7 Computation

In the last chapter, we saw how one could perform computations on symbols. The descriptions of the procedures had elements (themes) in common but did not put forth a common framework. They were ad-hoc and informally presented. For example, the procedure for addition involved sub-procedures that were not specified: How does one *determine* if there are two bits in a solution? How does one *find* the top column? How does one *know* which column is the "current" column? The procedures hinted at the possibility of physical instantiation, but they left doubts as to whether they could ultimately be implemented by a purely mechanical device.

And if they could be implemented, are we to believe that the brain would be composed of ever more such elaborate devices to accomplish its diversity and complexity of tasks? Put another way, is the brain a Rube Goldberg contraption where each component is a uniquely crafted physical mechanism designed to solve a particular idiosyncratic problem? Very unlikely. Wherever one finds complexity, simplicity is sure to follow.[1] It is a ubiquitous property of complex systems, both natural and engineered, that they are built upon simpler building blocks. And, this property tends to be hierarchical in nature. That is, the simpler building blocks themselves are often constructed from even simpler building blocks.

Could a single conceptual framework handle the many different computational problems that one could face? We saw that a set of primitive data, through combinatorial interaction, could serve all of our symbolic needs. Is there a similar set of computational primitives that can serve all of our computational needs? In the last chapter we gained an intuitive understanding of the nature of procedures that perform computations on symbols. Such concepts have existed for a long time; the word algorithm derives from the Persian Mathematician al-Khwarizmi who lived during the ninth century AD. Formalizing the concept of a procedure and what type of machine could implement these procedures without human aid, however, had to wait until the twentieth century.

[1] Credit to Perlis (1982): "Simplicity does not precede complexity, but follows it."

Formalizing Procedures

Twelve years before Shannon published his paper laying the foundations of information theory, Alan Turing published his paper (Turing, 1936) laying the foundations of the modern understanding of computation. Turing started from the intuition that we know how to compute something if we have a step-by-step recipe that, when carefully followed, will yield the answer we seek. Such a recipe is what we have called a *procedure*. The notion of a procedure was important for those working on the foundations of mathematics, because it was closely connected to an understanding of what constitutes a rigorous proof. Anyone who has scrutinized a complex and lengthy proof is aware of how difficult it is to be sure that there is not a cheat somewhere in it – a step that is taken that does not follow from the preceding steps according to a set of agreed upon rules.

Turing's enterprise, like Shannon's, was a mathematical one. He did not intend nor attempt to build an actual machine. Turing wanted to specify the elements out of which any procedure could be constructed in such an elementary and precise manner that there could be no doubt that each element (each basic step) could be executed by a mindless machine. The intuition here is that a machine cannot cheat, cannot deceive itself, whereas our minds do routinely deceive themselves about the cogency of their reasoning. Thus, he faced a twofold challenge: first, to specify some very simple operations that could obviously be implemented on a machine; and second, and by far the greater challenge, to make the case that those operations sufficed to construct any possible procedure, and were capable of performing all possible computations.

So far as we know, Turing succeeded. He created a formalization that defined a class of machines. The elements from which these machines were constructed were of such stark simplicity that it was clear that they were realizable in physical form. The machines that are members of this class are now referred to as *Turing machines*, and to date, the formalism has withstood any attempt to find a procedure that could not be identified with a Turing machine. By mathematically specifying the nature of these machines, and demonstrating their far-reaching capabilities, he laid a rigorous foundation for our understanding of what it means to say something is computable: Something is computable if it can be computed by a Turing machine. Part of the fascination of his work is that it showed that some perfectly well-defined functions were *not* computable. Thus, his formulation of what was computable had teeth; it led to the conclusion that some functions were computable and some were not. His work was closely related to and inspired by the slightly earlier work of Kurt Gödel, showing that there are (and always will be) perfectly well-formed formulas in arithmetic that cannot be proved either to be true or false using "finitistic" proof methods – despite the fact that, by inspection, the statements in question must in fact be true. Finitistic proofs do not make use of any steps that involve reasoning about the infinite, because mathematicians had come to mistrust their intuitions about what were and were not legitimate steps in reasoning about the infinite.

