FORTH
FOR BEGINNERS

FORTH for Beginners

FORTH FOR BEGINNERS

By Christopher Lampton

A Computer Literacy Skills Book
FRANKLIN WATTS 1985
New York London Toronto Sydney

Library of Congress Cataloging in Publication Data

Lampton, Christopher.
Forth for beginners.

(A Computer literacy skills book)
Includes index.
Summary: Introduces the computer language Forth and explains how to use it to write programs.
1. FORTH (Computer program language. [1. FORTH (Computer program language) 2. Programming (Computers) 3. Computers] I. Title. II. Series.
QA76.73.F24L36 1985 001.64'24 84-20865
ISBN 0-531-04849-7

CONTENTS

FORTH for Beginners

1 AND SO . . . FORTH

Forth, if we may start this book with a statement that will sound like an exaggeration, is unlike any other programming language that you are likely to encounter. If you are already familiar with a programming language—BASIC, for instance, or Pascal—you will be surprised by Forth. You might even think it a trifle bizarre. Possibly you will be infuriated by it. More likely, though, you will find it exciting, stimulating, and immensely useful. And you certainly won't find it boring.

Forth was developed in 1971 by Charles Moore, for the purpose of controlling telescopes in an astronomical observatory. Because he considered his new language to be a "fourth-generation" programming language, he called it "Fourth." But the computer on which he developed it restricted the names of programs to five letters. Thus "Forth" was born.

In this book, we will tell you how to write programs in Forth. This is not an exhaustive study of the language; it is an introduction. This book is written for two types of people: those who already know how to program a computer in a high-level programming language and those who don't, categories that should include just about everyone on this planet. For those who do know how to program, we will occasionally draw analogies between Forth and existing languages, particularly BASIC, showing the ways in which Forth resembles more conventional programming languages and the ways in which it does not. For those who don't already know how to program a computer, we will offer background explanations of what it is that computers do and the ways in which we can encourage them to do it.

The latter group, for instance, may not be entirely clear what we mean by computer programming or computer programming languages. For those readers, we will now offer a brief explanation. More experienced readers may skip ahead.

A computer is a device for processing information. What kind of information does a computer process? Just about any kind you can imagine. Information enters the computer in one form and leaves it in another.

As it enters the computer, the information is converted into a series of numbers, according to special encoding schemes. While being processed, these numbers are stored within a series of electronic circuits called the computer's *memory.* Each *memory cell* within a typical computer—that is, each of these electronic circuits—can hold a number from 0 to 255. And each memory cell is itself identified by a special number, called an *address,* so that we can identify the cells in which we stored specific pieces of information when we need to get that information back out.

A *program* is a series of instructions telling the computer how we want this information to be processed. These

Address—A number used to identify a specific memory cell within the memory of a computer.

Memory—A series of electronic circuits within a computer, used to store information and programs.

Memory cell—A single memory circuit within a computer, capable of holding a number between 0 and 255.

Program—A series of instructions to a computer, expressed in a programming language.

instructions are written in a special language called a *programming language.*

There are many different programming languages. However, there is only one language that is actually "understood" by a computer. It is called *machine language,* and it is made up purely of numbers.

Because it is awkward to program in numbers, computer experts long ago developed *high-level computer languages*—Englishlike languages that can be used to write computer programs. But before a program written in one of these languages can be executed by a computer—that is, before the instructions can be obeyed—the program must be translated into machine language. We can perform this translation by hand, of course, but it is traditional (and far easier) to let the computer do it for us. This translation is performed by a computer program, itself written in machine language, called a *compiler* or an *interpreter.* The difference between a compiler and an interpreter is that a compiler performs the translation before the program is

Programming language—A symbolic representation used for encoding instructions to a computer.

Machine language—The only programming language directly understood by a computer.

High-level computer languages—Computer programming languages designed for relatively easy use by human beings but which must be translated into machine language before they can be understood by a computer.

Compiler—A program that translates high-level programs (that is, programs written in high-level languages) into machine-language programs.

Interpreter—A program that translates the instructions in a high-level program into machine-language instructions; it does this while the program is being executed.

executed; an interpreter performs its translation *while* the computer is executing the program.

Theoretically, any high-level language can be either interpreted or compiled, depending on the whim of the programmer who develops the language system. In practice, however, certain languages are traditionally interpreted and certain other languages are traditionally compiled. BASIC, Logo, Lisp, and APL fall into the former group; FORTRAN, COBOL, C, and PL/1 fall into the latter. And a few maverick languages exist in a strange world somewhere between interpretation and compilation. Pascal is one of these. A Pascal program is first compiled into a kind of artificial machine language called *P-code* (short for pseudo-code), which is then interpreted as it is executed.

Forth is another maverick. Strictly speaking, Forth is neither a compiled nor an interpreted language; it is a *threaded language.* For instance, this means that a Forth system—the set of programs that allow you to program in Forth—contains *both* an interpreter and a compiler.

To give you a taste of Forth programming, and to show you some of the ways in which Forth differs from other high-level languages, such as BASIC, let's develop a short program routine in BASIC and then show how the same routine would look in Forth.

One of the simplest routines we could write would be one that adds two numbers together—say, 5 and 8—and prints the sum of the numbers on the computer's video display. In BASIC, we would write such a routine like this:

```
PRINT 5 + 8
```

BASIC programmers should have no trouble seeing how this short routine works. The keyword PRINT tells the com-

P-code—Short for pseudo-code; an artificial machine language that must be interpreted before it can be executed by a computer.

Threaded language—A language that stores its programs in a computer's memory using a specific form of representation called threaded code.

puter to display the result of the arithmetic operations that follow. The plus sign (+) tells the computer that it can find this result by adding 5 and 8. Thus, the computer prints the number 13 on the video display. In Forth, the same routine would look like this:

```
5 8 + .
```

Peculiar looking, isn't it? Hard to tell how this routine displays *anything* on the computer's screen. Yet, to a Forth programmer, this short routine is identical to the BASIC routine above.

In fact, if we look closely, we can see that every element of this Forth routine has a corresponding element in the BASIC routine (though you should not expect this to be true of all Forth programs). The two numbers, 5 and 8, are visibly present in both routines; so is the plus sign. Only the PRINT command seems to be missing in the Forth version. Thus, we can guess that this symbol:

```
.
```

must be the Forth equivalent of PRINT. And so it is. This symbol, which looks just like a conventional period but is pronounced *dot,* is a Forth word that tells the computer to print a number on the video display. (The dot symbol can be used only for printing numbers, not letters or words. Forth uses other words for these tasks.) In this case, the number that is to be printed is the sum of 5 and 8, or 13.

And yet, even though we have explained what each element of the Forth routine does, it still looks strange. Why? Because of the order in which these elements appear. The numbers appear first. The plus sign follows the numbers. The dot symbol follows everything. This is not the order in which we have been taught to write such things. It doesn't look right. It looks . . . backwards.

Ordinarily, when we write an *arithmetic expression*, we

Arithmetic expression—A sequence of numeric values and arithmetic operators (see below) that can be reduced to a single numeric value.

use a system called *infix notation.* This simply means that the *arithmetic operators*—that is, the symbols such as the plus sign, the multiplication sign, etc., that represent arithmetic operations—in the expression are "fixed" in between the numbers that they operate on, like this:

```
1 + 2
```

We know automatically, when we see the plus sign, that we are to add together the two numbers that surround it. The two numbers surrounding the operator—the numbers operated on by the operator—are called the *operands* of that operator. Operators always take two operands. If one of those operands is missing, as in this expression:

```
2 +
```

then we know instantly that something is wrong.

Infix notation seems like the natural way to perform arithmetic operations, so natural that it may have surprised you to learn that it even had a name. There are alternative ways of writing an arithmetic expression, however, and the only reason they seem less natural than infix is that most of us were taught infix notation when we were very young.

In Forth, we write expressions in *reverse Polish nota-*

Infix notation—A method of representing arithmetic expressions in which arithmetic operators are placed between the numeric values on which they operate.

Arithmetic operators—Symbols that represent arithmetic operations, such as addition (+), subtraction (−), division (/), multiplication (*), etc.

Operands—The values on which an operator operates.

Reverse Polish notation (postfix notation)—A method of representing arithmetic expressions in which arithmetic operators are placed after the numeric values on which they operate.

tion, so called because it was originally developed by a Polish mathematician, Jan Lukasiewicz. Reverse Polish is also referred to as *postfix notation.* (We will use both terms in this book.) In a postfix expression, the symbol for the operator is "fixed" *after* its operands, like this:

```
1 2 +
```

This means exactly the same thing as this:

```
1 + 2
```

except that it is written in postfix rather than infix notation; only the position of the operator is different. Postfix may take some getting used to, but it's really just as simple as infix.

We can also use postfix notation on more complex expressions. For instance, in a BASIC program we might write this expression:

```
2 * 4 + 18
```

This would multiply 2 by 4. (The asterisk [*] is the symbol for multiplication in most programming languages.) Then it would add the result to 18. In Forth, we would write this expression:

```
2 4 * 18 +
```

With our newfound knowledge of reverse Polish, it shouldn't be too hard to decipher this expression. The reverse Polish expression

```
2 4 *
```

means the same thing as the infix expression

```
2 * 4
```

That is, it tells the computer to multiply 2 by 4. The portion of the expression that follows this—

```
18 +
```

—might be a little harder to understand; essentially, it tells the computer to add 18 to the result of the preceding operation. In this example, the result of the previous operation

```
2 4 *
```

will be 8; thus, the computer is to add 18 and 8.

Converting subtraction and division to postfix is slightly trickier, because the order of the operations becomes significant. We must remember that the operands in reverse Polish appear in the same order as in infix, even though they precede the operator rather than surround it. Thus, we would write this expression:

```
7 - 3
```

like this:

```
7 3 -
```

Division, which is performed with the operator / , is handled similarly. This expression:

```
9 / 3
```

or "nine divided by three," would be written like this:

```
9 3 /
```

With this knowledge, we can look back at our Forth routine:

```
5 8 + .
```

with a better understanding of why it is written in the fashion that it is. The first part of the routine is a postfix expression that tells the computer to add 5 and 8. This is followed by the dot symbol, which tells the computer to display the numerical result of this expression. Note that, in Forth, instructions to the computer always follow the data to be manipulated by that instruction. In Forth, entire sequences of instructions to the computer are written in postfix notation. Reverse Polish notation pervades the entire structure of the language.

At this point, you might be asking why Forth can't just use infix notation, like most other programming languages. Why should you be required to learn a whole new system of notation in order to write Forth programs?

Well, as we shall see in the next chapter, reverse Polish notation is essential to the very nature of Forth. The language could not function without it. When Charles Moore designed the Forth language, he did not adopt postfix notation just to be cute or different; he adopted it because it offers some distinct advantages in speed and versatility. Forth is one of the most powerful and adaptable of high-level programming languages, and postfix notation is one of the reasons why. To understand this, we'll have to take a closer look at how a Forth system works—in particular, the way that the Forth interpreter views the words of the Forth language.

Before we do this, however, let's quickly review the kinds of arithmetic we can perform using Forth. Most versions of Forth offer five primary arithmetic operators:

+	addition
−	subtraction
*	multiplication
/	division
MOD	remainder

All of these operators work with *integer numbers*—that is, numbers that can have no fractional portions and that must remain within a specific numeric range, usually from −32767 to 32768. There are Forth operators, called double-length operators, that will work with larger Forth numbers, called double-length numbers, but we will not be discussing these numbers or operators in this book. It is also possible to work with fractions in Forth, though we will not be discussing those procedures in this book, either.

Integer numbers (integers)—Whole numbers; numbers that may not have fractional values.

You might be wondering how we can perform division operations when working with integers, since division frequently produces fractional results. The answer is that we really can't. That is, we can perform division, but we will not receive precise results, because any fractional portions of the quotient (the result of the division operation) will be lopped off. This process is called *integer division*—division without fractional values—and the result of the operation is called the *integer quotient.* Because integer quotients will not always be sufficient for our purposes, Forth also offers the MOD operator, which can be used in place of the division operator to produce only the remainder of a division operation. BASIC programmers may not be familiar with the MOD operator, though Pascal programmers should recognize it. If, for instance, we want to know what the remainder would be if we divided 10 by 3, we would write:

```
10 3 MOD .
```

Note that the order of the operands in this postfix expression is the same as it would be in a division operation. This routine will print the number 1—the remainder when 10 is divided by 3—on the video display.

There is one last point to bear in mind when writing arithmetic expressions in Forth. That is, there is no precedence of operators in reverse Polish. In algebra, and in most computer languages, arithmetic operations within a single expression are performed in a predetermined order. Multiplications and divisions are performed first, additions and subtractions next, and other operations at other prearranged times. Operations placed in parentheses are performed ahead of all others. In reverse Polish, however, we

Integer division—A form of division in which only the integer (whole) portion of the result is retained, and any remainder, or fraction, is discarded.

Integer quotient—The result of an integer division operation.

perform operations in the order in which we find the operators—and you cannot change that order by using parentheses. If an addition precedes a division, then we perform the addition first. In a sense, this is an advantage of reverse Polish. We do not have to remember a complicated—and sometimes seemingly arbitrary—operator precedence scheme. We can just take the operators as they come.

Suggested Projects

1. Write the following arithmetic expressions in reverse Polish notation:

```
45 + 532
7 / 4 + 8
-8 - 4 - 5
(384 + 9 / 6) * 19
```

NOTE: There are no "answers" to any of the suggested projects in this book. If you are unsure of how to proceed with a particular question, review the chapter again. If a particular program you write works when you run it on your computer, then the program is "correct." (There is usually more than one way to write a program correctly in a computer programming language.)

2. Write the following reverse Polish expressions in infix notation:

```
9 90 +
7 8 - 19 /
900 89 * 89 + 9 +
```

3. For BASIC programmers only: Translate the following BASIC statement into Forth:

```
PRINT 7 * (1 + 5)
```

4. You are a long-distance runner taking laps around a city block. You know that the distance around the block is 657 meters. You check your pedometer and find that you have run 9,633 meters. Write a Forth expression that will calculate how many times you have run *completely* around the track. (No fractions, please.) Write a second expression that will calculate how many meters you have run on your most recent lap. (Hint: The second expression will require the MOD operator.)

2 HOW FORTH STACKS UP

A programming language interpreter—the program that translates a high-level program into machine language while the high-level program is running—is like a magician. It performs a lot of fancy tricks, but it does most of its work out of sight of the audience, so that nobody can see the actual mechanics of the operation, only the results.

The fact is, with most high-level languages, we are better off not seeing the mechanics. In BASIC, for instance, we can simply pretend that writing the statement PRINT "HELLO" magically causes the word HELLO to appear on the video display, and not worry that the BASIC interpreter is huffing and puffing to do the work of getting that word onto the screen. If the BASIC interpreter is doing its job correctly, we shouldn't have to know the details.

