memory where you stash things. As you create new variables, remember to reserve space for them with a DIM command. Then write down what each variable does on a piece of paper or in your program as a REM command.

Here are the variables used in the main Secret Agent coding program:

| | |
|---|---|
| A$ | One-letter answers |
| L1$ | Original message line |
| L2$ | Coded message line |
| M1$ | Entire original message |
| M2$ | Entire coded message |
| I | FOR-NEXT loop counter |
| C1 | Number of lines in message |
| CD | Number of code—code #1, #2, or #3 |
| S | Length of message |

## BUILDING A DECODING MACHINE

You now have a coding machine. But you still need a *de*coding machine.

Building a decoding machine is pretty simple. You start by making a new copy of the coding program. Almost everything in it can be used in a decoding program. You only have to make

some adjustments to two of the subroutines—the Ob Code subroutine and the Scrambled Letters subroutine. The Upside-Down and Backward subroutine you can leave intact. It works the same in both directions: You feed it a message in English, and it turns it upside down and backward. Then, if you feed it the upside-down and backward message, it converts it back to English.

You change the Ob Code subroutine by erasing Lines 1042 to 1055. In place of the old Ob Code lines, type in:

```
1064 IF L1$(J+1,J+2)<>"B " THEN 1070
1066 J=J+2
```

Now when you select code #1, the decoding machine will translate your message in Ob Code back into English.

It is even easier to change the Scrambled Letters subroutine. You just change the +1 at the end of Line 2050 to a −1. Now when you select code #2, the decoding machine will translate your message in Scrambled Letters Code back into English.

When you change the lines, remember: Enter the new lines *exactly* as they are shown here. Include all the blank spaces and parentheses. They are all important.

Also, be on the alert for decoding machine nonsense. When a B (B-space) is preceded by a consonant, it will be erased during the decoding process. For example, BULB IS will be coded as BUB LB IB S in Ob Code. Then the Ob decoder will convert it to BULIS.

How do you get around this? Simple. On Line 1065, add an IF-THEN command to see if L1$(J,J) is equal to a vowel—A, E, I, O, U, or the "special" vowel Y. If it is not, make the computer jump to Line 1070 (THEN 1070).

# CHAPTER TEN

# THE TIC-TAC-TOE MACHINE

Computers do only what you tell them to. But sometimes that makes them pretty smart.

Take the computers that have been taught how to play chess. These computers can execute tens of millions of commands in a single second. Their chess-playing programs are thousands of lines long. The computers are so good at chess, they are better than 95 percent of all human chess players. This means that only the experts are good enough to beat the computers.

People programmed these computers. People wrote all the hundreds and hundreds of commands that the computers obey. Yet the computers are, in many cases, better chess players than the people who programmed them!

## ARTIFICIAL INTELLIGENCE

Chess-playing computers are a product of a new field known as *artificial intelligence* (or AI). AI scientists program machines to do things that would be considered intelligent if done by a human being—things like painting, learning, solving complicated math problems, diagnosing an infection, and playing an expert game of chess.

A game is a good way to experiment with AI. A game is a lot

more manageable than the real world. It has a clear set of rules. A computer can learn these rules and the strategy it must follow to win the game.

In addition, a game usually has a limited number of steps to follow between its start and finish. Some games, like chess, are incredibly complicated and have an almost infinite number of possible steps. But most games are simpler, and the number of steps in each game is relatively limited. High-speed computers can analyze a game by "looking ahead" to all the steps they must take in order to win a game. By looking ahead, the computer can avoid defeat. It sees which steps its opponent might be planning and tries to block the opponent's moves.

## TIC-TAC-TOE

In this chapter, we will build a tic-tac-toe machine. That is, we will write a program that plays tic-tac-toe.

Tic-tac-toe is an extremely simple game. There are only nine possible moves, even at the beginning of the game. The smallest child can usually play tic-tac-toe, it is so simple.

But appearances can be deceiving.

Programming a computer to play a good game of tic-tac-toe will be the biggest challenge we have faced in this book. You should enter the program into your computer and study it carefully. You will learn a lot about programming and about making computers "smart" game players.

