I was introduced to finite-state machines by my mentor Marvin Minsky. He presented me with the following famous puzzle, called the *firing squad problem*: You are a general in charge of an extremely long line of soldiers in a firing squad. The line is too long for you to shout the order to "fire," and so you must give your order to the first soldier in the line, and ask him to repeat to the next soldier and so on. The hard part is that all the soldiers in the line are supposed to fire at the same time. There is a constant drumbeat in the background; however, you can't even specify that the men should all fire after a certain number of beats, because you don't know how many soldiers are in the line. The problem is to get the entire line to fire simultaneously; you can solve it by issuing a complex set of orders which tells each soldier what to say to the soldiers on either side of him. In this problem, the soldiers are equivalent to a line of finite-state machines with each machine advancing its state by the same clock (the drumbeat), and each receiving input from the output of its immediate neighbors. The problem is therefore to design a line of identical finite-state machines that will produce the "fire" output at the same time in response to a command supplied at one end. (The finite-state machines at either end of the line are allowed to be different from the others.) I won't spoil the puzzle by giving away the solution, but it can be solved using finite-state machines that have only a few states.

Before showing you how Boolean logic and finite-state machines are combined to produce a computer, I'll skip ahead in this bottom-up description and tell you where we're going. The next chapter starts by setting out one of the highest levels of abstraction in the function of a computer, which is also the level at which most programmers interact with the machine.

CHAPTER 3

PROGRAMMING

The magic of a computer lies in its ability to become almost anything you can imagine, as long as you can explain exactly what that is. The hitch is in explaining what you want. With the right programming, a computer can become a theater, a musical instrument, a reference book, a chess opponent. No other entity in the world except a human being has such an adaptable, universal nature. Ultimately all these functions are implemented by the Boolean logic blocks and finite-state machines described in the previous chapter, but the human computer programmer rarely thinks about these elements; instead, programmers work with a more convenient tool called a *programming language*.

Just as Boolean logic and finite-state machines are the building blocks of computer hardware, a programming language is a set of building blocks for constructing computer software. Like a human language, a programming language has a vocabulary and a grammar, but unlike a human language there is an exact meaning in the programming language for every word and sentence. Most programming languages are universal, in the same sense that Boolean logic is universal: they can be used to describe anything a computer can do. Anyone who has ever written a program—or debugged a program—knows that telling a computer what you want it to do is not as easy as it sounds. Every detail of the computer's desired operation must be precisely described. For instance, if you tell an accounting program to bill your clients for the amount that each owes, then the computer will send out a weekly bill for $0.00 to clients who owe nothing. If you tell the computer to send a threatening letter to clients who have not paid, then clients who owe nothing will receive threatening letters until they send in payments of $0.00. Avoiding this kind of misunderstanding is what computer programming is all about. The programmer's art is the art of saying exactly what you want. In this example, it means making a distinction between clients who have not sent in any money and clients who actually *owe* money. To paraphrase Mark Twain, the difference between the right program and the almost-right program is like the difference between lightning and a lightning bug—the difference is just a bug.

A skilled programmer is like a poet who can put into words those ideas that others find inexpressible. If you are a poet, you assume a certain amount of shared knowledge and experience on the part of your reader. The knowledge and experience that the programmer and the computer have in common is the meaning of the programming language. How the computer "knows" the meaning of the programming language will be described later; first, we will discuss the grammar, vocabulary, and idioms of such languages.

TALKING TO THE COMPUTER

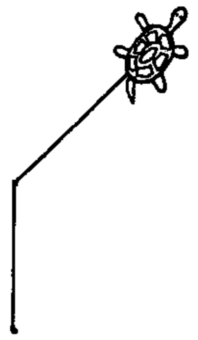
There are many different programming languages. The main reasons for this diversity are history, habit, and taste, but different programming languages also exist because they are good at describing different kinds of things. Each language has its own syntax. You need to learn the syntax in order to

write the language, but (like spelling and punctuation in a human language) syntax is not fundamental to the meaning or the expressive power of the language. What is important to the expressive power of the language is the vocabulary—the so-called *primitives* of the language—and the way the primitives can be combined to define new concepts.