This isn't true in Forth, however. In Forth, it is very helpful to have some idea of how the Forth interpreter works. We don't have to know every trivial detail, but at least we should understand the broad outline. For the trouble of understanding these things, Forth will give us greater control over the workings of our computer than any other language except for machine language itself. And the key to the Forth interpreter is . . . the *word.*

Word—An instruction in the Forth language.

In BASIC, a word that represents an instruction to the computer is called a command, or statement. In Forth, a word that represents an instruction to the computer is called, simply, a word.

There are many different words in Forth, and different versions of Forth may offer different sets of words. Fortunately, there is a standard set of words that is used by many versions of Forth, and this is the set that we will use in this book. In fact, there are at least two different standard sets of Forth words. One set is known as Forth 79, the other as FIG Forth. (The latter is so named because it was created by the *F*orth *I*nterest *G*roup.) In this book, we will try, as much as possible, to follow both of these standards. Fortunately, the differences between the two are not great.

The list of words understood by a given version of Forth is called the Forth *dictionary*. This "dictionary" is more than just an abstract concept. When you are programming in Forth, an actual Forth dictionary, complete with names and definitions of Forth words, must reside in your computer's memory. The Forth interpreter uses this dictionary to look up the meaning of the words in a Forth program. Therefore, in order to program in Forth, you must first load a Forth dictionary into the memory of your computer. This dictionary will probably come on a disk, though there are several versions of Forth that come as cartridges that can be plugged into your computer. (Only a few versions of Forth are available on cassette tape.) The manual that comes with your Forth disk or cartridge will tell you how to load the Forth dictionary into your computer's memory, and how to start using it once you have done this. Actually, you will probably load the dictionary as part of a package of programs that will also include the Forth compiler and interpreter, and perhaps the Forth editor. We will learn shortly what all of these elements of the Forth system do. (Many versions of Forth will also include an assembler,

Command (statement)—An instruction to the computer.

Dictionary—A list of all currently recognized Forth words, and their definitions, maintained by the Forth system.

which allows machine language to be incorporated directly into a Forth program.)

What is the difference between a Forth word and a word (or its equivalent) in a more conventional programming language? Primarily, it is in the way that the Forth word interacts with the rest of the program. In a language such as BASIC, words must usually be used in a specified syntactic arrangement with other parts of the program. That is, when we see a given word of the language, we know that it must appear in conjunction with certain other elements, or something is wrong. For instance, when we use the BASIC word FOR, we know that it must be followed by the name of a variable, which in turn will be followed by an equals sign, which in turn will be followed by an arithmetic expression, which will be followed by the word TO, which will be followed by another arithmetic expression that may or may not be followed by the word STEP and yet another expression. If any of these elements, save the STEP clause, is missing, we're in trouble. The program won't work. The interpreter will grumpily inform you that you have committed a SYNTAX ERROR.

Forth words aren't like that. Sure, they must be used in conjunction with other information. If we set up a program loop, for instance, to have the program repeat certain sequences of instructions, the loop word or words must know how many repetitions are desired or under what conditions the loop is supposed to end. But the way in which we specify this information isn't as rigid in Forth as in BASIC. There are lots of different ways of doing it.

In fact, we can look at Forth words as though they were independent entities, with lives and personalities of their own. Or, if this image is less than helpful, you can imagine each Forth word as a little black box with slots in it. You put information into some of these slots, and it comes back out other slots in a new form. And, if we write our program correctly, one Forth word can put information into the slots of another word. Most Forth words expect to receive information to be processed, and many will, in turn, output information, which can, in turn, be processed by other words. In this sense, a Forth program is less a single functioning entity than a collection of entities—words—acting together in mutual cooperation.

In BASIC, we can give instructions to the computer either in the immediate mode or in the programming mode—that is, we may ask the computer to perform the instructions immediately or store the instructions in its memory for later execution. In Forth, we are offered a similar choice.

When you first load your Forth system—that is, the Forth dictionary, compiler, and interpreter—a cursor will appear on the screen. This indicates that the computer is waiting for further instruction. It is waiting, in fact, for you to type an instruction or instructions in Forth. Whatever instructions you type will be executed as soon as you press the key marked RETURN (or ENTER) on the keyboard.

We type a series of Forth instructions just as we would type instructions in the BASIC immediate mode, by using the computer's keyboard as though it were a typewriter keyboard. If you make a mistake in typing, there will be a key that you may press to backspace the cursor and delete your error. On most systems, this key is called DEL, DELETE, or RUB, though on a few it may simply be represented by a leftward-pointing arrow. A typical Forth system will allow us to type up to eighty characters worth of Forth instructions in the immediate mode. When we have finished typing our instructions, we let the computer know that we have finished by pressing RETURN.

So that you can get the hang of typing Forth instructions, load your Forth system and type the following, pressing RETURN at the end of the line:

```
8 4 + 3 * 6 / .
```

As you should be able to tell by now, this will add 4 to 8, multiply the result of the addition by 3, divide the result of the multiplication by 6, and display the result. When you have typed this and pressed RETURN, you should see the following on the video display:

```
8 4 + 3 * 6 / . 6 OK
```

The result—6—is printed directly after the end of the instruction, immediately after the point at which you pushed the RETURN key. If you had wanted the result printed on the following line, you could have added the Forth word CR directly in front of the expression. This simply tells the Forth system to print a carriage return (move the cursor down to the beginning of the next line) on the display. The sequence after we had pressed RETURN would then look like this:

```
CR 8 4 + 3 * 6 / .
6 OK
```

Note that a carriage return was performed before the result of the expression was printed.

What we see happening on the screen at this point, however, is not the whole story. As we said earlier, it is important that we have some understanding of what the Forth interpreter actually does during the execution of a Forth word or series of words—that is, between the moment when you press RETURN and the moment the word OK appears on the display.

When you press the RETURN key, a specific sequence of events is set into motion. The Forth interpreter begins to read what you have typed. Since Forth programs are made up of only two possible elements, numbers and/or words, the interpreter sets out to determine whether what it sees is a number or a word. If what it sees is a word, it will look it up in the Forth dictionary, to find out what that word means. If what it sees is a number, the interpreter will execute a special routine that handles numbers.

In this example, the first thing the interpreter sees is CR. Since CR is, in fact, a word, the Forth interpreter—which is simply a program in the Forth system that monitors the execution of Forth words—looks up the word CR in its dictionary. When it finds it, the interpreter executes a routine that performs the action represented by the word CR—that is, a routine that prints a carriage return. This routine is, in fact, the "definition" of CR and is part of the dictionary. You might wonder what language this routine is written in. The answer is that it may be written in machine language, but it may also be written in Forth. Oddly, at least three-fourths of your Forth system is probably written not in machine language but in Forth itself. The interpreter, of course, is written in machine language.

Now the interpreter turns its attention to the next thing that we have typed, which happens to be the number 8. In

Forth, a number is considered every bit as much of an instruction as any word in the Forth dictionary. (In fact, some of the more commonly used numbers, such as 0 and 1, are actually allotted entries in the Forth dictionary, making them full-fledged Forth words.) When the Forth interpreter sees a number, that number is automatically placed in a portion of the computer's memory called the *stack.*

What is the stack? The stack is the way in which Forth words get information—the slot in the black box, as it were—and the way in which Forth words pass information between themselves. It is a concept that must be understood for one to fully appreciate the operation of the Forth interpreter.

It's not necessary to understand how a computer's memory works in order to understand the stack. The stack is simply a place where we store numbers (even numbers representing letters or other symbols). Of course, all computer memory is used to store numbers. What makes the stack unusual is the way in which those numbers are stored—and the way in which they are retrieved.

Perhaps the best way to understand the stack is by analogy. Picture a spring-activated plate holder, the kind you sometimes find in restaurants, often near the salad bar. Plates are put into the holder from the top. Once inside, they are supported by a spring that presses up from the bottom, holding the plates near the top where they can be easily reached. As more plates are put into the holder, however, the weight of the plates forces the spring downward, so that the top plate is always precisely even with the top of the holder, no matter how many plates are stacked underneath it. New plates are always put into the holder from the top—and always removed from the top when they are needed. Thus, the plate most recently put into the holder is always the first one removed, and the first one removed is

Stack—A specific storage location within the computer's memory, maintained by the Forth system, where items can be stored one at a time, but where only the most recently stored (but not yet removed) item may be removed at any time.

always the one most recently put in. (Of course, we may lift off the top plate, grab the one underneath it, and put the top plate back on. But this takes an extra effort.)

The microcomputer stack operates in similar fashion, except that it stores numbers rather than plates. We can put as many numbers in the stack as we wish, within the limits of available memory, but when we take a number out of the stack, we can take only the one off the "top"—that is, the one most recently put in (or at least the one most recently put in that has not yet been removed). Computer programmers call this a LIFO structure: Last In, First Out. We can imagine the stack as a pile of numbers, one on top of another, with only the top number easily accessible to us as programmers. (Of course, if we pull a number off the stack, the number directly underneath it becomes the top number on the stack, and thus is also available to us. And there are Forth words, as we shall later see, that let us slip older numbers off the stack by clever skulduggery, just as we can sneak a plate out of the middle of the plate holder.)

What good does the stack do us? Plenty.

When the Forth interpreter sees that we have typed a number, it *pushes* that number onto the stack. That number will remain on the stack until the interpreter has reason to remove it. However, if more numbers are pushed on top of this number, all of the numbers must generally be removed in the reverse order from which they were put in.

Let's watch the Forth stack in action. Here is the program line we typed earlier in this chapter:

```
CR 8 4 + 3 * 6 / .
```

We've already seen what happens when the interpreter sees the CR. Now we know that when the Forth interpreter sees the number 8, it pushes it onto the stack, like this:

```
8 ← bottom of the stack
```

Then, when it encounters the number 4, it will push it onto the stack, too, like this:

```
4
8 ← bottom of the stack
```

Now we have two numbers on the stack, one on top of the other, both waiting to be retrieved.

When the interpreter encounters the symbol +—that is, the plus sign—it looks it up in the dictionary to see if it is a Forth word. Since it is, the interpreter will execute the + word. The + word expects to find numbers on top of the stack—specifically, two numbers. It takes these two numbers from the top of the stack, adds them together, and puts the result back on top of the stack. (It doesn't matter if there are other numbers on the stack, below these numbers. They will be ignored by the + word, though they can be used by other routines once the numbers on top of them have been removed.)

In our example here, the top numbers on the stack are 4 and 8. Thus, the + routine adds 4 and 8 and pushes the number 12 (which is the sum of 4 and 8) back onto the stack. After this routine executes, the stack will look like this:

12 ← bottom of the stack

This may raise some questions in your mind. What if the + routine had found no numbers on the stack? Or what if they had been the wrong numbers? In the first case—no numbers on the stack—you would have received an error message reading STACK EMPTY! or something similar. In the second—wrong numbers on the stack—you would have received no error message, but the result of the addition would have been meaningless. Therefore, in Forth, it is your duty as a programmer to make sure that the proper numbers are on the stack at the proper time. Since most Forth words expect to find numbers on the stack, and/or leave numbers on the stack, we will provide you with diagrams of many of the new Forth words that we introduce, showing the state in which the words expect to find the stack, and the state in which they will leave the stack. (There are a few Forth words, such as CR, that do not affect the stack and we will inform you when this is the case.)

What sort of diagrams will we use? Well, here's a diagram for the + word:

```
(n1 n2 --- n3)
```

The three dashes (---) in the center represent the Forth word—in this case, +. The symbols preceding the dashes represent the state of the stack before the word is executed by the Forth interpreter. In this case, we see that there should be two numbers on the stack, which we have called n1 and n2. The symbols following the dashes represent the state of the stack after the word is executed. In this case, there is a single number left on the stack, which we have called n3. There may also be other numbers on the stack below these, but they are unaffected by the execution of +.

This fits what we were told, a few moments ago, about the word +. It takes the two numbers from the top of the stack, adds them together, and leaves a single number behind.

Let's return to our example. When we last left the Forth interpreter, it had executed the word +, which had pushed the number 12 on top of the stack, like this:

```
12 ← bottom of the stack
```

As it continues executing our instructions, the interpreter next encounters the number 3. It pushes this number on top of the stack, like this:

```
3
12 ← bottom of the stack
```

Then, when it encounters the word *, it executes the dictionary definition for *. This word takes the top two numbers off the stack, multiplies them together, and leaves the result on the stack. The stack diagram for * looks like this:

```
(n1 n2 --- n3)
```

The stack, after the execution of *, looks like this:

```
36 ← bottom of the stack
```

The number 36, of course, is 12 multiplied by 3. Next, the interpreter pushes the number 6 onto the stack, like this:

```
6
36 ← bottom of the stack
```

The interpreter then sees the word / and executes the definition for /. This word takes the top number off the stack, divides it into the second number on the stack, and leaves the result on the stack. The stack diagram for * looks like this:

```
(n1 n2 --- n3)
```

(You may note that this is the same diagram we have encountered before and is a fairly common stack diagram for Forth words, though we shall see a few diagrams later in this book that look quite different. This particular diagram is the stack diagram for all the arithmetic operators that we have discussed in this book: +, −, *, /, and MOD.)

The stack, after execution of /, looks like this:

```
6 ← bottom of the stack
```

The number 6 is the result of 36 divided by 6.

Finally, the interpreter sees the dot word and executes its definition. This word removes the top number from the stack and prints it on the video display, leaving nothing behind on the stack. Thus, the stack diagram for dot looks like this:

```
(n1 --)
```

And the state of the stack after dot executes is this:

```
bottom of the stack
```

Nothing is left on the stack. It is always a good idea to structure our Forth programs in such a way that the stack is empty when we are done—unless the program is designed to pass information on the stack to another program.

Incidentally, it is not necessary that we type all the instructions in a Forth routine on a single line, pressing RETURN once at the end. We can type instructions on several different lines, pressing RETURN after each. The stack will remain in whatever state it is left at the end of each line.

For instance, if we simply typed 8, then pressed RETURN, the interpreter would place the number 8 on the stack. Then, if we typed the number 4 on a separate line and pressed RETURN again, the number 4 would be placed on top of the number 8. Finally, if we typed the word + and pressed RETURN, it would add 4 and 8, even though they had been placed on the stack on an earlier line. It does not matter *how* a number got onto the stack, as long as it is there when required by the word currently being executed.

The stack is very important in Forth. If you don't understand how it works, it would be a good idea to go back over the material in this chapter until you have a firm grasp of this concept. The entire Forth language revolves around the stack, and we will be talking about it throughout the remainder of this book.

Suggested Projects

1. Write a sequence of Forth words that will multiply 4 by 3 and divide the result by 6.

2. Without looking back to the previous chapter, draw the stack diagram for the arithmetic operator − (subtraction). When subtraction is performed by this word, is the number on top of the stack subtracted from the number underneath it, or vice-versa? (Hint: Consider the order in which the two operands are written when we perform subtraction in Forth.)