When you type **RUN**, this is what you will see:

```
HI! I'M AN EXPERT AT TIC-TAC-TOE!

I BET I CAN BEAT YOU!

DIRECTIONS
(Q=QUICK;D=DETAILED;N=NO)?D
```

```
LT  left top
CT  center top
RT  right top

LM  left middle
```

```
LT | CT | RT
LM | CM | RM
LB | CB | RB
```

CM center middle
RM right middle

LB left bottom
CB center bottom
RB right bottom

```
DO YOU WANT X'S OR O'S?X
```

The computer introduces itself and immediately challenges you by claiming that it is better than you are.

It asks you if you want any directions. You can request detailed directions (by typing D), quick directions (with a Q), or no directions (N). In the example, you type D for detailed directions.

Why do you need directions for tic-tac-toe? Because you need to tell the computer *where* you want your X or your O. You need a special two-letter code that tells the computer exactly which of the nine positions you want to choose. When you ask for directions, the computer prints out these codes and shows you which code belongs to which position.

Next, the computer asks whether you want to play with X's or O's. In the example, you type X to get X's.

The computer flips a coin to see who goes first (we'll see how later). You get to call the outcome—heads or tails. In this example, you choose heads. The computer flips the coin. Heads it is! You go first!

Next, you see:

```
YOUR MOVE (OR
D=DIRECTIONS)?LT
```

```
X |   |
  |   |
  |   |
```

```
I AM THINKING OF MY MOVE.
HERE IT IS:
```

```
X |   |
  | O |
  |   |
```

You won the coin flip and went first. The computer asked you where you wanted to place your first X. You typed LT for "left top." The computer drew a picture of the tic-tac-toe grid to show you how things look after your move.

But what happens if you forgot which two-letter code you need for which position? Easy. When the computer asks YOUR MOVE, you type in D. The computer draws the tic-tac-toe grid and shows you the positions of all the two-letter codes. Then it asks you again: YOUR MOVE? You pick the code you need and type it in.

Next it's the computer's turn. It spends a moment thinking about its next move. Then it makes the move and displays the tic-tac-toe grid for you to see where it placed its O. In this case, the computer chose to put its O in the center. The code for that is CM for "center middle."

Now you have the hang of it, and the game picks up speed. You choose the places to put your X's, and punch in the two-letter codes. The computer responds quickly.

The game is almost over. You make the last move. You choose RM for "right middle." The computer displays the board and announces the final score:

```
YOUR MOVE(OR D=DIRECTIONS)

        X | O | X
       -----------
        X | O | X
       -----------
        O | X | O
***** IT'S A TIE!! *****

PLAY AGAIN? NO
```

It asks if you want to play again. You've had enough for now, so you type in NO.

## THE PROGRAM

The Tic-Tac-Toe program is pretty complicated, so we divided it into a main program and lots of small, "expert" subroutines.

There are no new commands in the Tic-Tac-Toe program, but the old commands are used in interesting new ways. You should enter the program, play the computer a few times, and spend time going over each line until you understand what it does.

The main program is listed below. It introduces the program (Lines 100–139), lets you choose X's or O's (Lines 180–200), and keeps track of whose turn it is (Lines 220–310). At the end (Lines 350–390), it lets you play the game again or quit:

```
 50 REM *** TIC-TAC-TOE GAME
 55 DIM BD$ (18)
 57 BD$="CMLTRTLBRBCTLMRMCB"
 60 DIM BD(9)
 62 DIM H$(1),C$(5)
 65 DIM F$(8)
 67 DIM A$(1)
 70 DIM MV$(2)
 80 DIM GM(24)
 82 FOR I=1 TO 24:READ N:GM(I)=N:NEXT I
100 PRINT :PRINT
110 PRINT "HI! I'M AN EXPERT AT TIC-TAC-TOE!"
120 PRINT :PRINT
135 PRINT "I BET I CAN BEAT YOU!"
136 PRINT "                    ------"
138 PRINT
139 FOR T=1 TO 600:NEXT T
148 PRINT
150 PRINT "DIRECTIONS(Q=QUICK;D=DETAILED;
    N=NO)";:INPUT A$
160 IF A$="N" THEN 180
165 IF A$="D" THEN GOSUB 2010:GOTO 180
```

```
167 IF A$="Q" THEN GOSUB 2200:GOTO 180
170 GOTO 148
180 PRINT
190 PRINT "DO YOU WANT X'S OR O'S";:INPUT A$
192 IF A$="X" THEN HP=88:CP=79:GOTO 200
194 IF A$<>"O" THEN 180
196 HP=79:CP=88
200 PRINT
205 GOSUB 1510:REM * CLEAR BOARD
210 GOSUB 1010:REM * FLIP
220 W=0:TN=1:REM * SET GAME COUNTERS
230 IF F$(1,1)="C" THEN TN=1
240 FOR I=1 TO 9
250 IF TN=-1 THEN GOSUB 3010:TN=1:GOTO 280
260 GOSUB 4010:TN=-1
280 GOSUB 5010:REM * DISPLAY BOARD
290 IF I>4 THEN GOSUB 6010:REM * WINNER?
300 IF W<>0 THEN I=9
310 NEXT I
350 PRINT :PRINT
360 PRINT "PLAY AGAIN";:INPUT A$
370 IF A$="N" THEN PRINT :PRINT :END
380 IF A$<>"Y" THEN 350
390 GOTO 200
```