Programming languages describe the manipulation of data, and one way in which these languages differ is in the kinds of data they can manipulate. The earliest computer languages were designed primarily to manipulate numbers and sequences of characters. Latter-day programming languages can manipulate words, pictures, sounds, and even other computer programs. But no matter what sort of data the language is designed to handle, it typically provides a way of reading the data's elements into the computer, taking the data apart, putting them together, modifying them, comparing them, and giving them names.

It's probably easier to illustrate these abstractions by describing a particular computer language—I've chosen Logo, which was designed by the educator and mathematician Seymour Papert as a computer language for children. Children can write programs in Logo which create and manipulate pictures, words, numbers, and sounds. Although the language is simple enough to be used by a ten-year-old, it embodies many of the features of the most sophisticated computer languages, including the ability to write programs that manipulate other programs. Logo is also an *extensible* language—that is, you can use Logo to define new words in Logo.

One of the simplest types of programs to write in Logo is a procedure for drawing a picture. You do this by giving directions to an imaginary turtle that lives on the screen. The turtle serves as a pen, moving around on the screen and leaving lines behind it. When the computer starts up, the turtle is found at the center of the screen, facing upward. If the child types the command FORWARD 10, the turtle takes ten steps forward—that is, upward—drawing a line ten units long. The



number 10 following the FORWARD command is called a *parameter;* in this case, the parameter tells the turtle how many steps to take. To draw a line in a different direction, the child must turn the turtle. The command RIGHT 45 will point the turtle 45 degrees (another parameter) to the right from its last heading. The next FORWARD command, with its parameter, will then draw a line in the new direction.

The child can use commands like FORWARD, BACKWARD, RIGHT, and LEFT to move the turtle around the screen and draw pictures, but this involves a lot of typing and soon becomes tedious. What makes the language interesting is its ability to define new words: for example, here's how the child could teach the turtle (program the computer) to draw a square:

```
TO SQUARE
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
END
```

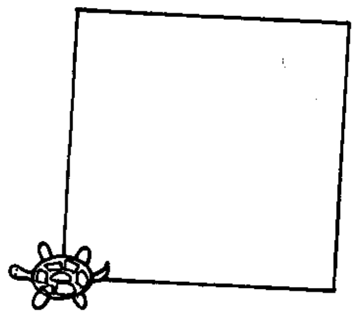


FIGURE 18

A square

Having defined the word "square," the child can then draw a square with ten units on a side simply by typing a new command: SQUARE. (Needless to say, the name "square" is arbitrary; the child could just as well have called the procedure BOX or XYZ and the turtle would do exactly the same thing. When children discover this, they often enjoy "fooling" the computer—for instance, calling the procedure that draws a square TRIANGLE, and vice versa.)

Once the word "square" has been defined, it becomes part of the computer's vocabulary and can then be used to define other words. For example,

```
TO WINDOW
SQUARE
SQUARE
SQUARE
SQUARE
END
```

Each square will be drawn in a different place, because the procedure for drawing the square leaves the turtle rotated 90 degrees. In computer terms, SQUARE is a *subroutine* of the

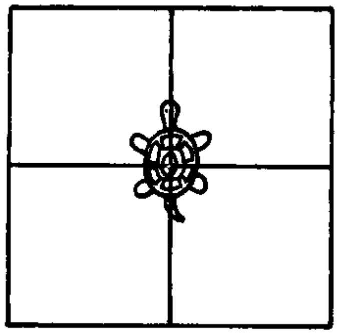


FIGURE 19

A window, made of four squares

program WINDOW, which *calls* it. The subroutine SQUARE, in turn, is defined using the *primitives* FORWARD and RIGHT. User-defined words in Logo can take parameters, too. For instance, a child can specify various sizes of square by specifying what parameter determines the length of each side.

```
TO SQUARE :SIZE
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
END
```

