CHAPTER 2

UNIVERSAL BUILDING BLOCKS

From now on, we can forget about wires and switches and work with the abstraction of logic blocks operating on 1's and 0's, a simple step that allows us to pass from the realm of engineering into the realm of mathematics. This is the most abstract chapter in the book; it will show you how the methods used to construct a tic-tac-toe machine can be used to construct almost any function. In it, we'll define a powerful set of building blocks: logical functions and finite-state machines. With these elements, it's easy to build a computer.

LOGICAL FUNCTIONS

In constructing the tic-tac-toe machine, we began by writing the game tree, whch gave us a set of rules for generating the outputs from the inputs. This turns out to be a generally useful method of attack. Once we write down the rules that specify what outputs we want for each combination of inputs, we can build a device that implements these rules using **And**, **Or**, and **Invert** functions. The logic blocks **And**,

Or, and **Invert** form a *universal construction set*, which can be used to implement any set of rules. (These primitive types of logic blocks are sometimes also called *logic gates*.)

This idea of a universal set of blocks is important: it means that the set is general enough to build anything. My favorite toy when I was a child was a set of interlocking plastic bricks called Lego blocks, with which I built all kinds of toys: cars, houses, spaceships, dinosaurs. I loved to play with these blocks, but they were not quite universal, since the only objects you could build with them had a certain squarish, stair-steppy look. Building something with a different shape—a cylinder or a sphere, for example—would require a new type of block. Eventually, I had to switch to another medium in order to build the things I wanted. But the **And**, **Or**, and **Invert** blocks of Boolean logic are a universal construction set for converting inputs to outputs. The best way to see how they form a universal set is to understand a general method for using them to implement rules. To start, we will consider *binary* rules—rules that specify inputs and outputs that are either 1 or 0. The tic-tac-toe machine is a good example of a function specified by binary rules, because the input switches and the output lights are either on or off—that is, either 1 or 0. (Later, we will discuss rules for handling letters, numbers, or even pictures and sounds as inputs and outputs.) Any set of binary rules can be completely specified by showing a table of the outputs for each possible combination of 1's and 0's on the inputs. For example, the rules for the **Or** function are specified by the following table:

	Input A	Input B	Output
	0	0	0
OR Function	0	1	1
	1	0	1
	1	1	1

The **Invert** function is specified by an even simpler table:

	Input	Output
Invert Function	0	1
	1	0

For a binary function with n inputs, there are 2^n possible combinations of input signals. Sometimes we won't bother to specify all of them, because we don't care about certain combinations of inputs. For example, in specifying the function performed by the tic-tac-toe machine, we don't care what happens if the human player plays in all squares simultaneously. This move would be disallowed, and we don't need to specify the function's output for this combination of inputs.

Complex logic blocks are constructed by connecting **And**, **Or**, and **Invert** blocks. In drawings of the connection pattern, the three blocks are conventionally represented by boxes of different shape (see Figure 9); the lines connecting on the left side represent inputs to the blocks, and the lines connecting on the right represent the output. Figure 10 shows how a pair of two-input **Or** blocks can be connected to form a three-input **Or** function; the output of this function will be 1 if *any one* of its three inputs is 1. It's also possible to string several **And** blocks together in a similar manner to make an **And** block with any number of inputs.

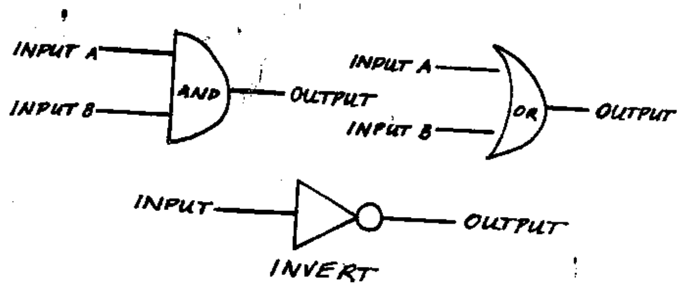

FIGURE 9

And, Or, and Invert Blocks

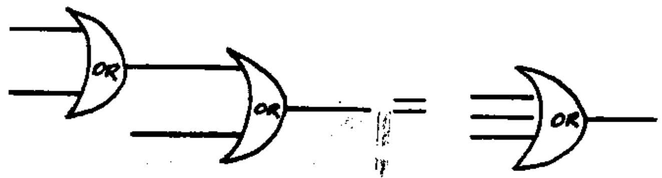

FIGURE 10

A three-input Or block made from a pair of two-input Or blocks

Figure 11 shows how an **And** block can be constructed by connecting an Inverter to the inputs and output of an **Or** block. (Here is De Morgan's theorem again.) The best way to get a feeling for how this works is to trace through the 1's and 0's for every combination of inputs. Notice that this illustration is essentially the same as Figure 6 in the previous chapter. It points up an interesting fact: we don't really need **And** blocks in our universal building set, because we can always construct them out of **Or** blocks and **Inverters.**