3 EXTENDING FORTH

One of the nicest things about Forth—and one of the things that sets it apart most dramatically from other programming languages—is that it allows us to add new words to its dictionary. Just as an English dictionary defines new words in terms of old words, we can define new words in the Forth dictionary in terms of old Forth words in the Forth dictionary (or in machine language, if your system has an assembler). For this reason, we say that Forth is an *extensible language*—a language that can be extended in almost any fashion we desire. A programmer who works frequently with Forth will eventually develop new words that prove particularly useful to him or her; these words may be saved on a disk and used in future Forth programs; they can even become a permanent part of the programmer's Forth dictionary. Thus, a Forth system grows with its user; the language changes and evolves with time, almost like a living organism. In effect, Forth allows you to develop your own custom computer language.

Extensible language—A computer programming language that allows new words and features to be added by the programmer.

This feature dictates the nature of Forth programs. Most consist of a series of word definitions. First, we create and then define a few simple new words using existing Forth words; then we create and define more complex words using our new words, until at last we have created a single word that is, by its definition, our program.

How do you create and define a new word in Forth? You use the Forth word : (pronounced *colon*). The colon word signals the interpreter that what follows is a new Forth word, and that what follows the new Forth word is the definition of the word. The definition must be terminated with the word ; (pronounced *semicolon*).

Suppose, for instance, that we wanted to create a new Forth word that simply added the numbers 1 and 2 and printed the result on the video display. Not a very useful word, admittedly, but nonetheless one that will provide an example of how to create and define a word in Forth. We could call our new word ONE-PLUS-TWO and define it like this:

```
: ONE-PLUS-TWO 1 2 + . ;
```

The colon tells the interpreter that ONE-PLUS-TWO is a new word that we intend to define. The rules for naming a new word in Forth are simple. You may use any name, of any length (up to a reasonable limit, defined by your particular system), and with any characters you desire in it. However, if you use the name of a word that is already in the Forth dictionary, the new word will replace the old one, which may not be the effect you intended. If this happens, most Forth interpreters will inform you. For instance, if you attempt to create a Forth word called, say, KEY (which is already the name of a standard Forth word), the computer will respond with KEY IS NOT UNIQUE or REDEFINITION: KEY or some similar message. If you do not wish to redefine the old word, you must immediately tell the computer to forget the new definition. This is done by typing the standard Forth word FORGET, followed by the Forth word that you wish to eliminate—for instance, FORGET KEY. Once the new definition of KEY is removed from the dictionary, the old definition—which was never actually dropped from the dictionary physically—will be reinstated. Note, however, that typing FORGET KEY will not only cause Forth to forget the word KEY, but also all words defined since you last defined KEY. And, since it forgets only the most recent definition of KEY, you must type FORGET KEY once for every definition of KEY that you wish to forget. (Of course, if you accidentally redefine the word FORGET, you're in trouble.)

Also note that some Forth systems only put the first three letters of a word into the dictionary, plus a notation as to how many characters are in the word. Thus, if your new word has the same first three characters and the same length as an existing Forth word, you may inadvertently replace the definition of the earlier word, because Forth will be unable to tell the words apart. On the other hand, many Forth systems compile the complete word into the dictionary, though there may still be a limit to the longest word that can be compiled.

The series of characters following the name of our new word, ONE-PLUS-TWO, constitutes the "definition" of the word. In this case, our new word is defined like this:

```
1 2 + .
```

This is the series of instructions that will now be executed by the interpreter every time it encounters the word ONE-PLUS-TWO. This series of instructions will be entered by your Forth compiler into your Forth dictionary for future reference. Whenever the interpreter sees the colon word, it will call the compiler and tell it, in essence, that there is a definition waiting to be compiled. The compiler not only stores the definition in the dictionary, it also translates the definition into a special form that can be executed more quickly than a series of words typed in the immediate mode. This special form is called *threaded code.* (Remember that we earlier referred to Forth as a threaded language.)

Threaded code—A method of representing computer programs within a computer's memory that combines aspects of interpreted and compiled languages.

Now, if we type the word ONE-PLUS-TWO in the immediate mode, like this:

```
ONE-PLUS-TWO
```

the interpreter will respond by printing the number 3 (which is 1 + 2) on the video display.

You'll notice that the instructions are not executed when we type the definition, but only when we type the word itself, without a colon in front of it. The colon tells the interpreter that these instructions are to be compiled by the compiler and stored in the computer's memory, for future reference. This is reminiscent of the way in which a BASIC interpreter responds when we precede a series of BASIC instructions with a line number. The program line is stored in the computer's memory, until we call it up for execution.

Sometimes it is useful to create new Forth words that expect to find values already on the stack. Suppose, for instance, that we wanted to create a word that would add 4 to the topmost value on the stack, and leave the result on the stack. We could define such a word like this:

```
: ADD-FOUR 4 + ;
```

Note that we have preceded the word + with a 4. This expression, when executed, will cause the number 4 to be pushed onto the stack. The word +, however, expects to find *two* numbers on the stack. Thus, when we execute this word, it will be necessary to push another number onto the stack first, like this:

```
7 ADD-FOUR .
```

The effect of this series of instructions will be to add 4 to the number 7 and print the result on the video display. It is just as though we had written this series of instructions:

```
7 4 + .
```

except that two of those instructions have now been given a name and turned into a Forth word called ADD-FOUR.

When we program in Forth, our program will consist largely of definitions of new words. When we create new words in Forth, we may then create even newer words in terms of those new words—that is, we may use the new words in colon definitions, just like any other Forth words. In fact, the Forth interpreter and compiler accord our new words all the privileges and respect of the words that come with the system. (Most Forth systems also come with a built-in assembler—a program that will allow you to write machine-language programs—so that advanced programmers may write definitions of new words in machine language instead of Forth. However, we will not discuss the assembler in this book, and you are not expected to be able to use it. Most Forth programmers are quite content to define new words using the Forth language. The assembler is mainly for programs that require extremely high execution speeds, or that need to interact with the computer hardware in a manner no existing Forth word allows.)

Forth is full of useful words that we may include in the definitions of new words. Many of these words allow us to manipulate the contents of the stack. For instance, the word DUP removes the word on top of the stack, duplicates it, and pushes two copies of it back onto the stack.

To illustrate how DUP works, let's throw some numbers on the stack and examine them. For instance, we can write:

```
5 .
```

By now you know that this will cause the number 5 to be printed on the display. The interpreter pushes the 5 on top of the stack. Then the dot word takes it back off the stack and prints it. To verify that there is nothing left on the stack, type a dot (.) and press RETURN. You should receive an error message telling you that nothing more remains on the stack.

Now, let's write the following:

```
5 DUP .
```

This, too, prints the number 5 on the display. However, if the word DUP duplicates the contents of the stack, we may rightfully suspect that something is still left on the stack. Sure enough, if we type:

and press RETURN, it will print another 5 on the display. The original 5 was duplicated by DUP, and it required two dots to print both of them.

Here is the stack diagram for DUP:

```
(n1 --- n1 n1)
```

What would we need a word like DUP for? Well, suppose we wanted to create a word that would automatically square a number—that is, multiply a number by itself. There is no such word in the standard Forth dictionary, but we can create one with the aid of DUP, like this:

```
: SQUARE DUP * ;
```

Now, if we write:

```
4 SQUARE .
```

the computer will print the number 16 on the display.

Let's run through the last operation to make absolutely sure we understand how it works. When the interpreter finds that we have typed the number 4, it pushes it onto the stack, so that the stack looks like this:

4 ← bottom of the stack

Then, when the interpreter encounters the word SQUARE, it looks it up in the dictionary and finds that it is defined as:

```
DUP *
```

The interpreter first executes the DUP word, which duplicates the top item on the stack. The stack now looks like this:

```
                    4
                    4 ← bottom of the stack
```

The interpreter then activates the * word, which takes the top two numbers of the stack and multiplies them by one another. After executing DUP, both of the numbers on top of the stack are 4s; thus, 4 is multiplied by itself and a result of 16 is pushed back on the stack.

Now that the interpreter has reached the end of the definition of SQUARE, it returns to the line of instructions that we have typed and determines that one more instruction, the dot word, remains. The interpreter activates the dot routine, which prints the contents of the stack—the number 16—on the display.

Our new word, SQUARE, may also be used to find the square of other numbers. For instance, if we wrote:

```
6 SQUARE .
```

the number 36 would appear on the video display.

The DUP word is also useful when we wish to maintain the number on top of the stack through several operations. You may have noticed that many of the words that we have discussed so far tend to destroy the original contents of the stack in the course of performing their defined tasks. If we wish to maintain a particular number on top of the stack, we must duplicate it before performing an operation that would destroy it. For instance, suppose we want to print a number on the video display, then square it with the SQUARE word and print the result. We could print the original number with the dot word, but this has the side effect of destroying the number on top of the stack in the course of printing the number. In other words, the number that we placed on top of the stack will no longer be there after dot executes. To get around this, we could push the number onto the stack twice, but this is not a very elegant solution and only works in situations where we know in advance what number is to be squared. (This will not always be the case.) A better solution would be to duplicate the number on the stack, like this:

```
3 DUP . SQUARE .
```

This series of words prints the number 3 on the display followed by the number 9, like this:

```
3 9
```

Let's follow through this series of instructions, determining the state of the stack after each. After the number 3 is pushed onto the stack, the stack looks like this:

```
3 ← bottom of the stack
```

After DUP executes, the stack looks like this:

```
3
3 ← bottom of the stack
```

The dot word prints the top number from the stack on the display, leaving the stack like this:

```
3 ← bottom of the stack
```

The word SQUARE then squares the number on top of the stack, pushing the result—9—back on the stack, where it is printed by the next dot word.

Another Forth word that manipulates the stack is SWAP. As the name implies, it "swaps," or exchanges, the topmost two items on the stack, bringing the second item on the stack to the top. Thus, if the stack looks like this:

```
14
27 ← bottom of the stack
```

SWAP will leave it looking like this:

```
27
14 ← bottom of the stack
```

The stack diagram for SWAP looks like this:

```
(n1 n2 --- n2 n1)
```

SWAP is useful because, although we will often pass numbers from one routine to another via the stack, the numbers will not always be in the proper order. The number required by a given routine may not be on top of the stack but, rather, submerged underneath other numbers. SWAP helps us bring such numbers to the top.

For instance, suppose we wish to perform a series of operations on the number 12, while still maintaining the results of those operations on top of the stack. We must duplicate the number before every operation, so that it will not be destroyed before it can be passed on to the next operation, but we must also swap the number back to the top of the stack after every operation is performed, else it will be buried under the results of the operation.

The series of operations might look like this:

```
12 DUP 4 * SWAP DUP 6 / SWAP DUP 7 + SWAP SQUARE
```

The above places 12 on the stack, duplicates it (for subsequent use), multiplies it by 4 and pushes the result on top of the stack, swaps the duplicate 12 back to the top of the stack, duplicates it again, divides it by 6, swaps 12 back to the top, duplicates it yet again, adds 7 to it, swaps 12 back to the top, and squares it.

When the operations are all finished, the results are all left on the stack, one above the other. Note that the results will be on the stack in the opposite order from that in which the operations were performed; that is, the result of the SQUARE operation will be on top of the stack, the result of the + operation will be directly underneath that, the result of the division under that, and the result of the multiplication will be on the very bottom. The number 12 will no longer be on the stack at the end because we did not duplicate it before the final operation.

Trace through this sequence of operations, determining what condition the stack is in after each word. See if you understand why it is necessary to use the word SWAP after each operation except the last.

If we wish to print the results of our operations on the display, we will need to use a series of dots, like this:

This will print the results out all in a row, like this:

```
144 19 2 48
```

If we wish to print the results on separate lines, we must place carriage returns between the dots, like this:

```
. CR . CR . CR .
```

This will print out the results like this:

```
144
19
2
48
```

If we *really* want to get fancy, we can annotate the results, like this:

```
." 12 SQUARED EQUALS " . CR
." 12 PLUS 7 EQUALS " . CR
." 12 DIVIDED BY 6 EQUALS " . CR
." 12 TIMES 4 EQUALS " . CR
```

The Forth word

```
."
```

is pronounced *dot-quote.* It is used to print a message on the video display. The message must follow the word and begin and end with quotation marks. Note that when we use dot-quote, we must always follow it with at least one space. This space is not considered part of the message to be printed. If we do not include this space, it will confuse the interpeter, which will think that the message following the quote is part of the word itself. For instance, if we typed

```
."HELLO, THERE"
```

the interpreter would think we were using a word called

```
."HELLO,
```

It would look this word up in the dictionary and would probably not find it there, thus causing a program error.

The above series of instructions will print our four results from the stack along with a message describing each. The result of executing these commands will be this display:

```
12 SQUARED EQUALS 144
12 PLUS 7 EQUALS 19
12 DIVIDED BY 6 EQUALS 2
12 TIMES 4 EQUALS 48
```

Actually, if you tried to type the instructions above to achieve this result, you may have run into a slight problem. Many Forth systems limit a single line of input to eighty characters or less. Unfortunately, this series of instructions takes up more than eighty characters; thus, we cannot type them all at once and therefore cannot see this display.

This is a disadvantage of working in the immediate mode (along with the inability to retrieve our instructions at a later time). In a later chapter, we will see that there is an alternative to writing instructions and definitions in the immediate mode. For the moment, however, we can get around this limitation by combining all of these commands into a single word, or into a series of words. For instance, we could create four new words, with their definitions following, like this:

```
: SQUARE-MS ." 12 SQUARED IS " . CR ;
: ADDITION-MS ." 12 PLUS 7 IS " . CR ;
: DIVISION-MS ." 12 DIVIDED BY 6 IS " . CR ;
: MULTIPLICATION-MS . " 12 MULTIPLIED BY 4 IS " . CR ;
```

The letters MS, as used here, are short for MESSAGE, because each of these words prints a message on the display.

Now we can write these instructions in a single eighty-character line, like this:

```
SQUARE-MS ADDITION-MS DIVISION-MS MULTIPLICATION-
MS
```

Even better, we can combine all these words into a single word, PRINT-RESULTS, like this:

```
: PRINT-RESULTS SQUARE-MS ADDITION-MS DIVISION-MS
MULTIPLICATION-MS ;
```

We can also combine the instructions in all the arithmetic operations performed on the number 12 so that they can be represented by a single new word, PERFORM-MATH. We do it like this:

```
: PERFORM-MATH 12 DUP 4 * SWAP DUP 6 / SWAP DUP 7 +
SWAP SQUARE
```

Now we can write the entire series of operations as a pair of words:

```
PERFORM-MATH   PRINT-RESULTS;
```

And, if we wish, we can go whole hog and combine these operations, in turn, into the single word, DO-IT, like this:

```
: DO-IT   CR PERFORM-MATH   PRINT-RESULTS
```

Now all we need do is type DO-IT in the immediate mode to produce this display:

```
DO-IT
12 SQUARED EQUALS 144
12 PLUS 7 EQUALS 19
12 DIVIDED BY 6 EQUALS 2
12 TIMES 4 EQUALS 48
OK
```

We have now created a complete Forth program. This, in fact, is the way that most Forth programs are constructed. First, we create and define a set of simple new words, then, using those simple new words, we create and define more and more complex new words, until finally we have a single word that contains our entire program within it. And all we need do to execute the program is to type that single word, in the immediate mode.