The colon in front of the command SIZE is an example of syntax. In Logo, the colon indicates that the word that follows is the name of a parameter, representing something else—in this case, the number to be supplied each time we "call" the subroutine named SQUARE. When SQUARE is

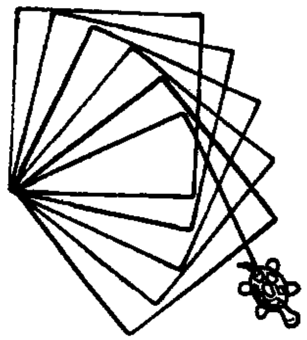


FIGURE 20

A design, drawing in progress

defined this way, the command SQUARE 15 will tell the computer to draw a square fifteen units on a side. The parameter name SIZE is, again, an arbitrary name and has meaning only within the definition of SQUARE: if all five occurrences of SIZE were replaced with X, say, the subroutine would do exactly the same thing.

There are other ways of writing a subroutine to draw a square. For instance, you can instruct the turtle to turn left four times, or to go backward four times. What's interesting is that it doesn't matter how SQUARE is defined; all that matters is what the subroutine draws and where it leaves the turtle. Other programs can call the SQUARE no matter how it is defined and whether or not it is a user-defined word or one of the language's primitives. In extending the language, the programmer uses the power of functional abstraction to create new building blocks.

One of the tricks that Logo-using children discover is that they can insert a word inside its own definition, a practice known as *recursion*. For instance, a child might generate a circular design consisting of many rotated squares, as follows:

```
TO DESIGN
SQUARE
RIGHT 10
DESIGN
END
```

The computer follows the DESIGN command by drawing a square, then turning the turtle 10 degrees and drawing another "design" in the same way. In this case, the recursive definition of "design" has a problem: it goes on forever. Each time the computer responds to the DESIGN command it draws a square, and then it must go on to another design, and on and on ad infinitum—rather as in the story of the guru who claimed that the earth was sitting on the back of (as it happens) a giant turtle. "And what is the turtle sitting on?" asked a student. "Another turtle," replied the guru. "And that turtle?" asked the student, beginning to grow skeptical. "It's no use asking," said the guru. "It's turtles all the way down."

The computer, in drawing the design, goes through the same process that human beings go through in trying to imagine the infinite stack of giant turtles, but the computer is not smart enough to notice that it's not getting anywhere. It will not halt until it is interrupted—an example of a common sort of program behavior called an *infinite loop*. Programmers often create infinite loops accidentally, and (as we shall see) it can be extremely difficult to predict when such loops will occur. This particular infinite loop is easily avoided by writing the program with a parameter specifying how many squares to draw.

```
TO DESIGN :NUMBER
SQUARE
RIGHT 10
IF :NUMBER = 1 STOP ELSE DESIGN :NUMBER –1
END
```

Thus defined, the DESIGN subroutine will do one of two things, depending on whether its parameter is 1 or some higher number. DESIGN 1 will draw just one square, but DESIGN 5, say, will draw a square, rotate, and then draw a DESIGN 4. DESIGN 4 will draw a square and then a DESIGN 3, and so on down to DESIGN 1, which will draw a square and stop.

This kind of recursive definition with a changing parameter is useful for producing anything that has a self-similar structure. A picture that contains a picture of itself is an example of a recursive, self-similar structure; such structures are commonly known as *fractals*. In the real world, self-similar structures don't go on forever: for instance, each branch of a tree looks a lot like a smaller tree, and each of these smaller branches has branches that look like still smaller trees. This recursion goes on for several levels, but eventually the branches are so small that they do not have branches of their own.