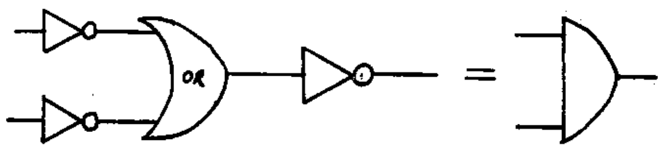

FIGURE 11

Making And **out of** Or

As in the tic-tac-toe playing machine, **And** blocks are used to detect each possible combination of inputs for which the output is 1, while **Or** blocks provide a roster of these combinations. For example, let's start with a simple function of three inputs. Imagine that we want to build a block that allows the three inputs to vote on the output. In this new

block, majority wins—that is, the output will be 1 only if two or more of the inputs are 1.

Inputs A B C	Majority Output
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

Figure 12A shows how this function is implemented.An **And** block with the appropriate **Invert** blocks as input is used to recognize each combination of inputs for which the output is 1; these blocks are connected by an **Or** block, which produces the output. This strategy can be used to create any transformation of inputs to outputs:

Of course, this particular method of using a separate **And** gate to recognize each combination of inputs is not the only way to implement the function, and it is often not the simplest way. Figure 12B shows a simpler way to produce the majority function. The great thing about the method described is not that it produces the best implementation but that it always produces an implementation that works. The important conclusion to draw is that it is possible to combine **And, Or,** and **Invert** blocks to implement *any* binary function—that is, any function that can be specified by an input/output table of 0's and 1's.

Restricting the inputs and output to binary numbers is not really much of a restriction, because the combinations of 1's and 0's can be used to represent other things—letters,

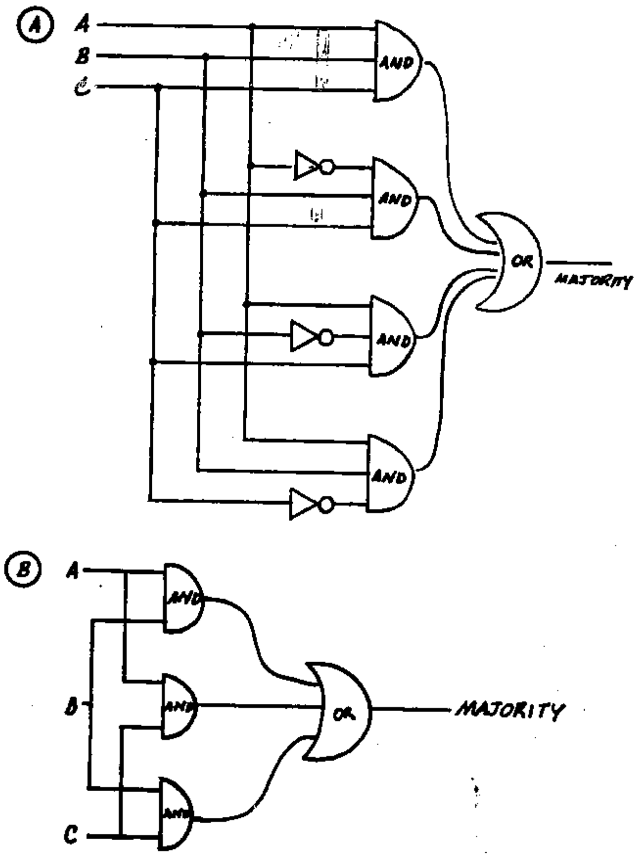

FIGURE 12

How the voting function is implemented by And, Or, and Invert Blocks

larger numbers, any entity that can be encoded. As an example of a nonbinary function, suppose we want to build a machine to act as a judge of the children's game of Scissors/Paper/Rock. This is a game for two players in which each chooses, in secret, one of three "weapons"—scissors,

paper, or rock. The rules are simple: scissors cuts paper, paper covers rock, rock crushes scissors. If the two children choose the same weapon, they tie. Rather than building a machine that plays the game (which would involve guessing which weapon the opponent is going to choose), we will build a machine that judges who wins. Here's the input/output table for the function that takes the choices as inputs and declares the winner as output. The table encodes the rules of the game:

Input A	**Input B**	**Output**
Scissors	Scissors	Tie
Scissors	Paper	A wins
Scissors	Rock	B wins
Paper	Scissors	B wins
Paper	Paper	Tie
Paper	Rock	A wins
Rock	Scissors	A wins
Rock	Paper	B wins
Rock	Rock	Tie

The Scissors-Paper-Rock judging function is a combinational function, but it is not a binary function, since its inputs and output have more than two possible values. To implement this function as a combinational logic block, we must convert it to a function of 1's and 0's. This requires us to establish some convention for representing the inputs and outputs. A simple way to do this would be to use a separate bit for each of the possibilities. There would be three input signals for each weapon: a 1 on the first input represents **Scissors**, a 1 on the second input represents **Rock**, and a 1 on the third input represents **Paper**. Similarly, we could use separate output lines to represent a win for player A, a win for player B, or a tie. So the box would have six inputs and three outputs.

Using three input signals for each weapon is a perfectly good way to build the function, but if we were doing it inside a computer we would probably use some kind of encoding that required a smaller number of inputs and outputs. For example, we could use two bits for each input and use the combination 01 to represent **Scissors**, 10 to represent **Paper**, and 11 to represent **Rock**. We could similarly encode each of the possible outputs using two bits. This encoding would result in the simpler three-input/two-output table shown below:

	A Inputs	B Inputs	Outputs
	01	01	00
	01	10	10
Scissors = 01	01	11	01
Paper = 10	10	01	01
Rock = 11	10	10	00
A wins = 10	10	11	10
B wins = 01	11	01	10
Tie = 00	11	10	01
	11	11	00

Computers can use combinations of bits to represent anything; the number of bits depends on the number of messages that need to be distinguished. Imagine, for example, a computer that works with the letters of the alphabet. Five-bit input signals can represent thirty-two different possibilities ($2^5 = 32$). Functions within the computer that work on letters sometimes use such a code, although they more often use an encoding with seven or eight bits, to allow representation of capitals, punctuation marks, numerals, and so on. Most modern computers use the standard representation of alphabet letters called ASCII (an acronym for American Standard Code for Information Interchange). In ASCII, the sequence 1000001 represents the capital letter **A**, and 1000010 represents the capital **B**, and so on. The convention, of course, is arbitrary.

Most computers have one or more conventions for representing numbers. One of the most common is the *base 2* representation of numbers, in which the bit sequence 0000000 represents zero, the sequence 0000001 represents the number 1, the sequence 0000010 represents 2, and so on. The description of computers as "64-bit" or "32-bit" indicates the number of bit positions in the representation used by the computer's circuits: a 32-bit computer uses a combination of thirty-two bits to represent a base-2 number. The base-2 number system is a common convention, but there is nothing that requires its use. Some computers don't use it at all, and most computers that do also represent numbers in other ways for various purposes. For instance, many computers use a slightly different convention for representing negative numbers and also have a convention called a *floating point* to represent numbers that have decimal points. (The position of the decimal point "floats" relative to the digits, so that a fixed number of digits can be used to represent a wide range of numbers.) The particular representation schemes are often chosen in such a way as to simplify the logic of the circuits that perform arithmetical operations, or to make it easy to convert from one representation to another.

Because any logical function can be implemented as a Boolean logic block, it is possible to build blocks that perform arithmetical operations like addition or multiplication by using numbers with any sort of representation. For instance, imagine that we want to build a functional block that will add numbers on an eight-bit computer. An eight-bit adder block must have sixteen input signals (eight for each of the numbers to be added), and eight output signals for the sum. Since each number is represented by eight bits, there are 256 possible combinations, and each can represent a different number. For example, we could use these combinations to represent the numbers between 0 and 255, or between −100 and +154. Defining the function of the block would be just a matter of writing down the addition table and then converting it to 1's and 0's, using the chosen representation. The table of 1's and 0's could then be converted to **And** and **Or** blocks by the methods described above.