Admittedly, this DO-IT program is not terribly useful, but we shall be looking at some considerably more practical applications for Forth in the next few chapters.

* * * * * * * *

Before we go on, however, let's take a look at some other standard Forth words that perform useful stack manipulations.

The word ROT (short for *rotate*) will move the third word on the stack to the top. If the stack looks like this:

```
5
4
1 ← bottom of the stack
```

the word ROT will rearrange it like this:

```
1
5
4 ← bottom of the stack
```

The stack diagram for ROT looks like this:

```
(n1 n2 n3 --- n2 n3 n1)
```

The Forth word OVER duplicates the second value from the top of the stack, leaving a copy of that value on top of the stack. Thus, if the stack looks like this:

```
 6
27 ← bottom of the stack
```

the word OVER will leave it like this:

```
27
 6
27 ← bottom of the stack
```

The stack diagram for OVER looks like this:

```
(n1 n2 --- n1 n2 n1)
```

The word PICK allows us to take a value from anywhere on the stack and copy it to the top. Before executing PICK, however, we must push a number on top of the stack that indicates which item on the stack we wish to duplicate. This number will not be taken into account when calculating how far down the stack the item to duplicate is. For instance, if we write 1 PICK, the number directly below the first will be duplicated to the top of the stack. (1 PICK is the exact equivalent of the DUP word.) If we write 2 PICK, the number that is located two positions below the first will be duplicated to the top of the stack. (This is equivalent to OVER.)

If the stack looks like this:

```
 56
  9
 22
 78
  3
678 ← bottom of the stack
```

and we write 4 PICK, the stack will then look like this:

```
 78
 56
  9
 22
 78
  3
678  bottom of the stack
```

Finally, the Forth word DROP performs one of the most useful stack functions of all. It removes the top number from the stack and does nothing with it. Often, in the course of a Forth program, we will find ourselves with numbers on the stack that are no longer needed. DROP allows us to get rid of them.

If the stack looks like this:

```
 7
18 ← bottom of the stack
```

it will look like this after the execution of DROP:

```
18 ← bottom of the stack
```

[40]

After a second execution of DROP, it will look like this:

bottom of the stack

The stack diagram for DROP is simple:

```
(n1 --- )
```

Suggested Projects

1. Create a new Forth word called CUBE that will raise the number on top of the stack to the third power—that is, multiply the number by itself two times.

2. Create a Forth word that takes a number off the stack, performs several arithmetic operations on it, and prints the result of each operation on the screen with appropriate annotations. Do your definition in steps, first creating simple words that perform part of the task, then creating a larger word that performs all of it.

3. Create a Forth word that takes three numbers off the top of the stack and puts them back on again in reverse order. Create a Forth word that does the same thing with four numbers.

4 CONTROL WORDS

In computer programming, it is not enough simply to perform arithmetic on numbers, to print numbers and messages on the computer's video display, or even to bounce numbers around on the stack. We must be able to create sequences of instructions and give the computer the ability to execute those instructions selectively. That is, the computer must be able to make decisions.

Every computer language in existence offers a set of instructions that allows the computer to choose between two or more alternative program paths. In Forth, we refer to these instructions as *control words*. Perhaps the most valuable of these control words is IF.

Those of you who have programmed in other computer languages will certainly recognize IF; this particular word is used in almost every computer language ever developed. And it is used in Forth in much the same fashion as it is used in other languages, including BASIC, except (of course) that it is used in a reverse Polish manner— and it expects to receive information from the stack.

Control word—A Forth word that affects the sequence in which instructions are executed within a Forth program.

Before we can use IF, though, we must be able to test the relationships between numeric values. The IF word can then choose between alternative program possibilities based on the results of these tests. There are three words in Forth, called *relational operators,* that test numeric relationships:

< less than
> greater than
= equal to

Although these resemble the relational operators used in other programming languages, they are in fact full-fledged Forth words, with definitions in the Forth dictionary. Each follows the same stack diagram:

```
(n1 n2 --- f)
```

The letter f in this stack diagram stands for *flag.* A flag is simply a number that represents the outcome of a test performed by a relational operator.

The < word tests to see if n1 is less than n2. If it is, < places a 1 on the stack. The 1 is a flag indicating that the outcome of the test was "true." If n1 is *not* less than n2—that is, if it is greater than or equal to n2—< places a 0 on the stack. The 0 is a flag indicating that the outcome of the test was "false."

Similarly, the > word tests to see if n1 is greater than n2, and places a 1 or a 0 on the stack accordingly. The = word tests to see if n1 is equal to n2 and places an appropriate flag on the stack.

Relational operators—Operators that test the relationships between numeric values, such as "greater than" (>), "less than" (<), "equal to" (=), and so on.

Flag—A value that indicates the outcome of a previous operation, such as a comparison of two numeric values. The value 0 is generally used to represent a false (or unsuccessful) outcome and the value 1 (or any other non-zero value) a true (or successful) outcome.

The IF word expects to find a flag on top of the stack—that is, it removes the number on the top of the stack and treats it as though it were a flag, whether it was actually intended as one or not. The IF word is not terribly choosy about what it will accept as a flag. Any non-zero number will be interpreted as a "true" flag—that is, a flag indicating that the result of the last comparison was true. Only 0, however, will be accepted as a "false" flag, a flag indicating that the result of the last comparison was false.

To demonstrate how IF works, let's create a word that will test to see if the top number on the stack is 5 and will print an appropriate message. Such a word and its definition might look like this:

```
: IS-IT-FIVE? 5 = IF ." YES, IT'S A FIVE" ELSE . " NO, IT'S
NOT A FIVE" ENDIF CR ." TEST COMPLETE" ;
```

This creates and defines a word called IS-IT-FIVE? Because this is a complex statement, and because it introduces a number of new concepts, we will analyze it word by word.

In this definition, we first place a 5 on the stack and test to see if it is equal to the number that was previously on top of the stack. This test is performed with the = word. If the number on the stack is equal to 5, the = word places a 1 (that is, a "true" flag) on the stack. If the top number on the stack is not equal to 5, = places a 0 ("false" flag) on the stack.

The IF word takes the flag back off the stack. If the flag is true—that is, non-zero—IF tells the interpreter to start executing the instructions that immediately follow the IF. The interpreter will continue executing these instructions until it encounters the word ELSE. Then it will skip ahead to the next instruction after the word ENDIF, which in this case is:

```
." TEST COMPLETE"
```

(The word ENDIF will be replaced by THEN on many Forth systems.)

On the other hand, if the flag is false—that is, 0—IF tells the interpreter to skip ahead and start executing the instructions following the word ELSE.

We may use IS-IT-FIVE? like this:

```
89 IS-IT-FIVE?
```

This would cause the words NO, IT'S NOT A FIVE to appear on the display, followed by TEST COMPLETE on the next line. On the other hand, if we typed this:

```
5 IS-IT-FIVE?
```

it would cause the words YES, IT'S A FIVE to appear on the display.

IS-IT-FIVE? is roughly equivalent to this statement in BASIC:

```
10 IF X=5 THEN PRINT "YES, IT'S A FIVE" ELSE PRINT "NO,
IT'S NOT A FIVE"
20 PRINT "TEST COMPLETE"
```

where X is equivalent to the number on top of the stack in the Forth version.

In Forth, every IF must be followed by an ENDIF (or a THEN). The ENDIF indicates where the IF statement terminates. Instructions following the word ENDIF are not affected by the IF and will execute regardless of what flags are on the stack.

The ELSE part of the IF statement is optional. If there is no ELSE, then everything between the IF and ENDIF will be skipped if the flag is false. When you use the ELSE, it is perfectly permissible to follow it with another If, even before the ENDIF. However, you must remember to include an ENDIF for every IF in a sequence of IF statements. All of these ENDIFs will usually be bunched together at the very end of the IF statement. For instance, here is a variation on IS-IT-FIVE? that will also inform us if the number on the stack is larger or smaller than 5:

```
: IS-IT-FIVE? DUP 5 = IF ." IT'S A FIVE" ELSE DUP 5 < IF ."
IT'S LESS THAN FIVE" ELSE." IT'S GREATER THAN FIVE"
ENDIF ENDIF ." TEST COMPLETE";
```

Note that we must duplicate the number on top of the stack before performing the first two steps, so that a copy of the number will be available for succeeding tests. Note also the two ENDIFs at the end of the statement, indicating the simultaneous termination of *both* IFs. The > word is used to test if the number on the stack is greater than 5. The < word is not used, because it can be safely assumed that if the number on the stack is neither equal to nor greater than 5, then it is certainly less than 5.

We can use this new version of IS-IT-FIVE? in the same fashion as the old one:

```
3 IS-IT-FIVE?
```

This will print IT'S LESS THAN FIVE, then TEST COMPLETE. And this:

```
7 IS-IT-FIVE?
```

will print IT'S GREATER THAN FIVE.

Since the word IF will accept any non-zero value as a "true" flag, it is not absolutely necessary to perform a test of numeric relationships before the IF is executed. For instance, it is never absolutely necessary to use a relational operator to see if the number on top of the stack is equal to 0; the IF word can perform this test on its own. Here, to demonstrate this point, is a word that tests for a value of 0 on the stack.

```
: IS-IT-ZERO? IF ." THAT'S NOT A ZERO" ELSE ." THAT'S
A ZERO" ENDIF ;
```

Do you see how this works? If there is a 0 on the stack, IF will treat it as a "false" flag, and the ELSE portion of the statement will execute. If there is a non-zero number on the stack, IF will mistake it for a "true" flag, and the first part of the statement will execute. There is no functional reason why we must use a relational operator to test for the zero value.

Another type of control word is the *loop word.* This is a

Loop word—A Forth word that causes a sequence of program instructions to execute more than once.

word, or actually a set of words, that causes a series of instructions to repeat over and over again, either for a specified number of times or as long as a specified condition is or is not true. There are several loop words in Forth.

BASIC programmers will be familiar with the BASIC words FOR and NEXT, which create a program loop that will execute a fixed number of times. The equivalent words in Forth are DO and LOOP. These words may not be used in the Forth immediate mode, only in a colon definition. They delimit the boundaries of the loop, with the word DO appearing at the beginning of the sequence of instructions that you wish to repeat and the word LOOP appearing at the end. Before the loop executes, we must arrange for two numbers to be on the stack, representing the upper and lower limits of the loop. For instance, here is a word that will print the numbers 0 to 9 on the video display:

```
: COUNT'EM 10 0 DO I . CR LOOP ;
```

This is equivalent to the BASIC program FOR I=0 TO 10 : PRINT I : NEXT I. The numbers 10 and 0, which we place on top of the stack before the DO word executes, are the *limit* of the loop and the *index* of the loop, respectively. When the loop begins, the 0 will be placed on the *return stack,* a location in the computer's memory similar (but not the same as) the stack on which we have been passing data between words. (This latter stack is sometimes referred to as the *parameter,* or *data, stack.*) Each time the end of the

Limit—The value in a DO loop at which the loop ends.

Index—The value in a DO loop that "counts" the number of times the loop executes.

Return stack—The location in the computer's memory where the index of a DO loop is stored.

Parameter (data) stack—The location in the computer's memory where values are passed from one Forth word to another.

loop is reached, the value on the return stack is increased by 1. If it has not yet exceeded the limit value—10, in this instance—control of the program is returned to the instruction immediately following the word DO. If it has exceeded the value of the limit, the value is removed from the return stack and control of the program passes through to the instruction immediately after the word LOOP.

The word I in the above example copies the value on top of the return stack to the top of the parameter stack. (Note that it does not remove the value from the return stack.) Once we have copied this value to the parameter stack, we are able to print it out with the dot word.

You can see the entire process in action by typing the above definition and then executing COUNT'EM in the immediate mode. When you do so, you will see the following display:

```
0
1
2
3
4
5
6
7
8
9
```

Because the loop will cease to execute the moment that the index reaches the limit value, you will notice that the above program never has a chance to print the number 10 on the screen. However, you should note that the loop does execute ten times in all, printing ten numbers on the display.

It is not even necessary that we count by ones. Forth offers a second way of ending DO-LOOPs—the Forth word +LOOP (pronounced *plus-loop*)—that allows us to specify the increment by which we wish to count. The stack diagram for +LOOP looks like this:

```
(n1 ---)
```

The +LOOP word removes this number from the parameter stack and adds it to the current count on the return stack. Thus, if we wished to count only even numbers from 100 to 200, we could write:

```
: EVEN-COUNT 200 100 DO I . 2 +LOOP ;
```

The 2 on the stack before +LOOP causes a value of 2 to be added to the value on the return stack for each pass through the loop. Thus, only even numbers are printed.

The +LOOP word can also be used to count through a loop backwards, by placing a negative value on the stack.

We may also nest DO-LOOPs one inside the other. Here, for instance, is a word that will count from 1 to 9 nine times:

```
: DOUBLE-COUNT 10 1 DO 10 DO I . LOOP CR LOOP ;
```

When we type DOUBLE-COUNT in the immediate mode, we will see the following display:

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

The first LOOP goes with the second DO, and the second LOOP goes with the first DO. Thus, the inner DO-LOOP will execute ten times every time the outer DO-LOOP executes once. Notice that we moved the CR in DOUBLE-COUNT out of the inner loop and into the outer loop by placing it between the two LOOPs. This gives us only one carriage return for every ten numbers; thus, each line is terminated with a carriage return.

Nested loops in a Forth word can often be confusing. It is easy, for instance, to forget which loop we are in at any given moment—and very difficult to understand how the loops work when we look at them again later. In fact, you may be having trouble understanding the definition of DOUBLE-COUNT given above. It would be better to define DOUBLE-COUNT in this manner:

```
: SINGLE-COUNT 10 1 DO I . LOOP ;
: DOUBLE-COUNT 10 1 DO SINGLE-COUNT CR LOOP ;
```

DOUBLE-COUNT will still perform in the same way as before, and it is still made up of two nested DO-LOOPs, but the loops are now contained in two separate words, and the definitions are easier to understand.

When two or more nested loops are executing simultaneously, the indexes of all the loops are maintained on top of the return stack, with the value of the innermost loop on top. We can always obtain the top value from the return stack by moving it to the parameter stack with the word I. However, Forth also allows us to obtain the values of the remaining loop indexes from the return stack by using the words J and K, which move the second and third values, respectively, from the return stack.

To sum up the DO-LOOP, its chief advantages are that (1) we can specify the precise number of times that a loop will execute and (2) we can take advantage of the internal count that the computer maintains of the number of passes it has made through the loop. (The number of times that the loop executes need not be specified in the definition of the word, however; rather, it may be passed to the loop as a pair of values placed on the stack by another word.)