A recursive Logo program for drawing a tree is shown below. This may give you some idea of how a computer program can contain an element of poetry—although in this case the theme is somewhat obscured by the details of positioning the turtle and bringing it back to the starting point. Here's a rough translation of what the program says: "A big tree is a stick with two smaller trees on top, but a little tree is just a stick." The picture produced by the tree program is shown next to it.

```
TO TREE :SIZE
FORWARD :SIZE
IF :SIZE <1 STOP ELSE TWO-
TREES SIZE/2
BACK :SIZE
END
```

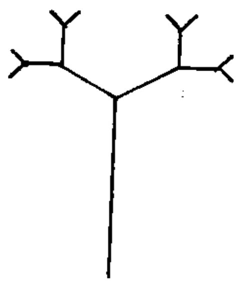


FIGURE 21

A tree

```
TO TWO-TREES :SIZE
LEFT 45
TREE :SIZE
RIGHT 90
TREE :SIZE
LEFT 45
END
```

This technique of defining things recursively turns out to be very powerful. Many of the types of data we like to manipulate—in particular, computer programs themselves—have recursive structures. Recursive definitions are extremely convenient for specifying operations on recursive data. The typical recursive definition has two parts—the first part describes what is to happen in a particular simple case, and the second describes how a more complex case can be reduced to something simpler. In the recursive tree, for example, the simple case is the tree smaller than 1 and the more complex case is the tree composed of a trunk and two small trees.

Another example is the definition of a palindrome, which we can define as follows: A word is a palindrome if it has less than two letters (the simple case) or if its first and last letter are the same and the letters in the middle form a palindrome (the recursive step). The simplest way to write a Logo program for recognizing palindromes would be to use this recursive definition.

There are many other computer languages: LISP, Ada, FORTRAN, C, ALGOL, and the like; most of the names are obscure acronyms (such as FORTRAN, for FORmula TRANslation, and LISP, for LISt Processing). Although these languages differ from Logo in details of vocabulary and syntax, they can all express the same kinds of procedures. Some, like FORTRAN, are limited in their ability to define operations recursively or to manipulate non-numerical data. Others, like C and LISP, allow the programmer to manipulate the underlying bits representing the data, which gives the programmer

more power—and more opportunity to make mistakes. In C, for example, it's perfectly possible to multiply two alphabetic characters; the result of this nonsensical operation will depend on the binary representation used by the machine. Languages like LISP offer the abstract as well as the lower-level functions. As a friend of mine, the computer scientist Guy Steele, once put it, "LISP is a high-level language, but you can still feel the bits sliding between your toes."

More recently, a new generation of languages has begun to emerge. These languages—Small-Talk, C++, Java—are *object-oriented*. They treat a data structure—for instance, a picture to be drawn on the screen—as an "object" with its own internal state, such as where it is to be drawn or what color it is. These objects can receive instructions from other objects. To understand why this is useful, imagine that you are writing a program for a video game involving bouncing balls. Each ball on the screen is defined as a different object. The program specifies rules of behavior that tell the object how to draw itself on the screen, move, bounce, and interact with other objects in the game. Each ball will exhibit similar behavior, but each will be in a slightly different state, because each will be in its own position on the screen and will have its own color, velocity, size, and so forth.

The most important advantage of an object-oriented programming language is that the objects—for instance, various objects in a video game—can be specified independently and then combined to create new programs. Writing a new object-oriented program sometimes feels a bit like throwing a bunch of animals into a cage and watching what happens. The behavior of the program *emerges*, as a result of the interactions of the programmed objects. For this reason, as well as the fact that object-oriented languages are relatively new, you might think twice about one for writing a safety-critical system that flies an airplane.

Learning a programming language is not nearly as difficult as learning a natural human language. Generally, once you have learned two or three, you can pick up others in a matter of a few hours, since the syntax is relatively simple and the vocabularies are rarely more than a few hundred words. But, as is true of a human language, there's a big difference between being able to understand the language and being able to write it well. Every computer language has its Shakespeares, and it is a joy to read their code. A well-written computer program possesses style, finesse, even humor—and a clarity that rivals the best prose.

MAKING THE CONNECTION

How can finite-state machines be used to carry out instructions written in a language like Logo? To answer this question, we go back to a more detailed level of discussion, involving Boolean logic. There are three major steps in this connection between finite-state machines and Logo: *first,* we will see how a finite-state machine can be extended, by adding a storage device called a *memory,* which will allow the machine to store the definitions of what it's asked to do; *second,* we will see how this extended machine can follow instructions written in *machine language,* a simple language that specifies the machine's operations; and *third,* we will see how machine language can instruct the machine to interpret the programming language—for instance, Logo. The rest of the chapter describes how all this works in some detail—far more detail than is strictly necessary to understand the rest of the book. The reader should not feel compelled to understand every step. The important thing to appreciate is how the layers of functional abstraction build upon one another, as is summarized in the last paragraph of the chapter.