We say that Turing succeeded "so far as we know," because his success cannot be proven. He was trying to make our notion of what is and is not computable rigorous. Since the notion of being computable is not itself rigorous, how can one say whether a rigorous formulation fully captures it? One could, however, show that he had failed by specifying a computation that a human (or machine), yet not a Turing machine, could carry out. In this regard, Turing's formulation has stood the test of time. Almost 70 years have passed since he published his formulation. In that time, there has been a dizzying development of computing machines and intense study of computation and its foundations by engineers, logicians, and mathematicians. So far, no one has identified a computation that we – or any other known thing – can do that no Turing machine can. We have developed computations of a complexity undreamed of in Turing's day, but they can all be done by a Turing machine. In fact, they all are done on modern computers, which are examples of *universal Turing machines* – Turing machines that can emulate any other Turing machine. This does not necessarily mean that we will never discover such a computation. Brains solve a number of computational problems that we do not currently know how to program a computer to solve – face recognition for example. Perhaps brains can do computations that a Turing machine cannot do. But, if so, we have no clue as to how they do it. Nothing we currently understand about computation in the brain presents any challenge to a Turing machine. In fact, all current formally specified models of what goes on in brains *are* implemented on contemporary computers.

The thesis that a Turing machine can compute anything that is computable is now called the Church-Turing thesis, because Alonzo Church, Turing's thesis advisor, developed a closely related logical formalism, the lambda calculus, intended, like Turing's, to formalize the notion of a procedure. Church's work was done more or less simultaneously with Turing's and before Turing became his graduate student. In fact, Turing went from Cambridge to Princeton to work with Church when he discovered that they were both working on the same problem. That problem was Hilbert's *Entscheidungsproblem* (decision problem), the problem of whether there could be a procedure for deciding whether a proposition in arithmetic was provable or not – not for proving it, just for deciding whether it was provable. The conclusion of both Church and Turing's work was that there cannot be such a procedure. That is, the decision problem was uncomputable. A closely related result in Turing's work is that there cannot be a computer program (procedure) for deciding whether another computer program will eventually produce a result (right or wrong) for a given input. This is called the *halting problem*. His result does not mean that the halting problem cannot be solved for particular programs and inputs. For simple programs, it often can. (For example, if the first thing a Turing machine does is to halt, regardless of the input, then there is no question that it halts for all inputs.) What the result means is that given *any* possible pair consisting of a (description of a) Turing machine and an input to be presented to that machine, no procedure can *always* determine if the given Turing machine would halt on the given input.

Both Gödel and Church believed that Turing's machines were the most natural formalization of our intuitive notions of computation. This belief has been borne out, as Turing's machines have had by far the greatest influence on our understanding of computation. Turing's work and the vast body of subsequent work by others that rests on it focuses on *theoretical computability*. It is not concerned to specify the architecture of a practical computing machine. Nor is it concerned to distinguish between problems that are in principle computable, but in practice not.

The Turing Machine

A Turing machine has three basic functional components: a long "tape" (the symbolic memory), a read/write head (the interface to the symbolic memory), and a finite-state processor (the computational machinery) that essentially runs the show.

The tape. Each Turing machine (being itself a procedure) implements a function that maps from symbols to symbols. It receives the input symbol(s) as a data string that appears on a *tape*. Turing's tape serves as the symbolic memory, the input, and the output for each procedure. The input symbol(s) get placed on the tape, the procedure is run, and the resulting output is left on the tape. He had in mind the paper tapes found in many factories, where they controlled automated machinery – the head of the teletype machine stepping along the tape at discrete intervals. Similarly, Turing's tape is divided into discrete (digital) cells. Each cell can hold exactly one of a finite number of (digital) atomic data. Successive cells of such data can thereby create the data strings that are the foundation of complex symbols. Turing imagined the tape to be infinitely long, which is to say, however long it had to be to accommodate a computation that ended after some finite number of steps. Turing did not want computations limited by trivial practical considerations, like whether the tape was long enough. This is equivalent to assuming that the machine has as much symbolic memory as it needs for the problem at hand. He also assumed that the machine had as much time as it needed. He did not want it to be limited by essentially arbitrary (and potentially remediable) restrictions on its memory capacity, its operating speed, or the time allowed it.