By adding two more inputs to the block, we could use similar techniques to build a block that not only adds but also subtracts, multiplies, and divides. The two extra control inputs would specify which of these operations was to take place. For instance, on every line of the table where the control inputs were 01, we would specify the output to be the sum of the input numbers, whereas in every combination where the control inputs were 10, we would specify the outputs to be the product, and so on. Most computers have logical blocks of this type inside them called *arithmetic units.*

Combining **Ands** and **Ors** according to this strategy is one way to build any logical function, but it is not always the most efficient way. Often, by clever design, you can implement a circuit using far fewer building blocks than the preceding strategy requires. It may also be desirable to use other types of building blocks or to design circuits that minimize the delay from input to output. Here are some typical puzzles in logic design: How do you use **And** blocks and **Inverters** to construct **Or** blocks? (Easy.) How do you use a collection of **And** and **Or** blocks, plus only two **Inverters,** to construct the function of three **Inverters**? (Hard, but possible.) Puzzles like this come up in the course of designing a computer, which is part of what makes the process fun.

FINITE-STATE MACHINES

The methods I've described can be used to implement any function that stays constant in time, but a more interesting class of functions are those that involve sequences in time. To handle such functions, we use a device called a *finite-state machine.* Finite-state machines can be used to implement time-varying functions—functions that depend not just on the current input but also on the previous history of inputs. Once you learn to recognize a finite-state machine, you'll notice

them everywhere—in combination locks, ballpoint pens, even legal contracts. The basic idea of a finite-state machine is to combine a look-up table, constructed using Boolean logic, with a memory device. The memory is used to store a summary of the past, which is the *state* of the finite-state machine.

A combination lock is a simple example of a finite-state machine. The state of a combination lock is a summary of the sequence of numbers dialed into the lock. The lock doesn't remember all the numbers that have ever been dialed into it, but it does remember enough about the most recent numbers to know when they form the sequence that will open the lock. An even simpler example of a finite-state machine is the retractable ballpoint pen. This finite-state machine has two possible states—extended and retracted—and the pen remembers whether its button has been pressed an odd or an even number of times. All finite-state machines have a fixed set of possible states, a set of allowable inputs that change the state (clicking a pen's button, or dialing a number into a combination lock), and a set of possible outputs (retracting or extending the ballpoint, opening the lock). The outputs depend only on the state, which in turn depends only on the history of the sequence of inputs.

Another simple example of a finite-state machine is a counter, such as the tally counter on a turnstile indicating the number of people who have passed through. Each time a new person goes through, the counter's state is advanced by one. The counter is a *finite* state because it can only count up to a certain number of digits. When it reaches its maximum count—say, 999—the next advance will cause it to return to zero. Odometers on automobiles work like this. I once drove an old Checker cab with an odometer that read 70,000, but I never knew if the cab had traveled 70,000 miles, 170,000 miles, or 270,000 miles, because the odometer had only 100,000 states; all those histories were equivalent as far as the odometer was concerned. This is why mathematicians often define a state as "a set of equivalent histories."

Other familiar examples of finite-state machines include traffic lights and elevator-button panels. In these machines, the sequence of states is controlled by some combination of an internal clock and input buttons such as the "Walk" button at the crosswalk and the elevator call and floor-selection buttons. The next state of the machine depends not only on the previous state but also on the signals that come from the input button. The transition from one state to another is determined by a fixed set of rules, which can be summarized by a simple state diagram showing the transition between states. Figure 13 shows a state diagram for a traffic-light