A computer is just a special type of finite-state machine connected to a memory. The computer's memory—in effect, an array of cubbyholes for storing data—is built of *registers,*

like the registers that hold the states of finite-state machines. Each register holds a pattern of bits called a *word,* which can be read (or written) by the finite-state machine. The number of bits in a word varies from computer to computer, but in a modern microprocessor (as I write this) it is usually eight, sixteen, or thirty-two bits. (Word sizes will probably grow with improvement in technology.) A typical memory will have millions or even billions of these registers, each holding a single word. Only one of the registers in the memory is accessed at a time—that is, only the data in one of the memory registers will be read or written on each cycle of the finite-state machine. Each register in the memory has a different *address*—a pattern of bits by means of which you can access it—so registers are referred to as *locations in memory.* The memory contains Boolean logic blocks, which decode the address and select the location for reading or writing. If data are to be written at this memory location, these logic blocks store the new data into the addressed register. If the register is to be read, the logic blocks steer the data from the addressed register to the memory's output, which is connected to the input of the finite-state machine.

Some of the words stored in the memory represent data to be operated upon, like numbers and letters. Others represent *instructions* that tell the machine what sequence of operations to perform. The instructions are stored in machine language, which, as noted, is much simpler than a typical programming language. Machine language is interpreted directly by the finite-state machine. In the type of computer we will describe, each instruction in machine language is stored in a single word of memory, and a sequence of instructions is stored in a block of sequentially numbered memory locations. These sequences of machine-language instructions are the simplest kind of software within the computer.

The finite-state machine repeatedly executes the following sequence of operations: (1) *read* an instruction from the memory, (2) *execute* the operation specified by that instruc-

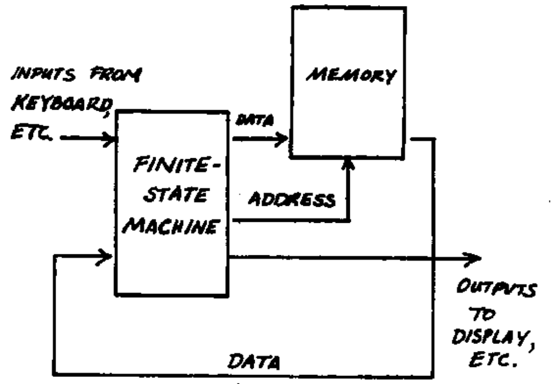


FIGURE 22

Finite-state machine connected to a memory

tion, and (3) *calculate* the address of the next instruction. The sequence of states necessary to do this is built into the Boolean logic of the machine, and the instructions themselves are specific patterns of bits—patterns that cause the finite-state machine to perform various operations on the data in the memory. For instance, the **Add** instruction is a unique pattern of bits that specifies which two registers in the memory are to be added together. Upon recognizing this pattern, the finite-state machine will go through a sequence of states that cause it to read the memory locations to be summed, add the numbers together, and write the sum back to the memory.

There are two basic types of instructions in most computers: processing instructions and control instructions. The *processing instructions* move data to and from the memory and combine them to perform arithmetic and logical functions. The addresses of the memory locations, or registers, are specified by the processing instructions. Typically, these instructions refer to only a few registers directly; other registers are referenced indirectly, because their addresses are

stored in other registers. For example, a **Move** instruction might move the data in register 1 to the address specified in register 2. If register 2 holds a pattern of bits that specifies the number 1,234, then the data will be moved to register 1,234. Other processing instructions combine data among the memory registers. There are also instructions to perform Boolean functions—**And**, **Or**, or **Invert**—on the patterns of bits in the registers.