The number of data (the elements from which all symbols must be constructed) used for a particular Turing machine is part of the description of that machine. While one can get by (using sub-encoding schemes) using just two atomic data (ostensibly '0' and '1'), it is often easier to design and understand Turing machines by using more atomic data. We will start by using these two atomic data along with a "blank" symbol (denoted '•') that will be the datum that appears on all cells that have never been written to. We will add more atomic data if needed to make our examples clear, keeping in mind that the machines could be redesigned to use only two atomic data. Thinking toward potential physical instantiation, one could imagine that the atomic data are realized by making the tape a magnetic medium and that each cell can contain a distinguishable magnetic pattern.

As previously noted, we enclose symbols such as '1', '0', and '•' in single quotes to emphasize that they are to be regarded as purely arbitrary symbols (really data), having no more intrinsic reference than magnetic patterns. In particular, they are not to be taken to represent the numbers 0 and 1. In fact, in the example we will give shortly, a single '1' represents the number 0, while the number 1 is represented by '11'. This, while no doubt confusing at first, is deliberate. It forces the reader over and over again to distinguish between the symbol '1' and the number 1, which may be represented by '1' or may just as well be represented by any other arbitrary symbol we may care to choose, such as '11' or '≈' or whatever else you fancy in the way of a symbol.

The symbols are simply a means of distinguishing between different messages, just as we use numbers on jerseys to distinguish between different players on an athletic team. The messages are what the symbols refer to. For many purposes, we need not consider what those messages are, because they have no effect on how a Turing machine operates. The machine does not know what messages the symbols it is reading and writing designate (refer to). This does not mean, however, that there is no relation between how the machine operates and what the symbols it operates on refer to. On the contrary, we structure the operation of different Turing machines with the reference of the symbols very much in mind, because we want what the machine does to make functional sense.

The read/write head. The Turing machine has a head that at any given time is placed in one of the cells of the tape. The head of the machine can both read the symbol written in a cell and write a symbol to it. Turing did not say the head "read" the cell, he said it "scanned" it, which is in a way a more modern and machine-like term in this age in which digital scanners are used at every check-out counter to read the bar codes that are the symbols of modern commerce. The head can also be moved either to the left or the right on the tape. This allows the machine to potentially read from and write to any cell on the tape. In effect, the read/write head can be thought of functionally as an all-in-one input transducer, output transducer, and mechanism to access and alter the symbolic memory.

The processor. What the head writes and in what direction the head moves is determined by the processor. It has a finite number of discrete processing states. A state is the operative structure of the machine; it determines the processor's response to the symbol the head reads on the tape. As a function of what state the processor is in, and what symbol is currently being read, the processor directs the head regarding what symbol to write (possibly none) and what move to make (possibly no move). The processor then also activates the next state. The finitude of the number of possible states is critical. If the number of states were infinite, it would not be a physically realizable machine. In practice, the number of states is often modest. The states are typically represented (for us!) by what is called a transition table. This table defines a particular Turing machine.

It is important to realize that allowing the tape (memory – the supply of potential symbols) to expand indefinitely is not the same as allowing the number of states of the machine to expand without limit. The tape cells are initially all "empty" (which we indicate by the '•' symbol), that is, every cell is just like every other. The tape has no pre-specified structure other than its uniform topology – it carries no information. It has only the capacity to record information and carry it forward in time in a computationally accessible manner. It records when it is written to and it gives back previously recorded information when it is read. As we explained when discussing compact symbols, a modest stretch of tape has the potential to symbolize any of an infinite set of different entities or states of the world. By contrast, each state of the machine is distinct from each other state, and its structure is specific to a specific state of affairs (pre-specified). This distinction is critical, for otherwise, as it is often claimed, one couldn't build an actual Turing machine. Such is the case only if one must deal with unbounded input, a situation that would never be presented to an actual machine. Arguments that the brain can't be a Turing machine (due to its infinite tape) but instead must be a weaker computational formalism are spurious – what requires the Turing machine architecture is not an issue of unbounded memory, it is an issue of being able to create compact procedures with compact symbols.

Our machine specifications denote one state as the *start state*, the state that the machine is in when it begins a new computation. There is also a special state called the halt state. When the machine enters this state, the computation is considered to be complete. When the machine begins to compute, it is assumed that the read/write head is reading the first datum of the first symbol that constitutes the input. Following tradition, we start our machines on the leftmost symbol. When the machine enters a halt state, the read/write head should be reading the first (leftmost) datum of the output.