Next comes the first subroutine—the FLIP subroutine (Lines 1000–1120). This subroutine is pretty simple. The heart of it is in Line 1050. On Line 1050, the RND function makes the computer select a 1 or 2 at random (by chance). A 1 stands for "heads," a 2 for "tails." The subroutine lets you choose heads or tails. It makes the flip. Then it tells you whether you won or lost the toss:

```
1000 REM *** FLIP SUBROUTINE
1010 PRINT "FLIP TO SEE WHO GOES FIRST."
```

```
1020 PRINT
1030 PRINT "HEADS OR TAILS";:INPUT H$
1035 IF H$(1,1)<>"H" AND H$(1,1)<>"T"
     THEN 1020
1040 PRINT
1045 PRINT "I FLIP THE COIN."
1046 PRINT :PRINT
1047 FOR T=1 TO 500:NEXT T
1050 IF INT(RND(1)*2)+1=1 THEN
     C$="HEADS":GOTO 1080
1060 C$="TAILS"
1080 PRINT "        ";C$;"!";
1085 IF C$(1,1)<>H$ THEN 1100
1087 PRINT "  YOU WIN!":F$="YOU"
1088 PRINT :PRINT "        YOU GO FIRST!"
1089 GOTO 1115
1100 PRINT "  YOU LOSE!":F$="COMPUTER"
1110 PRINT :PRINT "        I GO FIRST!"
1115 PRINT
1120 RETURN
```

The next subroutine (Lines 1500–1540) is real tiny. But it does an important job. It clears the tic-tac-toe board.

The tic-tac-toe board, or grid, is represented as an array—a list of nine numbers all stored in the computer under the name BD (for "board"). There are nine numbers because there are nine possible moves in tic-tac-toe.

Only three things belong in the nine positions on the tic-tac-toe grid—an X, an O, or an empty space. In the array BD, the X is represented by its ASCII code number—an 88. The O is represented by a 79. The empty space is represented by a 32.

So what does the "Clear the Board" subroutine do? It wipes all the old moves off the tic-tac-toe grid and replaces them with empty spaces. The grid is represented by the nine numbers in the

array BD. So the subroutine stores a 32 (an empty space) in all nine numbers:

```
1500 REM *** CLEAR BOARD SUBROUTINE
1510 FOR I=1 TO 9
1520 BD(I)=32
1530 NEXT I
1540 RETURN
```

The next subroutine (Lines 2000–2280) is long but extremely simple. It prints out the directions to the game:

```
2000 REM *** DIRECTIONS SUBROUTINE
2010 PRINT :PRINT
2020 PRINT "USE A 2-LETTER CODE TO POSITION"
2030 PRINT "YOUR X OR O:"
2040 PRINT
2045 FOR T=1 TO 500:NEXT T
2050 PRINT "T, M, AND B STAND FOR THE ROW:"
2060 PRINT
2070 PRINT "     T = TOP ROW"
2080 PRINT "     M = MIDDLE ROW"
2090 PRINT "     B = BOTTOM ROW"
2100 PRINT
2110 PRINT "L, C, AND R STAND FOR THE COLUMN:"
2120 PRINT
2130 PRINT "     L = LEFT SIDE"
2140 PRINT "     C = CENTER"
2150 PRINT "     R = RIGHT SIDE"
2160 PRINT
2180 PRINT "USE 2 LETTERS FOR EACH POSITION:"
2200 PRINT :PRINT
2210 PRINT "        LT | CT | RT"
2220 PRINT "        _____"
```

```
2230 PRINT "       LM | CM | RM"
2240 PRINT "       _____"
2250 PRINT "       LB | CB | RB"
2260 PRINT :PRINT
2280 RETURN
```