There will also be times when we don't want to specify the number of times that we want a loop to execute; rather, we will want it to keep executing only as long as a certain condition is true, or until a specific event takes place.

Fortunately, there are loop words in Forth that give us this capability. One of these is BEGIN. In fact, the word BEGIN serves as the introduction to two different types of loops. The first is the BEGIN-UNTIL loop. There is no precise equivalent to this loop in BASIC, though Pascal programmers will recognize it as the REPEAT-UNTIL loop.

As its name implies, a BEGIN-UNTIL loop will begin with the word BEGIN. This word doesn't actually *do* anything; it simply acts as a place marker, letting the interpreter know where the beginning of the loop is. The real action is performed by UNTIL. The UNTIL word expects to find a flag on the stack. If the flag is false—that is, if the condition that has been tested for is not yet true—the interpreter will go back to the word BEGIN and start executing instructions once again from that point. If the flag is true, on the other hand, the loop will terminate, and execution will continue with the word after UNTIL. Thus, the instructions between the words BEGIN and UNTIL will be repeated again and again until the condition being tested for becomes true. (Some Forth systems use the word END in place of UNTIL.)

For instance, let us define a word that will print the word HELLO on the display every time you press a key on the keyboard, until you press the space bar, at which point it will print the word GOODBYE and terminate execution. This definition will use a new Forth word: KEY. The KEY word waits for you to press a key on the keyboard, then places the *ASCII* value of that key on the data stack. ASCII stands for American Standard Code for Information Interchange. It is a numerical coding system by which almost all computers represent alphanumeric characters. For instance, in ASCII the number 65 stands for the capital letter A, the number 49 for the numeral 1, etc. The blank space, in ASCII, is represented by the number 32. This is the number generated within your computer when you press the space bar and is the same number that will be detected by the Forth word KEY.

Here is our new word:

```
: WAIT-FOR-SPACE-BAR BEGIN KEY ." HELLO" 32 = UNTIL.
" GOODBYE";
```

The word BEGIN establishes the start of the loop. KEY waits for you to press a key, then pushes the ASCII value of the character represented by that key onto the stack. For

ASCII—The American Standard Code for Information Interchange. A coding system that assigns numeric values between 0 and 127 to letters of the alphabet, numerals, punctuation marks, and so on.

instance, if you typed the letter A, KEY would place the number 65 on the stack. The words

```
." HELLO"
```

print HELLO on the display. The expression

```
32 =
```

checks to see if the number on the stack is equal to 32, which would be true if you had pressed the space bar. Thus, when you press the space bar, the = word puts a "true" flag on the stack; otherwise, it puts a "false" flag on the stack. UNTIL looks for the "true" flag and keeps looping back to BEGIN if it doesn't get it. Finally, when you press the space bar, the loop will end and

```
." GOODBYE"
```

will print the word GOODBYE on the display.

We can use a BEGIN-UNTIL loop, combined with the word KEY, as a way of inputting entire lines of text into the computer. To do this, however, we will require another new word: EMIT. The EMIT word removes the top number from the stack and displays the ASCII character represented by that number. For instance, if the number on top of the stack is 65, EMIT will display an A. EMIT is particularly useful when used in conjunction with KEY. The KEY word can be used to get ASCII values from the keyboard, and EMIT can be used to display the appropriate characters, like this:

```
KEY EMIT
```

It is simple enough to put these two words into a loop, like this:

```
: GET-TEXT BEGIN KEY DUP EMIT 13 = UNTIL ;
```

The word GET-TEXT, as defined here, will accept a string of characters typed from the computer's keyboard and print that string on the display, until you press the RETURN key, which generates an ASCII 13. This is all done with a simple BEGIN-UNTIL loop. KEY gets text from the keyboard, DUP makes a copy of it, EMIT prints it on the video, and 13 = checks for the pressing of the RETURN key—a useful little loop.

However, as we shall see later, Forth already offers us standard words that will do this, and a lot more.

The final type of Forth loop, also beginning with the word BEGIN, is the BEGIN-WHILE-REPEAT loop. (Some versions of Forth use the words BEGIN-IF-AGAIN.) There is no precise equivalent to this loop in either BASIC or Pascal, although it bears a certain resemblance to the WHILE-DO loop in Pascal and in certain other languages.

Once again, BEGIN is simply a place marker, letting the interpreter know where the loop begins. The word REPEAT is also a place marker; it indicates the end of the loop and forces the interpreter to return to the word BEGIN and start executing the instructions between the two words one more time. The real action of this loop is performed by the word WHILE, which will appear at some point in the instructions between BEGIN and REPEAT. Like UNTIL, the word WHILE looks for a flag on the stack. The difference between the two words is that WHILE looks for a "true" flag (non-zero) while UNTIL looks for a "false" flag (0). When WHILE finds a "true" flag, nothing happens. Execution continues with the words following the WHILE, until the word REPEAT is encountered, at which point the loop is repeated. On the other hand, if WHILE finds a "false" flag on the stack, it terminates the loop and causes the interpreter to jump ahead to the next instruction after REPEAT. In short, the BEGIN-WHILE-REPEAT loop will continue repeating as long as a certain condition is true and will terminate when that condition becomes false, unlike the BEGIN-UNTIL loop, which will continue repeating as long as the specified condition is false and terminate when it is true. The difference between the two loops is summed up in the words WHILE and UNTIL. One loop executes "while" the condition is true, the other executes "until" the condition is true.

The other difference between the BEGIN-WHILE-REPEAT loop and the BEGIN-UNTIL loop is that you can exit the BEGIN-UNTIL loop only at the very end, after all the instructions have been performed. The exit from the

BEGIN-WHILE-REPEAT loop, on the other hand, may come at any point in the loop, wherever you choose to position the word WHILE. When the loop is terminated, all of the instructions between WHILE and REPEAT will be skipped.

Suppose, to give just one example of how the BEGIN-WHILE-REPEAT loop is used, that we are writing a game program that allows the user to wager a certain amount of imaginary money on the outcome of an event. We might offer the imaginary gambler the chance to double his or her bet, and to double it again and again if this is desired. In fact, we might want to define a word that performs this doubling repeatedly—until the gambler responds to our doubling offer in the negative. This word could make good use of a BEGIN-WHILE-REPEAT loop. We can ask the reader to respond with a Y for "yes" and an N for "no." Input could be accepted with the KEY word. A test can be provided for an ASCII 89 (the code for the letter Y); if any other letter is inputted, the loop will be terminated. It is assumed here that the amount of the bet will be on top of the stack when the word DOUBLE-BET? is executed:

```
: DOUBLE-IT 2 * ;
: DOUBLE-BET? BEGIN ." DO YOU WISH TO DOUBLE YOUR
BET (Y/N)?" KEY 89 = WHILE DOUBLE-IT REPEAT ;
```

The first word, DOUBLE IT, simply takes the number on top of the stack and doubles it—that is, multiplies it by two—and puts the result back on the stack. The second word, DOUBLE-BET?, constitutes a single BEGIN-WHILE-REPEAT loop, which prints a message asking the user if he or she wishes to double the bet amount that is on top of the stack. KEY accepts input, and 89 = looks to see if the user types the letter Y. The loop will continue only as long as (WHILE) the input is the letter Y; Any other input will cause the loop to terminate. If the loop continues past WHILE—that is, if the user inputs Y—the amount on the stack is doubled with our new word DOUBLE-IT, and the loop repeats. Thus, the user will be asked repeatedly if he or she wishes to double the bet, and the bet will be doubled once for each Y typed.

There is one problem with this loop. It is possible to double the amount on the stack too many times. Suppose that when DOUBLE-BET executes there is a bet of $1,000 on top of the stack. One doubling will bring that to $2,000, two doublings to $4,000, and so on. With only six doublings, we'll be up to $64,000, which will create a problem. Most Forth systems allow only numbers up to 32,768 or thereabouts to be manipulated on the stack with the integer operators that we are using; larger numbers produce odd and unpredictable results. (As mentioned earlier, there is a special set of Forth words for manipulating double-length numbers, which would allow us to use numbers much longer than this, but we will not be discussing those numbers in this book.) One more doubling and we'll go over this limit. Since this will mess up our results (and probably bankrupt the house as well), we'll have to prevent our user from doubling a number larger than about 16,000. Thus, we will have to create two exit conditions for this loop—one if the user inputs a character other than Y and one if the number to be doubled is too large.

Alas, Forth allows us to place only a single WHILE within a BEGIN-WHILE-REPEAT loop, so we will have to place tests for both of these conditions in front of the single WHILE. How do we do this?

BASIC programmers may be able to guess the answer to this. We tie both conditions together with the word AND, which will cause the loop to terminate if the first condition AND the second are true. In Forth, AND is a fully defined dictionary word. Its stack diagram looks like this:

```
(f f --- f)
```

As you can see, AND expects to find a pair of flags on the stack. It removes these flags, compares them, and places a third flag back on the stack. AND is called a *logical operator*, because it makes a logical comparison between the flags.

Exactly what sort of comparison does AND make? If both of the flags on the stack are true, AND will place a "true" flag on the stack. However, if one or both of the flags

Logical operators—Operators such as ANDs, ORs or NOTs that perform logical operations on numeric values.

on the stack are false, AND will place a "false" flag on the stack. In short, AND behaves much like the English word *and*—and much like the logical AND of other computer languages, such as BASIC. For instance, when we say "Statement A is true AND statement B is true," we have made a statement that will be true only if both substatements are true; if either is false, then the whole statement is false. The logical operator AND works in much the same fashion.

Here is a version of DOUBLE-BET? that looks both for an ASCII Y on the stack and a bet value of less than 16,000:

```
: DOUBLE-BET? BEGIN ." DO YOU WISH TO DOUBLE YOUR
BET? (Y/N)" KEY 89 = SWAP DUP 16000 < ROT AND WHILE
DOUBLE-IT REPEAT ;
```

What's happening in this definition is rather hard to follow, so let's go through it a step at a time. As before, the loop begins with BEGIN, the user is prompted for a "yes" or "no" response, and the response is accepted with KEY. Also as before, we compare that response to 89, the ASCII code for Y. Then, however, we swap the two values on top of the stack.

Why do we do this? Well, consider what the stack looks like when DOUBLE-BET? begins executing. The amount of the bet is on top of the stack, like this:

```
1500
```

assuming that the current bet is $1,500. When we key an ASCII value from the keyboard, it is placed on top of the bet value, like this:

```
  89
1500
```

assuming that the user responds with a Y. We then push another 89 on top of the stack, like this:

```
  89
  89
1500
```

The = word removes the top two values from the stack, compares them, and places a "true" flag on the stack if they are equal—which, in this instance, is the case. With the "true" flag in place, the stack now looks like this:

```
   1
1500
```

At this point, we wish to check the value of the bet and see if it has exceeded our 16000 limit. The bet, however, is no longer on top of the stack. To test it, we must bring it to the top. This is the purpose of the SWAP. After SWAP executes, the stack will look like this:

```
1500
   1
```

So that our next test will not destroy the bet value, we DUP it, like this:

```
1500
1500
   1
```

Now we throw a value of 16000 on the stack, like this:

```
16000
 1500
 1500
    1
```

The < word checks to see if 1500 is less than 16000. If so—as is the case here—it places a "true" flag on the stack. The stack now looks like this:

```
   1
1500
   1
```

Now we need to compare the two flags, using the AND word. However, we must first get them together on top of the stack, without the bet value between them. Thus, we use ROT to rotate the second flag to the top position on the stack, like this:

```
   1
   1
1500
```

The AND word compares the two flags. Because they are both true, it places a "true" flag on the stack, like this:

```
   1
1500
```

Finally, the "true" flag is taken off the stack by WHILE. Because the flag is true, the rest of the loop executes.

When one of the flags become false, however—either because the user inputs a letter other than Y or because the amount of the bet exceeds our limits— the AND word will make a different comparison. For instance, if the top of the stack looks like this before AND executes:

```
   0
   1
6000
```

then AND will place a "false" flag on the stack, like this:

```
   0
6000
```

The WHILE will register the "false" flag and terminate the loop, skipping over DOUBLE-IT.

There is another Forth word besides AND that performs comparisons between flags. It is OR, a word that will be familiar to those with previous programming experience. The stack diagram for OR will be identical to that for AND. Like AND, the OR word removes two flags from the top of the stack, compares them, and places a flag on the stack based on the results of the comparison. If either one or the other or both of the flags are true, OR places a "true" flag on the stack. If neither of the flags is true, OR places a "false" flag on the stack. OR is much like the English word *or*. When we say "Statement A is true OR statement B is true," this entire statement will be true if one *or* the other of the substatements is true *or* if both are true.

While the AND word is useful for comparing two conditions in a BEGIN-WHILE-REPEAT loop, to see if both remain true, the OR word is useful for comparing two conditions in a BEGIN-UNTIL loop, to ascertain if either condition has become true. And, of course, both AND and OR are also useful when used with the IF word.

Suggested Projects

1. Create and then define a Forth word that takes two numbers off the top of the stack and prompts the user to input either a plus sign (+) or a minus sign (−). If the user inputs a plus sign the word adds the two numbers and prints the result; if the user inputs a minus sign, it subtracts the number on the top of the stack from the number underneath it. (Hint: The ASCII value for + is 43, and the ASCII value for − is 45.)

2. Create and then define a Forth word that prints every sixth number from 654 to 1786. Create and define a Forth word that prints every thirteenth number from 7 to 1227.

3. Create and define a Forth word that prompts the user to type a character, then fills the video display with copies of that character.

4. Create and define a Forth word that prompts the user to type a number from 0 to 9, then doubles that number over and over again, printing the result

after each doubling, until its value exceeds 20000. (Hint: The ASCII value of 0 is 48 and the ASCII value of 9 is 57. The ASCII values of all other numbers fall sequentially between these two values.) Write this word so that it will reject any input that is not a number from 0 to 9.

5 FORTH AND MEMORY

Computers, as we noted way back in Chapter One, are devices for processing information. In order to process this information, they must be able to store it, and the location of this storage is the computer's memory. However, there are a number of different ways in which a computer can store information in its memory. So far we have concentrated on one particular way of storing information in memory—the stack.

Thre is a reason that we have concentrated so heavily on the stack. The stack is what makes Forth unique, what gives it its flavor, and it is what allows Forth to manipulate data at very high speeds. It is always a simple matter for a Forth program to get access to the stack.

However, it is not always possible for our Forth program to keep all data on the stack at the same time. For one thing, the size of the stack may be limited. Some Forth systems maintain larger stacks than others, but it is always possible for the stack to "overflow"—a condition that can wreak havoc with the operation of our programs. (This isn't a situation that you should lose sleep worrying about, though. It takes a lot of data to overflow the stack on most Forth systems.)

When we don't require immediate access to a number, we can store it elsewhere in the computer's memory—using *variables.* Variables should be familiar to anyone with previous programming experience, but those who think they already know what variables are and what they do should tread cautiously at first around Forth variables; they might not behave exactly as you'd expect.