The response of the machine in a given state to the read symbol has three components: what to write to the tape, which way to move (right or left or no move), and which state to then enter (transition to).

- *Writing*. The Turing machine can write any of the atomic data to a cell. We also allow the Turing machine not to write at all, in which case it simply leaves the tape as is.
- *Moving*. After it has written (or not written), the machine can move to the left one cell or it can move to right one cell. It may also choose to stay in its current position (not move).
- *Transitioning (changing state)*. After writing and moving, the machine changes (transitions) from its current state to another (possibly the same) state.

That the machines thus described were constructible was obvious in Turing's original formulation. Turing also needed to show his machines could compute a wide variety of functions. Turing's general strategy was to devise transition tables that implemented the elementary arithmetic operations of addition, subtraction, multiplication, division, and ordering.[2] All of mathematics rests ultimately on the foundation provided by arithmetic. Put another way, any computation can be reduced to the elementary operations of arithmetic, as can text-processing computations, etc. – computations that do not appear to be arithmetical in nature.

[2] Turing's landmark paper actually achieved four major results in mathematics and theoretical computation. He formalized the notion of a procedure, he demonstrated that the decision problem was undecidable, he demonstrated the existence of universal Turing machines (universal procedures), and he discovered the class of numbers referred to as computable numbers. Our immediate concern is the formalization of the notion of a procedure.

FEB-05-2015 12:19 P.007

Table 7.1 State transition table for the successor machine

State	*Read*	*Write*	*Move*	*Next state*
S_{start}	'1'	none	L	S_{start}
S_{start}	'•'	'1'	none	S_{halt}

Turing Machine for the Successor Function

Let us follow along Turing's path, by considering machines that compute the functions f_{is_even} and f_+ from the previous chapter. Before we do this, however, we will show an even more basic example – a Turing machine that simply computes the successor function on a unary encoding, that is, it adds one to an integer to generate the next integer. This machine, starting from zero, can generate each integer, one after another, by composing its previous result with itself. As usual, before creating our procedure we must settle on a code that we will use for the integers. In this case we will use the unary code, which will make for a particularly simple transition table and a good introduction to the machines. As we already indicated, we will let '1' be the symbol for the number 0 – perverse as that may seem. We will use this scheme for each unary example that we give. The symbol for the number n will be a string of $n + 1$ '1's. so '11' is the symbol for 1; '111', the symbol for 2; '1111', the symbol for 3, and so on. Each symbol for a number has one more '1' than the number it is a symbol for.

We will start our machine off with an input of zero. From this, we can repeatedly run it to generate successive integers. In the beginning, the read/write head of our Turing machine will be reading the cell containing the symbol '1'. All the other cells contain the blank symbol ('•'). Therefore the input is zero. This machine only demands two states (S_{start}, S_{halt}). Table 7.1 shows the transition table for this machine. The first column contains the state the machine may be in (in this case only S_{start} – the halt state need not be included as it reads no input and does nothing). The other columns contain the data that may be read by the head (in this case only '1' and '•'). Each entry signifies, given this combination of state and symbol read, what symbol to write ('1' or '•'), what move to make (L, R, none), and what state to transition to next.

Typically, such tables are shown pictorially in the form of a *state diagram*, which tends to be easier to follow. Table 7.1 would be transcribed into this format as shown in Figure 7.1. Here, the states are shown as circles. The transitions that occur from state to state are shown by arrows going from state to state. Each arrow coming out of a state is annotated first by the symbol that when read causes that transition, second by the symbol that is written, and third by the move of the read/write head.

When the procedure starts, the tape will look like '. . . • • $\boxed{1}$ • • . . .'. The box surrounding a symbol indicates the position of the read/write head. When the computation is finished, the tape looks like '. . . • • $\boxed{1}$ 1 • • . . .'. The machine starts in

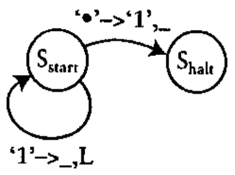

Figure 7.1 State diagram for the successor machine. The states are represented by circles, with the transitions shown by arrows. The arrows are annotated. The annotation shows what the machine read, followed by an arrow, followed by what it wrote, and, following a comma, how it moved the head (L = left one cell, R = right one cell). If it wrote nothing or did not move, there is a _.

state S_{start}. In this state, reading a '1' causes it to move the head to the left one cell, without writing anything, and remain in the S_{start} state. Now, having moved one cell to the left, it reads a '•' (a blank). Reading a blank when in the S_{start} state causes it to write a '1', and enter S_{halt}, ending the computation. The procedure implements what we might describe in English as "Put a '1' before the first symbol" – however, nothing has been left to interpretation. We don't need to invoke a homunculus that understands "put", "before," or "first symbol." One cannot help but be astonished by the simplicity of this formulation.