The *control instructions* determine the address of the next instruction to be fetched; this address is stored in a special register called the *program counter*. Normally, instructions are fetched sequentially from successive memory locations, so the address in the program counter increases by 1 after each successive instruction. Control instructions allow some other number to be loaded into the program counter, thereby affecting the sequence to be executed. The simplest control instruction is the **Jump** instruction, which stores a specified address in the program counter so that the next instruction will be fetched from that address. A variation of the **Jump** instruction is a conditional **Jump**, which loads the program counter with a different address only when a specified condition has been met—such as the patterns in two registers being the same. If the condition is not met, then the conditional **Jump** will have no effect and the next instruction will be fetched sequentially.

If the same sequence of instructions needs to be executed over and over repeatedly, then a conditional **Jump** can be used at the end of the sequence to move the program counter back to the beginning as many times as is necessary. This operation is called a *loop*, and we saw an example of it in the description of programming in Logo. The execution of the sequence will be repeated until the condition on which the jump depends is no longer satisfied. If a set of instructions is to be repeated ten times, say, one of the memory registers can be used to count the number of iterations of the loop.

Exactly which instructions a computer recognizes varies from computer to computer. Computer designers can (and do) spend years arguing about what makes an optimal instruction set. A typical argument is over the comparative merits of a *reduced instruction set computer* (RISC), which uses a simple, minimal set of instructions, and a *complex instruction set computer* (CISC), which employs a rich, complex, and powerful set of instructions. This argument has little import to the programmer, however, since any reasonable instruction set can simulate any other. Historically, the commercial success of one or another sort of computer seems to have had almost nothing to do with the complexity of the instruction set or any other detail of internal design. In fact, some of the most successful computers—such as the microprocessors used in most personal computers—are generally regarded by computer designers as having poorly designed instruction sets. The details of machine design are of little importance to the users of the machines.

One reason the complexity of the instruction set doesn't much matter has to do with the subroutines. Subroutines allow sequences of instructions to be used over and over from many places within the program. In effect, the subroutine-calling convention allows the programmer to define new instructions by using sequences of other instructions. The program accesses a subroutine by using a **Jump** instruction to load the program counter with the address of the subroutine; but before doing so, the computer saves the previous contents of the program counter in a special memory location. At the end of the subroutine, another instruction reads this return address and jumps back to the location from which the subroutine was called.

This process of subroutine calling can work recursively, in the sense that subroutine sequences may have jumps to other subroutines within them, and so on. A subroutine can even call itself, in a recursively defined function. In order to keep track of this nesting of subroutines, the computer needs a systematic way of storing the return addresses, so that it will know where to return when each subroutine is completed. It cannot just store all the return addresses in the same special location, because when subroutines are nested, it needs to remember more than one return address. Normally, the computer stores return addresses in a group of sequential locations known as a *stack*. The most recent return address is stored at the "top of the stack." The memory stack works just like a stack of dinner plates: items are always added or removed from the top—a last-in, first-out storage system that works perfectly for storing the return addresses of nested subroutines, because a subroutine is never finished until all of its nested subroutines are finished.

Some subroutines are so useful that they are always loaded into the computer. This set of subroutines is called the *operating system*. Useful operating-system subroutines include those that write or read characters typed on the keyboard, or that draw lines on the screen, or otherwise interact with the user. The computer's operating system determines most of the look and feel of the interface to the user. It also governs the interface between the computer and whatever program is being run, since the operating system's subroutines provide the program with a set of operations that are richer and more complex than the machine-language instructions.

In fact, as long as the same pattern of bits produces the same effect, the programmer doesn't care whether a function is implemented by the computer hardware or by the operating-system software. The same program operating on two different types of computers might in one case perform an arithmetical operation in the hardware and in the other perform it by means of an operating-system subroutine. Similarly, the operating system of one type of computer may allow it to emulate the entire instruction set of another type of computer. Computer manufacturers sometime use such emulation to make the newer model of their computers act like the earlier models, so that older software can run without modification.