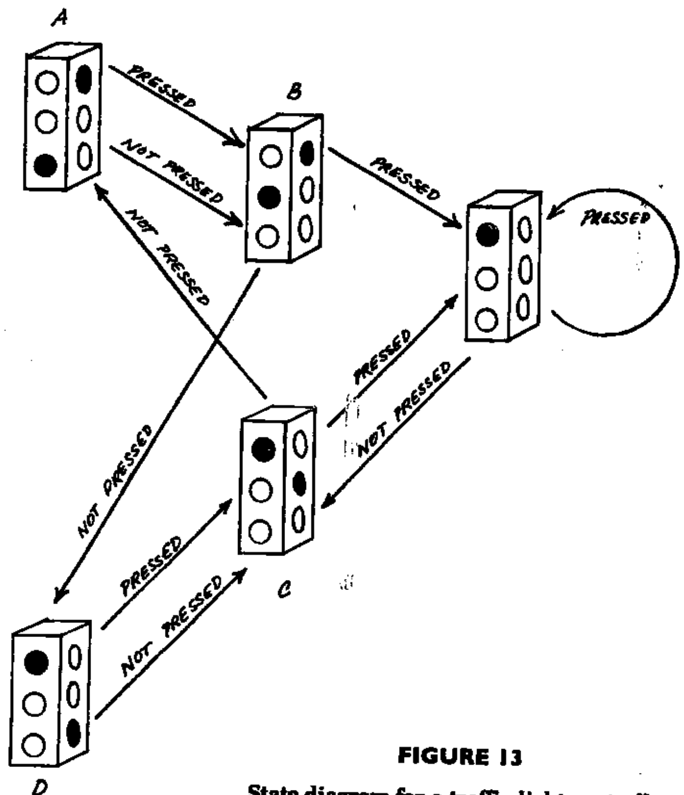

FIGURE 13

State diagram for a traffic-light controller

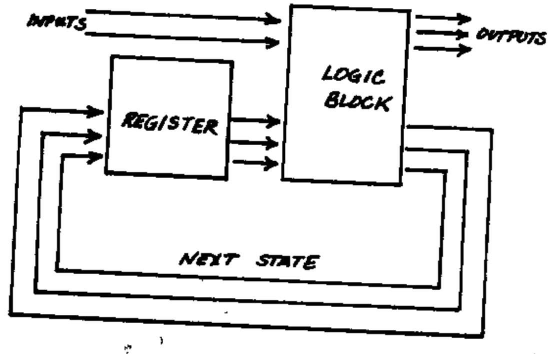

FIGURE 14

Finite-state machine, with logic block feeding register

controller at an intersection where the light turns red in both directions after the Walk button is pressed. Each drawing of light represents a state and each arrow represents a transition between states. The transition depends on whether or not the "walk" button is pressed.

To store the state of the finite-state machine, we need to introduce one last building block—a device called a *register*, which can be used to store bits. An *n*-bit register has *n* inputs and *n* outputs, plus an additional timing input that tells the register when to change state. Storing new information is called "writing" the state of the register. When the timing signal tells the register to write a new state, the register changes its state to match the inputs. The outputs of the register always indicate its current state. Registers can be implemented in many ways, one of which is to use a Boolean logic block to steer the state information around in a circle. This type of register is often used in electronic computers, which is why they lose track of what they're doing if their power is interrupted.

A finite-state machine consists of a Boolean logic block connected to a register, as shown in Figure 14. The finite-state machine advances its state by writing the output of the

Boolean logic block into the register; the logic block then computes the next state, based on the input and the current state. This next state is then written into the register on the next cycle. The process repeats in every cycle.

The function of a finite-state machine can be specified by a table that shows, for every state and every input, the state that follows. For example, we can summarize the operation of the traffic-light controller by the following table:

Inputs:			*Outputs:*	
Walk Button	**Current State**	**Main Road**	**Cross Road**	**Next State**
Not Pressed	A	Red	Green	B
Not Pressed	B	Red	Yellow	D
Not Pressed	C	Yellow	Red	A
Not Pressed	D	Green	Red	C
Not Pressed	Walk	Walk	Walk	D
Pressed	A	Red	Green	B
Pressed	B	Red	Yellow	Walk
Pressed	C	Yellow	Red	Walk
Pressed	D	Green	Red	C
Pressed	Walk	Walk	Walk	Walk

The first step in implementing a finite-state machine is to generate such a table. The second step is to assign a different pattern of bits to each state. The five states of the traffic-light controller will require three bits. (Since each bit doubles the number of possible patterns, it is possible to store up to 2^n states using n bits.) By consistently replacing each word in the preceding table with a binary pattern, we can convert the table to a function that can be implemented with Boolean logic.

In the traffic-light system, a timer controls the writing of the register, which causes the state to change at regular intervals. Another example of a finite-state machine that advances its state at regular intervals is a digital clock. A digital clock with a seconds indicator can be in one of 24 x 60 x 60 = 86,400

possible display states—one for each second of the day. The timing mechanism within the clock causes it to advance its state exactly once per second. Many other types of digital computing devices, including most general-purpose computers, also advance their state at regular intervals, and the rate at which they advance is called the *clock rate* of the machine. Within a computer, time is not a continuous flow but a fixed sequence of transitions between states. The clock rate of the computer determines the rate of these transitions, hence the correspondence between physical and computational time. For instance, the laptop computer on which I am writing this book has a clock rate of 33 megahertz, which means that it advances its state at a rate of 33 million times per second. The computer would be faster if the clock rate were higher, but its speed is limited by the time required for information to propagate through the logic blocks to compute the next state. As technology improves, the logic tends to become faster and the clock rate increases. As I write these words, my computer is state-of-the-art, but by the time you read this book computers with 33 megahertz clock rates will probably be considered slow. This is one of the wonders of silicon technology: as we learn to make computers smaller and smaller, the logic becomes faster and faster.