The next subroutine (Lines 3000–3090) is the heart of the whole game. It is where the computer decides on its next move:

```
3000 REM *** COMPUTER TURN SUBROUTINE
3010 PRINT
3020 PRINT "I AM THINKING OF MY MOVE."
3030 FOR T=1 TO 300:NEXT T
3040 PRINT
3050 PRINT "HERE IT IS:"
3055 IF I>3 THEN GOSUB 3110:IF J=26 THEN 3090
3057 IF I>2 THEN GOSUB 3210:IF J=26 THEN 3090
3060 FOR J=1 TO 9
3070 IF BD(J)=32 THEN BD(J)=CP:J=9
3080 NEXT J
3090 RETURN
```

The computer follows three simple strategies. First, on Lines 3100 to 3180, it follows a "win first" strategy. It checks the tic-tac-toe grid to see if it can win the game with a certain move. If so, the computer makes that move:

```
3100 REM *** COMPUTER WIN? SUBROUTINE
3110 FOR J=1 TO 22 STEP 3
3120 IF BD(GM(J))+BD(GM(J+1))+BD(GM(J+2))
     =2*CP+32 THEN F=J:J=23
3130 NEXT J
3140 IF J<>26 THEN 3180
3150 FOR J=F TO F+2
```

```
3155 IF BD(GM(J))=CP THEN 3170
3160 BD(GM(J))=CP:J=25
3170 NEXT J
3180 RETURN
```

Second, if the first strategy didn't work, the computer follows the "block your move" strategy, which appears on Lines 3200 to 3280. The computer searches the tic-tac-toe grid to make sure you are not about to defeat it. If you are, the computer counters your move.

```
3200 REM *** COMPUTER SAVE SUBROUTINE
3210 FOR J=1 TO 22 STEP 3
3220 IF BD(GM(J))+BD(GM(J+1))+BD(GM(J+2))
     =2*HP+32 THEN F=J:J=23
3230 NEXT J
3240 IF J<>26 THEN 3280
3250 FOR J=F TO F+2
3255 IF BD(GM(J))=HP THEN 3270
3260 BD(GM(J))=CP:J=25
3270 NEXT J
3280 RETURN
```

Third, if the first two strategies fail, the computer follows the "best possible move" strategy outlined on Lines 3060 to 3080. The computer has a priority list of moves, from best to worst. It checks the positions on the tic-tac-toe grid, starting with the best move, working its way down to the worst move, and putting an X or an O in the first empty space it finds.

## COMPUTER INTELLIGENCE

The computer's strategies seem pretty smart, but they are based on searching through simple lists of "good moves" using FOR-

NEXT loops. The main list of moves is stored in the nine positions in the BD array. The positions are ordered in terms of how good a move they are.

I set up the array, so the ordering is based on my judgment, not the computer's. I looked at the nine positions on the tic-tac-toe grid and numbered them from "best" move to "worst" move:

```
2 | 6 | 3
---------
7 | 1 | 8
---------
4 | 9 | 5
```

In my judgment, the best move was the center position, so I gave it the number 1. The next best move seemed to be any of the four corner positions, so I gave them the numbers 2 through 5. The outer-middle positions were given the numbers 6 through 9, since they seemed to be the least desirable positions.

These positions are represented (in "best-to-worst" order) in the array BD. BD(1) is the best position (the center). BD(9)—the bottom middle—is the worst.

But how is the computer to know if a single move separates it from either victory or defeat? What if the computer has two O's in a row? How does it know? Or what if you have two X's in a row? How does the computer know that?