In Forth, a variable is a word that represents the memory address at which a piece of information, usually a number, is stored. As we noted in the first chapter, a memory address is a number that identifies a specific location in a computer's memory. Thus, we can use variables to store information in memory and to keep track of where that information is stored.

We must declare a Forth variable before it is used. A *variable declaration* consists simply of the Forth word VARIABLE followed by the name we wish to give to the variable. To declare a variable called QUANTITY, for instance, we would write the following declaration:

```
VARIABLE QUANTITY
```

When it sees this declaration, the Forth interpreter will enter the name QUANTITY into the Forth dictionary. After this annotation, it will leave two bytes of computer memory free, for storing numeric values. (A byte is a measure of computer memory.) When we assign a value to the variable, it will be stored in these bytes.

In fact, we can store a value in QUANTITY at the time of declaration by placing the number in front of the word VARIABLE, like this:

```
150 VARIABLE QUANTITY
```

This stores the value 150 in the two bytes reserved for the

Variables—Locations within the computer's memory, generally identified by symbolic labels, that can be used to store numeric values.

Variable declaration—The assignment of a label to represent a location within the computer's memory.

variable QUANTITY. (Some versions of Forth require that we store a value in a variable at the time of declaration.)

The stack diagram for a variable looks like this:

```
(--- address)
```

The next time that the interpreter sees the name of our newly declared variable, it will place upon the stack the address of the two bytes of storage space allotted to that variable. For example's sake, let's say that the memory address of variable QUANTITY is 23781. If the stack looks like this before QUANTITY is referenced in a program:

```
   67
```

it will look like this afterward:

```
23781
   67
```

For this address to be useful to us, however, we also need a means of getting the contents of variable QUANTITY back out of that address. To do this, we use the Forth word @ (pronounced *fetch*). The stack diagram for @ looks like this:

```
(address --- nl)
```

The fetch word removes the address from the stack and literally "fetches" the number at that address, leaving it on the stack. (The Forth word @ is rather like the BASIC word PEEK, the primary difference being that @ retrieves a value stored in two bytes of memory rather than one. The precise Forth equivalent of PEEK would be C@, or *C-fetch*, which would fetch a single byte value from memory.) Thus, if we wish to retrieve the value that we have stored in variable QUANTITY, we must write:

```
QUANTITY @
```

When the interpreter sees the word QUANTITY, it will push the address of QUANTITY onto the stack. The @ word will go to that address and retrieve the value stored there. If the value stored at that address is 150, this pair of instructions will leave a value of 150 on the stack, just as though we had written the literal number 150 in our program. Now we may manipulate this value in any of the normal ways in which Forth allows us to manipulate numbers. For instance, we can print it as a number with the dot word, like this:

```
QUANTITY @ .
```

We can also add it to a number, subtract it from a number, etc.

If we wish to change the value stored at location QUANTITY, we can use the Forth word ! (pronounced *store*). The stack diagram for ! looks like this:

(nl address ---)

The ! word takes a number and an address off the stack and stores the number at that address in memory. (The Forth word ! is rather like the BASIC word POKE, except that it stores a two-byte number in memory rather than a one-byte number. The precise Forth equivalent of POKE would be C!, pronounced *C-store,* which stores a single-byte number in memory.) We can use ! to give a value to a variable, like this:

```
25 QUANTITY !
```

This stores the number 25 at location QUANTITY and is roughly the equivalent of the BASIC statement LET QUANTITY = 25.

It is sometimes difficult for programmers who have experience in other languages to grasp the idea that variables in Forth represent addresses, not values. In BASIC, for instance, a variable is usually treated by the interpreter as though it were the actual value that is stored in the computer's memory; a Forth variable is actually the *address* of that storage. You must always remember that in Forth it is necessary to go through an extra step to get the value out of memory, after the name of the variable has been referenced. It is not enough to say, simply, QUANTITY and expect the value of the variable QUANTITY to appear magically on the stack. You must say QUANTITY @.

(Actually, in languages such as BASIC, a variable is treated differently depending on the context in which it is used. When a variable appears in an expression, it is treated as though it were a value. On the other hand, when it appears on the left-hand side of an assignment statement—such as LET B = 5—it is treated as though it were a storage address. There is no such ambiguity about the role of the variable in Forth.)

To show how variables can be put to good use, here is the definition of a Forth word called PRICE?, which could be used in an inventory control program. PRICE? accepts a number from the stack representing a specific item in the inventory of a store. It then prints out the current price of the item.

```
87 VARIABLE EGG-PRICE 99 VARIABLE MILK-PRICE
225 VARIABLE CHEESE-PRICE 550 VARIABLE HAM-PRICE

: PRICE? ." THE PRICE OF THAT ITEM IS: "
          DUP 1 = IF EGG-PRICE @ . ." CENTS"
     ELSE DUP 2 = IF MILK-PRICE @ . ." CENTS"
     ELSE DUP 3 = IF CHEESE-PRICE @ . ." CENTS"
     ELSE     4 = IF HAM-PRICE @ . ." CENTS"
     ELSE ." UNKNOWN"
     ENDIF ENDIF ENDIF ENDIF ;
```

(In some versions of Forth, this definition may be too large to type in the immediate mode. You can get around this by breaking it up into smaller definitions, as we demonstrated in the last chapter. However, the next chapter will reveal how we can write definitions of almost any size that we wish.)

Before we define the word, we declare four variables called EGG-PRICE, MILK-PRICE, CHEESE-PRICE, and HAM-PRICE, storing values at each of these locations presumably representing the current price of these items. PRICE? then prints out this price, based on the item number that we have pushed on the stack. Item number 1, for instance, prints out the price of eggs, item number 2 the price of milk, item number 3 the price of cheese, and item number 4 the price of ham. If we place any other number on the stack, we are informed that the price is "unknown." We can use PRICE? like this:

```
3 PRICE?
```

And the computer will respond:

```
THE PRICE OF THAT ITEM IS: 250 CENTS
```

Or like this:

```
15 PRICE?
```

And the computer will respond:

```
THE PRICE OF THAT ITEM IS: UNKNOWN
```

The advantage of using variables in word PRICE? is that these values may change from time to time, and if they do, we need merely change the variables that contain them. This is especially valuable because we may have other words besides PRICE? that make reference to these values, and it would be awkward to have to make changes in all of them. This way, we can make all of the necessary changes simply by altering the values of the variables. For instance, we can create a word called PRICE-CHANGE that changes these prices. PRICE-CHANGE will expect to find two values on the stack—the number of the item to be changed on top of the stack and the new price of that item directly below it. Its definition might look like this:

```
: PRICE-CHANGE DUP 1 = IF DROP EGG-PRICE ! ELSE
               DUP 2 = IF DROP MILK-PRICE ! ELSE
               DUP 3 = IF DROP CHEESE-PRICE ! ELSE
               DUP 4 = IF DROP HAM-PRICE ! ELSE
                          DROP ."NO SUCH ITEM"
               ENDIF ENDIF ENDIF ENDIF ;
```

Follow through this definition carefully. We duplicate the number on top of the stack for each IF except the last, so that the number will be available to all succeeding IFs. However, once the value has been identified by an IF statement, it is no longer necessary to keep it on the stack—in fact, it gets in our way—so we drop it. The next number down on the stack is the value to be stored at the variable address. We then push the address of the variable onto the stack, on top of this value, and store the value at the address with the ! word. If the number on the stack is not one of our four item numbers, we announce that there is no such item and drop the remaining numbers from the stack. The four ENDIFs terminate the four IF statements.

If we know that a value will not be changed in the course of a program but nonetheless still wish to bestow a label on that value, we can define it as a *constant.* Pascal programmers will be familiar with constants, though there is no equivalent to constants in BASIC. A Forth constant superficially resembles a variable and is defined in similar fashion. However, while a Forth variable is a label that represents an address, a constant is a word that, when executed, pushes a value onto the stack.

We declare a constant like this:

```
12 CONSTANT DOZEN
```

This declares a constant called DOZEN, which is given the value 12. Now, if we use the word DOZEN later in the program, the interpreter will place the value 12 on the stack. For instance, if we write:

```
DOZEN 4 * .
```

the computer will calculate 4 times 12—that is, four dozen—and print the value on the display: 48. Note that it is not necessary to "fetch" the value of the constant in order to push it onto the stack.

Constants would be useful in conjunction with the words PRICE? and PRICE-CHANGE, which we defined earlier in this chapter. With these words, we must specify a particular inventory item by number, either 1, 2, 3, or 4,

Constant—A label that represents a value (rather than the location at which that value is stored).

which stand for eggs, milk, cheese and ham, respectively. This requires that we memorize these numbers and what they stand for. These words would be much easier to use—and easier for anyone else to understand—if we created some constants to use in place of these numbers. We can make the following constant declarations:

```
1 CONSTANT EGGS 2 CONSTANT MILK 3 CONSTANT
CHEESE 4 CONSTANT HAM
```

Now we can legitimately make Forth statements such as these:

```
CHEESE PRICE?
```

and

```
625 HAM PRICE-CHANGE
```

This makes it considerably more obvious that we are asking for the price of cheese and requesting a price change for ham than simply writing:

```
3 PRICE?
```

or

```
655 4 PRICE-CHANGE
```

Thus, we have made the program easier to use for all concerned—and possibly easier to write as well. If we make these constant declarations at the beginning of the program, which is always a good idea, we can use them in the actual definitions of PRICE-CHANGE and PRICE?, replacing the meaningless numbers that we used in the original definition.

Variables, while they might not be as necessary in Forth as in other languages, are still a powerful programming tool. Constants, while they might seem at first less useful than variables, also have their rightful place in the programmer's tool kit. You are encouraged to use both in the programs you write yourself.

Suggested Projects

1. Rewrite the definitions for PRICE? and PRICE-CHANGE using the four constants EGGS, MILK, CHEESE, and HAM.

2. Write a declaration for a Forth variable called NUMBER-OF-TIMES and assign it an initial value of 21. Write a short statement that will print the value of NUMBER-OF-TIMES on the video display. Write a statement that multiplies the value of NUMBER-OF-TIMES by 7. Write a statement that doubles the value stored in NUMBER-OF-TIMES. Write a statement that assigns a value of 1002 to NUMBER-OF-TIMES.

3. Create a variable called MINUS-TIMES and assign it a value of 81. Write a statement that subtracts the value of MINUS-TIMES from the value of NUMBER-OF-TIMES and prints the result on the video display.

4. Create and define a Forth word that asks you for a number between 0 and 9 and saves the number you choose in a variable called THIS-IS-IT.

6 ON THE SCREEN

Almost every computer language comes with an *editor*—a special program that allows us to create programs written in that language, make changes in those programs, save those programs on an external memory device such as a disk drive, and retrieve those programs from external storage.

Forth is no exception. So far, however, we don't seem to have created anything that could actually be edited. We have written definitions that have been compiled into the Forth dictionary, but the original definitions that we have written have been lost. Since the compiled definitions are written in an unreadable threaded code, what is left for us to go back and edit? Or to save on a disk drive?

The answer is: nothing. Once we have written a Forth definition and compiled it into the dictionary, we cannot make changes in it; we can only replace it with a newer definition. And most Forth systems lack any facilities for saving the compiled contents of the Forth dictionary to disk; thus, once we turn off the computer, all of our new definitions are gone and cannot be recalled.

What to do? No programmer wants to slave for hours

Editor—A computer program that assists in the creation of computer programs.

over a complex program only to lose that program as soon as the computer is put to rest for the evening. Fortunately, your Forth system provides an answer to this dilemma: the *Forth screen.*

A screen is a section of the computer's memory exactly one kilobyte (1,024 bytes) in size. Since a byte is the amount of computer memory needed to store a single ASCII character, a screen can be used to contain 1,024 characters of Forth programming, although the programmer may choose to fill a certain number of these bytes with blank spaces, just to make the program more readable when it appears on the video display. Once you have finished working on a Forth screen—either composing a new screen or editing an old one—most Forth systems will automatically save the screen onto a floppy disk, so that it will not be wiped out when the power is turned off. (A few Forth systems may store screens on cassette tape, using a system different from that we are describing. However, in this book, we will assume that you are using a disk-based Forth system.) In fact, most Forth systems save screens so automatically that you may not be aware that it has happened, unless you notice the sound made by your disk drives as the information is transferred.

Forth screens are identified by number. Although your Forth system may reserve only enough room in your computer's memory to save a few screens at any one time, they are usually numbered into the hundreds, the numbers of screens being limited only by the amount of space you have for screen storage on your disks. If your disk drives can hold 160 screens, then the highest numbered screen will be 160. Together, these screens will be able to hold 160 kilobytes (160K) worth of Forth programming.

Different Forth systems allow you to create Forth screens in different ways. You will have to read the manual for your Forth system before you can create screens. On some systems, for instance, you must type the word LIST

Forth screen—A section of computer memory, either in the computer's internal memory or on an external storage device such as a disk, that contains 1,024 characters of Forth programming or data.

preceded by the number of the screen you wish to write on. (LIST takes this number off the stack.) Forth will then display the contents of that screen on the display of your computer. You can then make changes to it using a special set of editing commands, as defined by your manual. Other systems may require you to type the word EDIT, preceded by the number of the screen you wish to edit. Still other systems may require you to run a special editor program, which will have its own unique set of commands, like a word processing program, which may not resemble any other commands used by your Forth system.

When you write a Forth word or definition on a Forth screen, it is not immediately executed or compiled. It is simply stored, in memory or on disk, as a series of ASCII characters. This is analogous to writing programs in BASIC, where the commands are not executed until we type RUN.

When we wish the interpreter to execute the instructions in a Forth program stored on screens, we use the command LOAD. LOAD expects to find a number on the stack—the number of the screen to be loaded. Thus, if we type 5 LOAD, screen #5 will be loaded. When we say that the screen will be loaded, we mean that the Forth system will get the screen off the disk (if it is not already in the computer's memory), and the interpreter will execute the commands on that screen. If the screen contains definitions to be compiled into the dictionary, the interpreter will call the compiler to compile them. If the program stretches for more than one screen, as most programs will, we must load each screen until all are loaded. Some Forth systems allow us to place a special word (-->) at the end of a Forth screen, which tells the interpreter to load the next numbered screen. Thus, if we place the --> word at the end of screen #5, the interpreter will automatically begin to load screen #6 as soon as screen #5 is loaded.

Once the screens have been loaded, it is as though you had typed all of the word definitions on those screens in the immediate mode. They are now in the dictionary and you may use the words defined on those screens as though they were standard Forth words.

Because Forth saves screens automatically to the disk, you will find that it is possible to write very long Forth programs; in fact, you may write Forth programs that are actually larger than the memory of your computer, because at any one time the majority of the program is stored on the disk, not in the computer's memory. It is only necessary that your computer have sufficient internal memory for the compiled dictionary that is created when these screens are loaded. Since compiled Forth code generally takes up far less memory space than the screens from which that code is compiled, this is not as much of a limitation as it sounds. Forth screens make it seem as though the computer has a much larger internal memory than it actually does, since the computer juggles the screens on and off the disk in a way that is almost transparent to the programmer. Thus, Forth is a good language to use on computers with limited internal memory.