The operating system normally includes all the subroutines that perform input/output—that is, operations allowing a program to interact with the outside world. This interaction is effected by connecting certain locations in the computer's memory to input devices such as a keyboard or a mouse and output devices such as a video display terminal. For example, the space bar on the keyboard might be wired to memory register 23, so that the data read from address 23 is 1 if the space bar is pressed and 0 if it is not. Another memory register might control the color displayed on a certain dot on the screen. If every dot on the screen displays data stored in a different memory location, the computer can draw any pattern on the screen just by writing the appropriate pattern into memory.

With the exception of its input/output mechanisms, the computer we've just described is simply a finite-state machine connected to a memory. Both these elements can be constructed entirely from registers and Boolean logic blocks, using the techniques described in chapters 1 and 2. The finite-state machine that controls the computer is complicated, but it is no different in principle from the finite-state machine that controls a traffic light. Designing the machine is simply a matter of going through the details of memory data, address, and state sequences for each instruction to be executed, and then converting this state table into Boolean logic. Recall that since both the finite-state machine and the memory are made of registers and blocks of logic, both can be implemented by a number of technologies: electronics, hydraulics, sliding sticks.

TRANSLATING THE LANGUAGE

So we have established a chain of connections between the technology and the instructions. But how do the instructions execute a program written in a language—Logo, for instance—when that language is written in words and the instructions are patterns of bits? The answer is that the necessary translation is performed by the computer itself.

The translation process performed by the computer is similar to the process that a patient, meticulous human translator would use to translate a document written in an unfamiliar language, given a dictionary written in the language itself. The human translator can look up the meaning of any unknown word in the dictionary, and if words in the dictionary definition are also unknown, these words can be looked up as well. This process continues until the translator reaches a definition couched in words whose meanings are known. In this analogy, the translator's (that is, the computer's) dictionary is the program, and the words known to the computer are the aforementioned primitives of the program language. These primitives are defined directly, as simple sequences of machine instructions. For instance, when the computer looks up the definition of the Logo language primitive FORWARD, it finds the sequence of machine instructions that will draw the appropriate line on the screen.

To understand how a computer translates Logo primitives into machine language, it is helpful to understand the conventions that a computer uses to represent the Logo programs within its memory. One way to store a Logo program in a computer's memory is as a sequence of characters in adjacent memory locations, with each character being stored at a single location. In its memory, the computer keeps a directory of the addresses of the instruction sequences corresponding to each command name. This directory is stored in memory as a list of names paired with their addresses. The computer is able to find the location of an object with a given name by searching the directory for the name and finding the corresponding address. When the computer is asked to execute a particular command, it looks up the name in the directory to find out where its definition is stored.

Some of this process of looking things up and finding the corresponding sequences of machine language can be done before the program is executed. This saves time, because if the program is going to be executed more than once, there's no point in looking up the same things over and over again. When most of the work of conversion is done beforehand, the translation process is called *compilation,* and the program that performs the compilation is called a *compiler.* If most of the work is done while the program is being executed, then the process is called interpretation, and the program is called an *interpreter.* There is no hard and fast line between the two.

WELCOME TO THE HIERARCHY

We are now in a position to summarize how a computer works, from top to bottom. Most readers will have lost track of the details, but *remember that it is not important to remember how every step works!* The important thing to remember is the hierarchy of functional abstractions.

The work performed by the computer is specified by a *program,* which is written in a *programming language.* This language is converted to sequences of *machine-language* instructions by *interpreters* or *compilers,* via a predefined set of subroutines called the *operating system.* The instructions, which are stored in the *memory* of the computer, define the operations to be performed on data, which are also stored in the computer's memory. A *finite-state machine* fetches and executes these instructions. The instructions as well as the data are represented by patterns of *bits.* Both the finite-state machine and the memory are built of storage *registers* and *Boolean logic blocks,* and the latter are based on simple *logical functions,* such as *And, Or,* and *Invert.* These logical functions are implemented by *switches,* which are set up

either *in series* or *in parallel,* and these switches control a physical substance, such as water or electricity, which is used to send one of two possible signals from one switch to another: *1* or *0.* This is the hierarchy of abstraction that makes computers work.