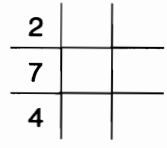
This is where another array—GM—comes in. GM is a list of twenty-four numbers. The numbers are stored in GM, from GM(1) to GM(24), on Line 82, using a FOR-NEXT loop and a READ command. The READ command reads the numbers from a DATA command on Line 7000:

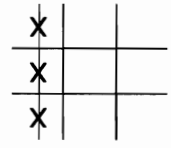
```
7000 DATA 2,4,7,1,6,9,3,5,8,2,3,6,1,7,8,4,5,9,1,2,5,1,3,4
```

The twenty-four numbers represent positions on the tic-tac-toe grid (see the above diagram) and can be divided into eight sets of

three numbers. You need three X's or three O's in a straight line to win a game. Each of these eight sets represents the "three X's in a row" or "three O's in a row" needed to win a game.

For example, suppose you have GM(1)=2, GM(2)=4, and GM(3)=7. If you put an X in each of these positions, you will beat the computer and win the game:




The other seven sets of three numbers are the same. They all represent possible ways of getting three X's (or O's) in a row and winning the game.

Now you have the key to the strategy the computer follows in the "win first" subroutine (Lines 3100-3180) and the "block the opponent" subroutine (Lines 3200-3280). In both subroutines, the computer searches the tic-tac-toe grid and uses the GM array to see if someone is about to win.

If the computer finds two of its own O's in a row and a blank space next to them, then it puts a third O in the blank space—and wins!

But if the computer finds two of your X's and a blank space, then it knows you will probably win on your next turn. So it blocks you. It puts one of its O's in the blank space. Now you can't win.

## YOUR TURN

The next subroutine in the program (Lines 4000-4080) is where the computer interprets and stores *your* move:

```
4000 REM *** HUMAN TURN SUBROUTINE
4010 PRINT
```

```
4020 PRINT "YOUR MOVE(OR D=DIRECTIONS)";:
     INPUT MV$
4030 IF MV$="D" THEN GOSUB 2200:GOTO 4020
4040 FOR J=1 TO 17 STEP 2
4050 IF BD$(J,J+1)<>MV$ THEN 4060
4052 IF BD((J+1)/2)=32 THEN BD((J+1)/2)=
     HP:J=18:GOTO 4060
4054 PRINT :PRINT
4055 PRINT "THAT SPACE IS FILLED!! TRY
     AGAIN.":PRINT :J=17
4060 NEXT J
4070 IF J<>20 THEN 4010
4080 RETURN
```

This subroutine is where the computer translates the two-letter code you give it into the ASCII code representing the letter X (88)—or the letter O (79), if you chose to play with O's instead of X's.

To make the conversion, the computer uses another array, BD$. BD$ stores eighteen letters (see Line 57). Eighteen letters make nine two-letter pairs—the nine two-letter codes that represent all the positions on the tic-tac-toe grid. They are stored in the same order as in the BD array—from the "best" position to the "worst" position.

When you enter a two-letter code, the computer checks: (1) to see if it is valid, and (2) to see if it is for a position that is already occupied (and thus off limits). If it is a valid code and for an empty position, the computer converts the two-letter code to the number code for your X or O and stores the number in the correct position in the BD array.

## DISPLAY THE GAME

The next subroutine (Lines 5000–5090) displays the current status of the game:

```
5000 REM *** DISPLAY SUBROUTINE
5010 PRINT :PRINT
5020 PRINT "        ";CHR$(BD(2));"I";
     CHR$(BD(6));"I";CHR$(BD(3))
5030 PRINT "        _____"
5040 PRINT "        ";CHR$(BD(7));"I";
     CHR$(BD(1));"I";CHR$(BD(8))
5050 PRINT "        _____"
5060 PRINT "        ";CHR$(BD(4));"I";
     CHR$(BD(9));"I";CHR$(BD(5))
5070 PRINT :PRINT
5090 RETURN
```

This is a simple subroutine. It draws the tic-tac-toe grid and uses the CHR$ function to convert the ASCII code number for X's (an 88), O's (a 79), and empty spaces (a 32) into X's, O's, and empty spaces on the grid. After each turn, the computer draws the grid to show you how the game is going.