When we write a program on a Forth screen, it is traditional to place a special message on the first line of the screen, framed in parentheses, explaining what the contents of the screen are, like this:

```
( WORD-PROCESSOR, VERSION 2, BY JOE DOAKS )
```

The left parenthesis is a Forth word that tells the interpreter to ignore everything that follows, until it encounters a right parenthesis. Thus, we may write any kind of message that we wish on a Forth screen and it will be ignored by the interpreter—as long as it is surrounded by parentheses. (Note, however, that there must be at least one space between the left parenthesis and the message that follows.) This gives us a means of "documenting" our programs, of placing remarks and comments in them that will remind us of how the Forth words that we are defining are used. Although it is wise to begin every Forth screen with such a comment, there is no reason to restrict comments to the first line of the screen. They may be placed at any point on a Forth screen where our programming intentions may not be clear. (We may also use parentheses in the immediate mode, but there is no real point to it, since the comment thus enclosed will not be saved for later reference.)

To see the contents of any screen being maintained by your Forth system, whether that screen is currently in the computer's memory or on a disk, type the word LIST, preceded by the number of the screen you want to see. If you are interested in the contents of screen 123, for instance, you would type 123 LIST. This will cause the computer to get screen 123 from the disk, if it is not already in memory, and display its contents on the computer's video display.

Forth screens cannot only be used for storing programs, but they can also be used to store data from Forth programs, thus allowing us to store files of information on a disk. One of the words that allows us to do this is BLOCK.

The stack diagram for BLOCK looks like this:

(nl --- ADDRESS)

BLOCK expects to find a block number on the stack. It takes that number from the stack, gets that screen from the disk, then stores it in the computer's internal memory. It then places the address of the first byte of that screen on the stack.

Thus, if we write 5 BLOCK, the interpreter will read screen #5 from the disk, store it in the computer's memory, and place the address of screen #5 on the stack. This address is the address at which the screen is stored in memory—that is, the address of the first byte of screen #5.

Once we have used the BLOCK word to get a block from the disk, we may write on that block using standard Forth words. What can we write? Anything we want. And when we are finished writing, we can have the computer store the block back on the disk.

For instance, you may recall a Forth word, defined in an earlier chapter of this book, that we called GET-TEXT. It allowed us to type text on the keyboard of the computer and see that text displayed. Here, once again, is the definition of GET-TEXT:

```
: GET-TEXT BEGIN KEY DUP EMIT 13 = UNTIL ;
```

This word could be used as the core of a word processing program, say, or any program that requires the input of a large amount of ASCII text. However, it has one main deficiency. It doesn't save the text that we type. We can easily remedy that.

Look at this definition:

```
0 VARIABLE STORAGE
: STORE-TEXT 10 BLOCK STORAGE ! BEGIN KEY DUP EMIT
DUP STORAGE @ ! 1 STORAGE +! 13 = UNTIL ;
```

There is a new Forth word in this definition: +! (pronounced *plus store*). The +! word does the work of several other words. It takes a value and an address off the stack and adds the value to the value already stored at the address.

In the definition of STORE-TEXT, we use +! to add 1 to the value at location STORAGE.

What does STORE-TEXT do? Like GET-TEXT, it gets a string of characters from the keyboard. However, it also stores those characters in memory using the C! word. Where in memory does it store them? In block 10, which is the area of memory where screen #10 is stored. We accomplish this by establishing a variable called STORAGE and using it to store the address of block 10, like this:

```
10 BLOCK STORAGE !
```

Then we create a loop that inputs ASCII characters, pushes them onto the stack, and stores them at the location "pointed to" by STORAGE, like this:

```
STORAGE @ C!
```

Note that this phrase stores the stack value at the memory address stored in STORAGE, not at location STORAGE itself. Thus, the value of variable STORAGE is unchanged by this operation; instead, the ASCII characters are stored in block 10, which is pointed to by the address stored in STORAGE.

We then use the words 1 STORAGE +! to increase the address stored in STORAGE by 1, so that STORAGE now points to the next address in the computer's memory. In this way, we store a string of characters at a string of memory locations, until we press RETURN (which generates an ASCII 13).

If you're having trouble understanding exactly how STORE-TEXT stores its text, don't worry about it. As it happens, Forth already supplies us with a word that does pretty much the same thing as STORE-TEXT. Since this word is already in your Forth dictionary, there's really no reason for us to reinvent the wheel. The word is EXPECT.

Here's the stack diagram for EXPECT:

(address nl ---)

EXPECT takes an address and a number from the stack. It then gets that number of characters from the keyboard of the computer and stores them in the computer's memory, starting at the address. Thus, we can accomplish pretty much the same thing as STORE-TEXT by writing:

```
10 BLOCK 100 EXPECT
```

This will allow you to type a hundred characters on the keyboard—you may terminate input before you reach this number by pressing RETURN—and will store those characters beginning at the start of block 10.

Try it and see. Type 10 BLOCK 100 EXPECT in the immediate mode. The computer will seem to pause as it waits for you to type some characters on the keyboard. Type a few characters and press RETURN. The characters are now stored in the computer's memory.

You might wonder, however, just what good this does us. How, for instance, can we get these characters back out?

There are, in fact, several ways to get the characters back out. One of them is to use the Forth word TYPE. The stack diagram for TYPE is identical to the one for EXPECT. TYPE takes an address and a number from the stack. It gets that number of characters, beginning at that address, and displays them on the video monitor.

To retrieve the characters that we typed with EXPECT, we would write:

```
10 BLOCK 100 TYPE
```

This displays one hundred characters from memory, beginning at the start of block 10. Suppose, for instance, that we typed the words FOUR SCORE AND SEVEN YEARS AGO, OUR FOREFATHERS BROUGHT FORTH UPON THIS CONTI-

NENT after entering the EXPECT word. Now we type 10 BLOCK 100 TYPE. The following might appear on the display:

```
FOUR SCORE AND SEVEN YEARS AGO, OUR FOREFATH
ERS BROUGHT FORTH UPON THIS CONTINENT DUP TEXT
C!
```

Why is our input followed by junk? Since we did not type a full hundred characters, there was apparently some random garbage left in the computer's memory after the end of the word CONTINENT.

Most likely this garbage is left over from whatever was on screen #10 before we started typing on it. Thus, if we plan to store text on a Forth screen, we might do well to clean it up first. How can we do this? With the Forth word FILL or the Forth word BLANKS.

The word FILL allows us to "fill" an area of the computer's memory with an ASCII character of our choice. If we wish to cover an entire Forth screen with blank spaces, for instance, we can use the FILL command to fill an area of memory with 32s, the ASCII value for a blank space. The FILL command expects to find three numbers on the stack, like this:

(address n1/n2 ---)

The FILL command will then fill memory starting at the address and continuing for nl addresses with ASCII character n2. To fill block 10 with ASCII spaces, we would write:

```
10 BLOCK 1024 32 FILL
```

This would fill all 1,024 bytes of block 10 with ASCII 32s—that is, blank spaces. If we wished to fill block 10 with another character, we would substitute the ASCII code for that character in place of 32. For instance, if we wanted to fill block 10 with the letter F, we would write 10 BLOCK 1024 70.

Actually, we don't have to do quite this much work to fill block 10 with spaces. Forth also offers us a word called

BLANKS that takes an address and a number of bytes from the stack and fills the designated area with ASCII blanks, like this:

```
10 BLOCK 1024 BLANKS
```

This will fill block 10 with blanks.

Now that we have cleared an entire screen to blanks, we may write on it without fear that our subsequent printout will be filled with garbage. Now we can type 10 BLOCK 100 TYPE and see only what we have deliberately stored on block 10:

```
FOUR SCORE AND SEVEN YEARS AGO, OUR FOREFATH
ERS BROUGHT FORTH UPON THIS CONTINENT
```

Nonetheless, there will be a lot of blank spaces after the end of CONTINENT, which might get in the way of subsequent text to be typed. We can get rid of these by placing the word -TRAILING (pronounced *not trailing*) directly in front of the word TYPE. -TRAILING checks the area to be typed and adjusts the number of bytes on the stack so that it doesn't encompass any trailing blanks—that is, blank characters after the last character of text. Thus, we would write:

```
10 BLOCK 100 -TRAILING TYPE
```

and receive a printout of our text that doesn't have any trailing blanks.

Now that we've found a way to store text on Forth screens, it would be useful to save that text to the disk. That way, we could use Forth words such as EXPECT and TYPE to create word processing programs, data base programs, or even Forth editors (if you should want to improve on the one that came with your Forth system; most Forth editors, after all, are written in Forth).

However, Forth will not automatically store our scribbled-on screen back onto the disk. First, we must tell the interpreter that the screen has been "updated"—that is, that we have made changes on it since it was last loaded from the disk. This is done with the Forth word UPDATE, which says that the current block—the one most recently referred to in a BLOCK command—has been changed. Now, if the space in the computer's memory occupied by the block should later be used to store a different screen, all of your changes to the screen will automatically be stored on the disk. However, to be on the safe side, we might also use the Forth word FLUSH, which causes the computer to save all screens back onto the disk immediately. Thus, to type a line of text and store it on a floppy disk we need four words—BLOCK, EXPECT, UPDATE, and FLUSH—in addition to the other words required to clean off the screen, type out its contents, etc.

Once we have stored our text on the disk, we can check to see that it's really there by typing LIST, preceded by the number of the screen. The contents of the screen will be listed to the video display and the text that we've typed should be there also.

Suppose, for instance, that we wanted to write a Forth program that would accept a list of ten names from the computer's keyboard and store those ten names on a floppy disk. Such a program might look like this:

```
( NAME-LIST PROGRAM )
: GET-NAME 64 EXPECT ;
: CLEAR-BLOCK BLOCK 1024 BLANKS ;
: NAMES 10 CLEAR-BLOCK 576 0 DO 10 BLOCK I + GET-
  NAME CR 64 + LOOP UPDATE FLUSH ;
```

The first new word, GET-NAME, simply accepts sixty-four characters from the keyboard and saves them starting at the address that it finds on that stack. The second new word, CLEAR-BLOCK, takes the number of a memory block off the stack and fills that block with ASCII blanks. The third word, NAMES, clears block 10, sets up a loop that executes GET-NAME ten times (incrementing the storage address by sixty-four characters each time with 64 +LOOP), storing the names input by GET-NAME sequentially through block 10, and stores the contents of the block on the disk using UPDATE and FLUSH.

Run this program, then type 10 LIST to make sure that the ten names that you typed were stored on the disk. Since the LIST command usually supplies identifying numbers for each line of a Forth screen, your list will probably look something like this:

```
SCR # 10
 0 DAVID BISCHOFF
 1 SUSAN WIENER
 2 DENNIS BAILEY
 3 MAURY SOLOMON
 4 CHARLES SHEFFIELD
 5 RICH WEINSTEIN
 6 PATRICIA TUCKER
 7 RONALD REAGAN
 8 RICHARD SIMMONS
 9 ELIZABETH STANFORD
10
11
12
13
14
15
```

Note the six extra lines at the bottom of the screen. If we wished, we could also fill these with names, by increasing the size of the loop. Furthermore, we could store names across a large number of Forth screens by tinkering with the definition of NAMES a little. Perhaps we could set up a second loop that would go from screen to screen. In this fashion, we could store a large number of names. Of course, all those blank spaces on the screen after each name wastes a lot of storage space; none of these names uses up the full sixty-four characters allotted to it. With a little ingenuity, we could store those names a little more efficiently. This, however, is left as a project for the reader.

Note also that our NAMES word is actually a rudimentary word processing program. Although it does not offer us any method of editing text, it does give us a way of getting text into memory and saving it on an external storage device. (One of the amazing things about Forth is how efficiently it lets us do things like this, which might take hundreds of commands in other languages.)

It is only a short step from being able to store and save text to being able to edit it; interested readers are encouraged to develop their own word processing programs in Forth, which is a much better language for such applications than, say, Pascal, which offers very poor text-handling facilities.

There are a number of other words in Forth that can be used for manipulating text stored in the computer's memory or on the disk drive. (And, of course, you can use these words to define your own text-handling words in turn.) For instance, the word CMOVE can be used to transport a series of memory bytes from one section of memory to another—or even between screens. The stack diagram for CMOVE looks like this:

(address1 address2 nl ---)

CMOVE removes three numbers from the stack. The first number tells it the starting address of the block of memory to be moved. The second number tells it the starting address of the block it is to be moved to. The third tells it the number of bytes to be moved. If, for instance, we wished to move our list of names from screen #10 to screen #15, we might write this:

```
10 BLOCK 15 BLOCK 1024 CMOVE
```

This places on the stack the addresses of block 10 and 15 and the number of bytes in a block. The CMOVE word then moves all of these bytes from one block to the other. If we then type UPDATE and FLUSH, these changes will be reflected on the disk.

Other Forth words that deal with disk storage, text, and the contents of the computer's memory are also available. Thumb through your Forth manual and find out what they are.

Those of you who have programmed in other languages may find it odd that we have to go to this much trouble to manipulate text in Forth. In BASIC, for instance, we manipulate strings of ASCII characters by storing them in string variables. The statement LET A$ = "NOW IS THE TIME FOR ALL GOOD MEN", to use but one example, stores the string NOW IS THE TIME FOR ALL GOOD MEN in variable A$. (The dollar sign indicates that this is a variable for storing text.) Most versions of Forth have no string variables, however; instead, Forth gives us a set of rudimentary but powerful tools, such as the words we have been examining in the last few pages, and lets us store our own strings—and define our own systems of string variable storage. The result is that string manipulation can be more awkward at first in Forth than in BASIC; ironically, however, this means that string manipulation is also more powerful in Forth. We are not limited to certain predefined ways of storing and manipulating strings. For example, a string of characters in most versions of BASIC is limited to 256 characters; in Forth, it may be as big as the computer's memory, or at least as big as the portion of the computer's memory not being used for other purposes.

And we need not be limited to storing our strings using the BLOCK word. We may deliberately create space in the Forth dictionary for storing strings with the ALLOT word, in effect creating our own string variables.

Ordinarily, when we create a Forth variable, two bytes of memory are set aside for the storage of values in that variable. This is sufficient for storing integer numbers, but we would only be able to store two characters of a string in such a variable. However, we may enlarge the number of bytes set aside for storage in a variable with the word ALLOT. The word ALLOT must be used as soon as possible after a variable is created, because it only affects the most recently declared variable. ALLOT expects to find a single number on the stack, like this:

```
(n1 ---)
```

ALLOT will add n1 bytes to the allotted space in the most recently declared variable. For instance, if you have just defined a variable called MANUSCRIPT, the compiler will have automatically allotted two bytes of storage space to the variable. However, if you wish to add more bytes of storage space (for storing a manuscript, say), you must allot the extra space. For instance, the statement

```
0 VARIABLE MANUSCRIPT 1000 ALLOT
```

will add 1,000 bytes of storage space to variable MANUSCRIPT, for a total of 1,002 bytes. You may now fill these 1,002 bytes with text by using the word EXPECT, like this:

```
MANUSCRIPT 1002 EXPECT
```

Then, if we wish to print the contents of MANUSCRIPT back on the display, we can use the TYPE command. Or, if we wish to save it on a disk, we can use the MOVE command to move a copy of it to a screen block, where it can be updated and flushed.