One reason finite-state machines are so useful is that they can recognize sequences. Consider a combination lock that opens only when it is given the sequence 0–5–2. Such a lock, whether it is mechanical or electronic, is a finite-state machine with the state diagram shown in Figure 15.

A similar machine can be constructed to recognize any finite sequence. Finite-state machines can also be made to recognize sequences that match certain patterns. Figure 16 shows one that recognizes any sequence starting with a 1, followed by a sequence of any number of 0s, followed by a 3. Such a combination will unlock the door with the combination 1–0–3, or a combination such as 1–0–0–0–3, but *not* with the combination 1–0–2–3, which doesn't fit the pattern. A more complex

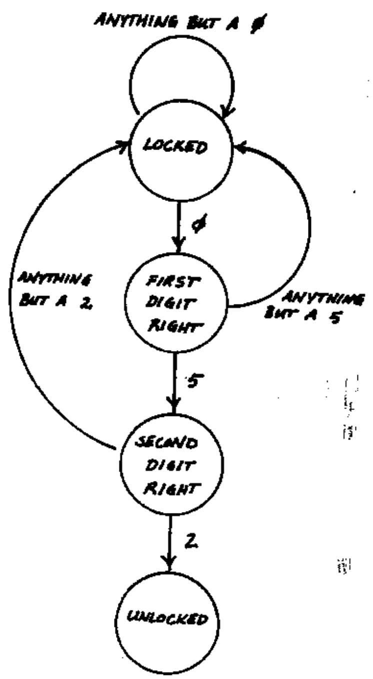

FIGURE 15

State diagram for a lock with combination 0-5-2

finite-state machine could recognize a more complicated pattern, such as a misspelled word within a stream of text.

As powerful as they are, finite-state machines are not capable of recognizing all types of patterns in a sequence. For instance, it is impossible to build a finite-state machine that will unlock a lock whenever you enter any palindrome—a sequence that is the same forward and backward, like 3–2–1–1–2–3. This is because palindromes can be of any length, and to recognize the second half of a palindrome you need to remember every character in the first half. Since there are infinitely many possible first halves, this would require a machine with an infinite number of states.

A similar argument demonstrates the impossibility of building a finite-state machine that recognizes whether a given English sentence is grammatically correct. Consider the simple sentence "Dogs bite." The meaning of this sentence can be changed by putting a qualifier between the noun and the verb; for instance, "Dogs that people annoy bite." This sentence can in turn be modified by putting another phrase in the middle: "Dogs that people with dogs annoy bite." Although the meaning of such sentences might be expressed more clearly, and although they become increasingly difficult to understand, they are grammatically correct. In principle, this process of nesting phrases inside of one another can go on forever, producing absurd sentences like "Dogs that dogs that dogs that dogs annoy ate bit bite." Recognizing such a sentence as grammatically correct is impossible for a finite-state machine, and for exactly the same reason it's difficult for a person: you need a lot of memory to keep track of all those dogs. The fact that human beings seem to have trouble with the same kinds of sentences that stump finite-state machines has caused some people to speculate that we may have something like a finite-state machine inside our head for understanding language. As you will see in the next chapter, there are other types of computing devices that seem to fit even more naturally with the recursive structure of human grammar.

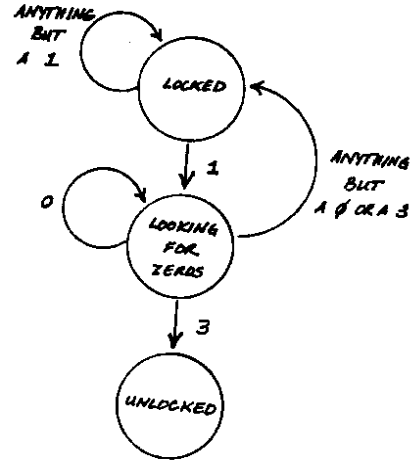

FIGURE 16

State diagram to recognize sequences like 1,0,3 and 1,0,0,0,3

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