### WHO'S THE WINNER?

The next subroutine is long but simple. It makes the computer search through the tic-tac-toe grid looking for a winner. If it finds three X's in a row, it declares you the winner. If it finds three O's in a row, it declares itself the winner. If it finds neither, it declares a tie:

```
6000 REM *** WINNER? SUBROUTINE
6010 FOR J=1 to 22 STEP 3
6020 IF CP=BD(GM(J)) AND CP=BD(GM(J+1)) AND
     CP=BD(GM(J+2)) THEN W=W+1:J=22
6030 NEXT J
6040 FOR J=1 TO 22 STEP 3
6050 IF HP=BD(GM(J)) AND HP=BD(GM(J+1)) AND
     HP=BD(GM(J+2)) THEN W=W+2:J=22
```

```
6060 NEXT J
6064 PRINT :PRINT
6065 IF W=0 AND I=9 THEN 6150
6070 IF W=0 THEN 6210
6080 PRINT :PRINT
6090 IF W<>1 THEN 6120
6100 PRINT " *** I WIN!! I WIN!! ***"
6110 GOTO 6200
6120 IF W<>2 THEN 6150
6130 PRINT " *** YOU WIN!! YOU WIN !! ***"
6140 GOTO 6200
6150 PRINT " ***** IT'S A TIE!! *****"
6200 PRINT :PRINT
6210 RETURN
```

# CHAPTER ELEVEN

## NOW, INVENT YOUR OWN GAMES!

There are dozens of different kinds of computer games, including sports games, board games, strategy games, word games, number games, and adventure games. But the best of these games have certain things in common.

First, they have *action*. There is nothing worse than a boring game, where nothing happens. When you create your own games, think of ways to speed up the action.

Second, a good game has *suspense*. Keep your human game player hanging. You create suspense when your player expects something to happen. It might be good or bad. He or she doesn't know which—until it happens!

Third, a good game sometimes depends on *surprise*. One of the ways to create surprise is to make some parts of the game depend on luck. Luck makes things unpredictable. Your game player doesn't know what will happen next.

When you are thinking up your own games, here are some general rules you might follow:

First, when you can, add sound effects, music, and color. This makes the game more realistic and exciting.

Second, when you are thinking up a new game, pretend that you are the computer. Think up the messages the computer would print on the video screen. Think up the questions it might

ask. Think of what the computer will do if the person playing the game gives a certain answer.

Play the whole game as if you were the computer. Write down the steps the computer follows. This will be a big help later on when you try to write the game program. Then you can take each step and convert it into the computer's language.

Third, get a package of graph paper. Your video screen is divided into little squares, just like the graph paper. Find out how many rows (lines from top to bottom) and how many columns (lines from left to right) there are on your screen. Count out the rows and columns on the graph paper. Draw a line around the edges to create a graph-paper "copy" of the video screen.

You can use the graph paper to figure out where you are going to put your game pictures and messages. You can draw game pictures (monsters, faces, weapons) by putting little X's into the squares on the graph paper. Later, you can type the commands into the computer that fill in the squares on the video screen to make the pictures you want.

Fourth, when you are ready to type in your game program, worry first about the main things your game is supposed to do, and save the details for later. Details in computer programs can drive you batty! Missed details are the *bugs* (or errors) that every game inventor dreads. So don't get bogged down in the game details right at the start. Otherwise, you may never finish writing the game.

Fifth, don't try to write your game program in one sitting. That's like trying to invent a time machine while you're eating lunch or watching TV. Maybe it can be done, but it's not easy.

Instead, divide your program up into steps. Each step is like a miniature program—a subroutine, or helper program. Write up each step, one at a time, and test it on the computer. Does it work? Yes? Then it's time to go on to the next step. This step-by-step approach (sometimes called "Divide and Conquer") will make complicated games easier to program.

Sixth, don't write a game for yourself. You know what the game does, how it's supposed to be played, and what things to type into the computer. Instead, write the games for a complete beginner—maybe someone who has never even seen a computer. This will make your games easy to understand and easy to play.

Seventh, be ready to test your games until you're ready to drop. Computer bugs lurk in the dark, deep sections of your games. As soon as you think you've wiped them all out, a new one leaps out. It's good to catch most of these bugs before you try a new game out on your family or friends.