To further demonstrate how strings are handled in Forth, let's attempt to duplicate, in Forth, a simple BASIC program for accepting someone's name from the keyboard of the computer and greeting them with that name in turn. In BASIC, such a program might look like this:

```
10 PRINT "WHAT IS YOUR NAME?"
20 INPUT A$
30 PRINT "NICE TO MEET YOU, ";A$
```

The BASIC command INPUT accepts a string of characters from the keyboard and stores it in a string variable. We can simulate the INPUT command in Forth with the EXPECT word. In Forth, this program might look like this:

```
0 VARIABLE NAME 20 ALLOT
: HELLO CR ." WHAT IS YOUR NAME?" CR
    NAME 20 EXPECT CR
    ." NICE TO MEET YOU, "
    NAME 20 -TRAILING TYPE
```

A lot more elaborate than the BASIC version, the string-handling commands in Forth are also a lot more versatile and powerful than their BASIC equivalents.

Suggested Projects

1. Create and define a Forth word that fills an entire screen with the letter A (ASCII 65) and prints its contents on the video display.

2. Create and define a Forth word that accepts a string of ASCII characters from the keyboard and compares it with a string of ASCII characters already in memory, to see if they are identical.

3. Create and define a Forth word that moves the contents of variable MANUSCRIPT (all 1,002 allotted bytes) onto a screen and saves it to the disk.

7 PUTTING THE PIECES TOGETHER

Like Humpty Dumpty after his fall, the Forth language may sometimes seem like a collection of disparate pieces that must be patched together with spit and glue before it can do anything useful. Fortunately, it doesn't require all of the king's horses and all of the king's men to get a Forth program into functioning order.

Nonetheless, the idea of developing a complete, complex program in Forth may seem rather intimidating at first. There is something almost chaotic about Forth, with all of those words performing their own little tasks, more or less ignoring one another except when they pass numbers back and forth via the stack. The prospective Forth programmer may be forgiven for wondering how he or she is going to pull all of those words together into a coherent whole.

You shouldn't be intimidated. It's true that Forth may require more work from the programmer than other high-level languages, but it also frees the programmer from some serious constraints. The power of Forth is that it can be almost any kind of programming language that you want it to be, but this in turn requires that you put a great deal of thought into deciding just what kind of language you want it to be. This decision should be based, at least in part, on the nature of the program that you are trying to write.

The development of complete programs in Forth calls for a special programming method called *top-down programming*. Those of you who have programmed in other languages may already be familiar with top-down techniques. However, while programmers working in any language can benefit from top-down methods, programming in Forth is almost impossible without them.

When you put together a program in top-down fashion, you have to look at the big picture first. You must ask yourself what the program is going to do, and then write a brief outline detailing the major steps that it will take to accomplish this. You can then break each of those major steps into a sequence of smaller steps, and each of those smaller steps into still smaller steps, and so forth, until the outline has resolved itself into a series of actual program statements.

Notice that this description of top-down programming sounds a lot like a description of a Forth program, but in reverse. In Forth, a program begins with a few simple words being used to create a few slightly less simple words that are used to create slightly more complex words that are used to create still more complex words, and so forth until we have one big word that executes the entire Forth program. It follows, then, that when we write a Forth program using top-down techniques, we simply write it backwards.

This may sound odd, but it is certainly the best way to write a program in Forth. We start out by writing the definition for that final word, the one that will cause our entire program to execute. Then we write definitions for all of the new words that we used in writing that definition, and then we write definitions for all of the new words that we used in writing those definitions, and so on, until we create a series of definitions that contain no new words. At that point, we have our Forth program.

To demonstrate how this works, let's design a Forth

Top-down programming—A system for writing computer programs in which the broad features of the program are determined first and then the details are developed subsequently.

program in top-down fashion, starting with the most complex portion of the program and working our way down to the least complex. First, of course, we have to decide what our program will do.

Suppose, for instance, that we wish to create a Forth program that will draw a picture of a Christmas tree and print the message MERRY CHRISTMAS on the video display. Such a program can be summed up in a single word—DRAW-TREE—and that in fact can be the name of the word that will activate the entire program. We might want to draw our tree with asterisks, so that it will look like this:

```
       *
     *   *
    *     *
   *       *
  *         *
 *           *
*             *
 *****   *****
    *     *
    *     *
    *******
```

This may not be the most complex computer program imaginable, but it is one with subtle ramifications that might not be obvious immediately. Of course, we could write this program as a series of

```
."
```

statements that would print each line of the tree separately, or we could simply draw the tree on a Forth screen and call it up to the video display. However, neither of these methods seems quite fair; we need to write a program that will not only draw this tree but will figure out the number of asterisks and the number of spaces required to draw each line. Fortunately, there are fairly obvious mathematical relationships between the lines in this tree, and so we may well be able to come up with a formula that will allow us to describe the tree in program terms.

It is apparent that our tree breaks down into two sections: the upper tree and the trunk. In top-down fashion, then, we may break program DRAW-TREE down into the following definition:

```
: DRAW-TREE UPPER-TREE TRUNK ;
```

The individual sections of the tree, in turn, will be drawn by using a mathematical formula that will calculate the number of spaces and the number of stars in each section of the tree. We can break each section down into a series of lines, each line being made up of stars and spaces. The individual lines will each begin with a series of spaces, which we will represent by the variable FIRST-SPACES; a single star; a second series of spaces, which we will represent by the variable SECOND-SPACES; and a second star. Only the number of spaces will change from line to line, and we will need a special word, called CALCULATE-NEXT, to calculate the number of spaces in each section for each line. Thus, the definition of UPPER-TREE might look like this:

```
0 VARIABLE FIRST-SPACES 0 VARIABLE SECOND-SPACES
: UPPER-TREE CR 11 FIRST-SPACES ! -1 SECOND-SPACES
! DRAW-FIRST-SPACES 1 STAR CR
6 0 DO CALCULATE-NEXT DRAW-FIRST-SPACES 1 STAR
DRAW-SECOND-SPACES 1 STAR CR LOOP
4 STAR 5 SPACES 6 STAR;
```

The definition of TRUNK is fairly simple:

```
: TRUNK CR  3 1 DO 8 SPACES 1 STAR 6 SPACES 1 STAR
            8 SPACES 7 STAR ;
```

In turn, we must now define all of the undefined words.

```
: DRAW-FIRST-SPACES FIRST-SPACES @ 0 DO 1 SPACES
 LOOP ;
: DRAW-SECOND-SPACES SECOND-SPACES @ 0 DO 1
 SPACES LOOP ;
: CALCULATE-NEXT FIRST-SPACES DUP @ 1 - SWAP !
                 SECOND-SPACES DUP @ 2 + SWAP ! ;
: STAR 0 DO ." *" LOOP ;
: SPACES 1 DO 32 EMIT LOOP ;
```

(The word SPACES will already be defined on many systems.)

To put these together into a complete Forth program, we need only type all of the definitions in reverse order (with the variable declarations at the front, of course), like this:

```
( CHRISTMAS TREE )
0 VARIABLE FIRST-SPACES 0 VARIABLE SECOND-SPACES
: STAR 0 DO ." *" LOOP ; : SPACES 1 DO 32 EMIT LOOP ;
: CALCULATE-NEXT FIRST-SPACES DUP @ 1 - SWAP !
SECOND-SPACES DUP @ 2 + SWAP ! ;
: DRAW-SECOND-SPACES SECOND-SPACES @ 0 DO 1
 SPACES LOOP ;
: DRAW-FIRST-SPACES FIRST-SPACES @ 0 DO 1 SPACES
 LOOP;
: TRUNK CR 3 1 DO 8 SPACES 1 STAR 6 SPACES 1 STAR CR
LOOP 8 SPACES 7 STAR ;
: UPPER-TREE CR 11 FIRST-SPACES ! -1 SECOND-SPACES
! DRAW-FIRST-SPACES 1 STAR CR
6 0 DO CALCULATE-NEXT DRAW-FIRST-SPACES 1 STAR
DRAW-SECOND-SPACES 1 STAR CR LOOP
4 SPACES 5 STAR 6 SPACES 5 STAR ;
: DRAW-TREE UPPER-TREE TRUNK ;
```

Of course, it would be possible to break some of the longer definitions, like UPPER-TREE, down into a series of smaller definitions, to improve clarity, and add extra steps to the top-down process. This is left to the reader, however, who is free to modify this program as he or she desires.

Suggested Projects

Develop a Forth programming project of your own. Start with a top-down outline, break it down into smaller segments, and write the resulting program on a series of Forth screens. Good luck!

APPENDIX

SOME FORTH WORDS

! (n addr ---)
—Stores value n (number) at an address in the computer's memory.

(
—Defines the beginning of a comment field within a Forth program.

* (n1 n2 --- n3)
—Multiplies the top two numbers on the stack and leaves the result on top of the stack.

+ (n1 n2 --- n3)
—Adds the top two numbers on the stack and leaves the result on top of the stack.

+LOOP (n ---)
—Adds number to the loop index on the return stack and returns control to the most recent uncompleted DO statement.

− (n1 n2 --- n3)
—Subtracts number 1 from number 2 and leaves the result on the stack.

−TRAILING (addr n1 --- addr n2)
—Adjusts the count in number 1 to remove trailing blanks from the string beginning at an address, leaves the address and the adjusted count on the stack. Generally used before the TYPE word.

. (n ---)
—Outputs the numeric value on top of the stack to the video display.

."
—Prints a string of text to the video display. Text must follow this word, after a separating space, and be terminated by a quote mark.

:
—Tells the Forth compiler to compile the following word and its definition into the Forth dictionary. Must be followed by a semicolon.

;
—Terminates a colon definition. See above definition.

/ (n1 n2 --- n3)
—Divides number 1 by number 2 and leaves the result on top of the stack.

< (n1 n2 --- f)
—Compares the top two numbers on the stack. If number 1 is less than number 2, leaves a "true" flag (1) on top of the stack; else leaves a "false" flag.

= (n1 n2 --- f)
—Compares the top two numbers on the stack. If number 1 is equal to number 2, leaves a "true" flag (1) on top of the stack; else leaves a "false" flag.

> (n1 n2 --- f)
—Compares the top two numbers on the stack. If number 1 is greater than number 2, leaves a "true" flag (1) on top of the stack; else leaves a "false" flag.

@ (addr --- 1)
—Fetches the number stored at an address and leaves it on top of the stack.

ALLOT (n ---)
—Opens a specified number of bytes of storage at the current location in the dictionary.

AND (n1 n2 --- n3)
—Performs a logical AND on numbers 1 and 2 and leaves the result (number 3) on top of the stack.

BEGIN
—Designates the beginning of a BEGIN-UNTIL or BEGIN-WHILE-REPEAT loop.

BLOCK (n --- addr)
—Leaves on the stack the address of the beginning of a particular block number. Loads the block from disk if necessary.

CMOVE (addr1 addr2 n ---)
—Moves a specified number of bytes of memory starting at address 1 to the memory locations starting at address 2.

CR
—Outputs a carriage return to the video display.

DO (n1 n2 ---)
—Designates the beginning of a DO-LOOP. Places number 1 on the return stack as the loop index and continues looping until the index value equals number 2.

DROP (n ---)
—Removes the top number from the stack.

DUP (n1 --- n1 n1)
—Duplicates the top element on the stack.

ELSE
—Indicates alternative action to be taken in an IF statement if the flag on the stack is not true.

EMIT (n ---)
—Displays the value on top of the stack as an ASCII character.

ENDIF
—Designates the point at which execution resumes after an unexecuted IF or ELSE statement.

EXPECT (addr n ---)
—Inputs characters from the keyboard and stores them starting at an address, until specified number of characters has been received or a carriage return is typed.

FILL (addr n1 n2)
—Fills a specified number of bytes of memory starting at an address with the value number 2.

FORGET {NAME}
—Removes the definition of {NAME}, and of all subsequently defined words, from the Forth dictionary.

I (--- n)
—Copies the top value from the return stack to the parameter stack.

IF (f ---)
—Removes flag from the stack. If flag is true (non-zero), executes all statements until an ELSE is encountered, then skips to the next ENDIF (or THEN) statement. If flag is false (zero), skips to next ELSE or ENDIF, whichever is encountered first.

J (--- n)
—Copies the second value from the return stack to the parameter stack.

KEY (--- n)
—Inputs a character from the keyboard and leaves its ASCII value on the stack.

LIST (n ---)
—Displays the contents of a specified screen.

[97]

LOAD (n ---)
—Compiles the contents of a specified screen, loading it from disk if necessary.

LOOP
—Increments the value on the return stack by 1 and returns control to the most recent uncompleted DO statement.

MOD (n1 n2 --- n3)
—Divides number 1 by number 2 and leaves the remainder on top of the stack.

NOT (f1 --- f2)
—Reverses the logical value of the flag on the stack. If the flag is 0, it becomes 1; if the flag is not 0, it becomes 0.

OR (n1 n2 --- n3)
—Performs a logical OR on the top two values on the stack and leaves the result on top of the stack.

OVER (n1 n2 --- n1 n2 n1)
—Copies the second element on the stack to the top of the stack.

PICK (n ---)
—Copies the value of a specified item in the stack to the top of the stack.

REPEAT
—Transfers control to the most recent uncompleted BEGIN statement.

ROT (n1 n2 n3 --- n3 n2 n1)
—Moves the third element in the stack to the top and the top element to the third position.

SWAP (n1 n2 --- n2 n1)
—Reverses the top two elements on the stack.

THEN
—Designates the point at which execution resumes after an unexecuted IF or ELSE.

TYPE (addr n ---)
—Outputs the specified ASCII characters beginning at an address to the video display.

VARIABLE {NAME}
—Opens space in the Forth dictionary to store the value of variable {NAME}.

WHILE (f ---)
—Terminates the execution of a BEGIN-WHILE-REPEAT loop and passes control to the words following the next REPEAT statement if the value of flag f is true (non-zero).

INDEX

ABOUT THE AUTHOR

Christopher Lampton is the author of more than twenty books for Franklin Watts, including a number of popular First Book and Impact titles. Of late he has turned his attention to the world of computers, writing on a variety of programming languages and teaching the basics to beginners.

Chris first became a computer enthusiast when he purchased a Radio Shack computer to use for word processing. He has since acquired eight more computers.

Chris lives in Maryland, near Washington, D.C., and has a degree in broadcast journalism. In addition to his writings in the areas of science and technology, he has authored four science-fiction novels.