And last, no matter how easy your program is, and no matter how hard you've searched for bugs, expect someone to "blow it up" the first time they sit down and play.

You never know what things people will type in when they play a computer game. Anyone can make a mistake, punch the wrong key, or give a weird answer. You can't prepare for everything, but you can hide error traps in your program. These catch most mistakes people make. They print out a message like "I don't understand that . . . Try again." Then they give the person another chance to type in the right answer.

Now get going and invent your own games.

And good luck!

# GLOSSARY

*ABS.* A BASIC function. The ABS function makes a negative (minus) number positive.

*BASIC.* The most popular language for small computers. BASIC is a large program that is usually stored in the computer's circuits, so it is ready to use as soon as you turn on the computer. When you type commands into the computer, BASIC translates them into computer bits (0's and 1's). The bits represent pulses of electricity.

*Bugs.* Mistakes or errors in a program.

*Commands.* Instructions to the computer in a language that it understands. The games in this book are made up of commands in the BASIC language. You enter the commands into the computer's memory by typing them in on the computer's keyboard. When you type RUN, the computer obeys your commands.

*DATA.* A BASIC command. After the word "DATA," you put information (data) needed for a particular game. The computer reads the information into its memory with a READ command. See also *READ.*

*DIM.* Short for the BASIC command called "Dimension." Lets you name a variable (cubbyhole) in the computer's memory and reserve enough space to fit letters or numbers in the variable.

*END.* A BASIC command. When the computer comes to the END command, it assumes that your program is finished and it doesn't look for any new commands. The program stops, and the computer types "READY."

*Error traps.* When a computer program asks a person for information, the computer should check for a mistake or an unexpected answer. For example, it can use an IF-THEN command to try to match the person's answer with the answer it expects. If the answers don't match, the computer can jump back (with a GOTO command) and ask the question again. It might also print a message ("SORRY . . . TRY AGAIN.").

*FOR-NEXT.* A pair of BASIC commands that put the computer into a loop. The instructions for the loop are sandwiched between the FOR command and the NEXT command. See also *Loop.*

*Function.* A "helper" command, such as RND, INT, LEN, or ABS. See also individual entries for explanations.

*GOSUB.* A BASIC command followed by a line number. When the computer sees a GOSUB command, it jumps to the designated line in the program. That line is the first line in a subroutine. The computer obeys the subroutine commands until it comes to a RETURN command. Then it "returns" to the command immediately after the GOSUB command. See also *Subroutine.*

*GOTO.* A BASIC command followed by a line number. When the computer sees a GOTO, it jumps to the designated line in the program. Unlike GOSUB, GOTO is not followed by a RETURN command.

*Graphics.* Computer pictures. Most personal computer languages have commands that make the computer "draw" pictures on the video screen. These commands (such as DRAWTO and PLOT) are known as graphics commands.

*IF-THEN.* A BASIC command that makes computers obey the command if certain conditions are true. Directly following the IF part of the command is the condition. If the condition

is true, then the computer obeys the command following the word THEN.

*INPUT.* A BASIC command that causes the computer to print a question mark on the video screen, then stop and wait for you to type in some information. When you press the ENTER or RETURN key on your keyboard, the information you have just entered is stored in the variable (cubbyhole) whose name follows the word "INPUT."

*INT.* A BASIC function that rounds off a fraction and converts it into a whole number (integer).

*LEN.* A BASIC function that identifies the number of letters or punctuation symbols stored in a string variable (cubbyhole).

*LET.* A BASIC command that allows the computer to make a copy of a letter, one or more words, or number. Then it stores that copy in a variable (cubbyhole) in the computer's memory.

*Line number.* The first component of a line of instruction in a BASIC program. The computer obeys the commands in the program by keeping track of the line numbers. It obeys the commands with the lowest number first, then the next highest number, and so on.

*Loop.* The obeying by a computer of the same commands (on the same line numbers) over and over again. In BASIC, the loop begins with a FOR command and ends with a NEXT command.

*PRINT.* A BASIC command that causes the computer to display a message on the video screen.

*Program.* A list of commands, or instructions, to the computer in a computer language. In BASIC, each line in the program begins with a line number.

*READ.* A BASIC command related to the DATA command. When the computer sees the READ command, it looks for the DATA command further down in the program. The computer makes a copy of the value following the DATA command

and stores it in the variable (cubbyhole) following the word "READ."

*REM.* A BASIC command standing for "remark" or "remember." REM tells the computer that this is a message that the person is writing to himself or herself. The computer ignores the information following the REM command.

*RETURN.* A BASIC command in a subroutine. When the computer sees a RETURN command, it "returns" to the command immediately after the GOSUB command that started the subroutine. Also, RETURN is a key on the computer keyboard that is used to get the computer to accept the information on the line just typed in. See also *GOSUB.*

*RND.* A BASIC function that calculates, or chooses, a random number.

*RUN.* A BASIC command used to get a computer to look in its memory for a program (list of commands). It immediately begins obeying the program, one line at a time, starting with the lowest line number.

*Subroutine.* A small "helper" program inside of a larger program. When the computer sees a GOSUB command, it begins obeying the subroutine on the line indicated. See also *GOSUB.*

*Variable.* A "cubbyhole" in the computer's memory with a name such as A$, X, or NAME. You create a variable with the DIM command. Variables that store numbers used in arithmetic are called *numeric variables.* Variables that store letters, and numbers not used in arithmetic operations are called *string variables.*

# FOR FURTHER READING

## MAGAZINES

*Digit* (monthly). 2342 North Point, San Francisco, CA 94123. 415/565-3221. Features programs for kids.

*Family Computing* (monthly). Scholastic Inc., 730 Broadway, New York, NY 10003. 212/505-3000.

## BOOKS


Ahl, David H., ed. *BASIC Computer Games (Microcomputer Edition).* Morristown, NJ: Creative Computing Press, 1978.

————————. *More BASIC Computer Games.* Morristown, NJ: Creative Computing Press, 1979.

Boom, Michael. *Understanding Atari Graphics.* Sherman Oaks, CA: Alfred Handy Publishing Co., Inc., 1982.

Chiu, Lin, and Henry Mullish. *CRUNCHERS: 21 Simple Games for the Timex/Sinclair 1000 (2K).* New York: McGraw-Hill/VTX, 1983.

D'Ignazio, Fred. *Atari in Wonderland: 43 Learning Games on Your Atari Computer.* Rochelle Park, NJ: Hayden Book Company, 1983.

————————. *Electronic Games.* (a First Book). New York: Franklin Watts, Inc., 1982.

Lipscomb, Susan Drake, and Margaret Ann Zuanich. *BASIC Fun: Computer Games, Puzzles, and Problems Children Can Write.* New York: Avon/Camelot, 1982.

Mateosian, Richard. *Inside BASIC Games.* Berkeley, CA: Sybex, 1981.

Moore, Herb, Judy Lower, and Bob Albrecht. *Atari Sound and Graphics: A Self-Teaching Guide.* New York: John Wiley & Sons, Inc., 1982.

Myers, Roy E. *Microcomputer Graphics (With Apple II Examples).* Reading, MA: Addison-Wesley Publishing Company, 1982.

Thornburg, David D. *Picture This! (An Introduction to Computer Graphics for Kids of All Ages).* Reading, MA: Addison-Wesley Publishing Company, 1982.

# INDEX

For a thorough understanding of computers and computer science, be sure you have *all* the titles in Franklin Watts' new Computer-Awareness First Book series. Titles just published include:

CAREERS IN THE COMPUTER INDUSTRY
COMPUTER LANGUAGES
COMPUTERS IN OUR WORLD, TODAY AND TOMORROW
DATA PROCESSING
INVENT YOUR OWN COMPUTER GAMES
PROGRAMMING IN BASIC

And coming soon:

COMPUTER GRAPHICS
ROBOTS
THE SCIENCE OF ARTIFICIAL INTELLIGENCE
WORD PROCESSING

Also coming soon are the Watts' Computer Literacy Skills books, which will teach popular programming languages to beginning programmers. Titles include:

BASIC FOR BEGINNERS
COBOL FOR BEGINNERS
FORTRAN FOR BEGINNERS
PASCAL FOR BEGINNERS

A FRANKLIN WATTS LIBRARY EDITION
531-04637-0