If we run the machine again, it will end up with '111' written on the tape, which is our symbol for 2. If we run it again, we get '1111', our symbol for 3, and so on. We have created a machine that carries out (computes) the successor function; each time it is run it gives our symbol for the number that is the successor (next number) of the number whose symbol is on the tape when we start the machine.

Turing Machines for f_{is_even}

Next we consider Turing machine formulations of the parity function (predicate) f_{is_even}: $D^{\circledast} \rightarrow \{0, 1\}$, that maps a string of bits to '1' if the input bits encode for an even number and '0' otherwise. A compact approach would use a compact procedure with compact symbols, however, the Turing machine is certainly capable of implementing a compact procedure for f_{is_even} on a non-compact representation (unary), and a non-compact procedure (look-up table) on compact nominal symbols (binary strings).

We first implement f_{is_even} using the unary encoding scheme from above (in which a single '1' encodes for 0). Figure 7.2 gives the state diagram for our procedure. Table 7.2 shows the transition table for this same procedure.

This machine steps through each '1', erasing them as it goes. It shifts back and forth between states S_{start} and S_1. The state it is in contains implicit (non-symbolic) knowledge of whether it has read an odd or even number of '1's; if it has read an odd number of '1's, it is in S_1. Given our unary encoding of the integers in which '1' refers to 0, '11' to 1, '111' to 2, and so on, this implies even parity. This is an example of an appropriate use of state memory. As it moves along the data string,

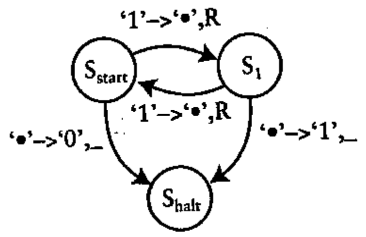

Figure 7.2 State diagram for the machine that computes the parity of integers represented by unary symbols. The machine is in S_{start} when it has read a sequence of '1's that encodes an odd integer; it is in S_1 when it has read a sequence encoding an even integer. If it reads a blank while in S_{start} (indicating that it has come to the end of the symbol string), it writes a '0' (the symbol for odd parity) and enters S_{halt}; if it reads a blank while in S_1, it writes a '1' (the symbol for even parity) and enters S_{halt}.

Table 7.2 State transition table for the parity machine on unary integers

State	*Read*	*Write*	*Move*	*Next state*
S_{start}	'1'	'•'	R	S_1
S_{start}	'•'	'0'	none	S_{halt}
S_1	'1'	'•'	R	S_{start}
S_1	'•'	'1'	none	S_{halt}

the information regarding the parity of the string – which is the only information that must be carried forward in time – only has two possibilities. This amount of information, one bit, does not grow at all as a function of the input size. The fact the machine is "reborn" each time it enters S_{start} or S_1 does not hinder its operation – the machine is compact even though the symbolic encoding is not. Once it finds the '•' (signifying that it has run out of '1's), the machine transitions to the halt state – using its implicit knowledge (the state it is in) to dictate whether it should leave a '1' or a '0'. This "knowledge" is non-symbolic. While states of the machine carry information forward in time, they do not do so in a form that is accessible to computation outside of this procedure. However, the procedure leaves behind on the tape a symbol for the parity of the number, and this is accessible to computation, because it is in memory (on the tape), where other procedures can read it.

This procedure, although implemented by a Turing machine, can be implemented on a weaker computational mechanism called a *finite state automaton*, which is a Turing machine with a multi-state processor but no read/write symbolic memory. It reads each datum in order and then produces an output. This machine for f_{is_even} never backtracks on the tape, and it never writes to it. That the problem can be solved without writing to symbolic memory means that it is solvable by such weaker

FEB-05-2015 12:19 P.010

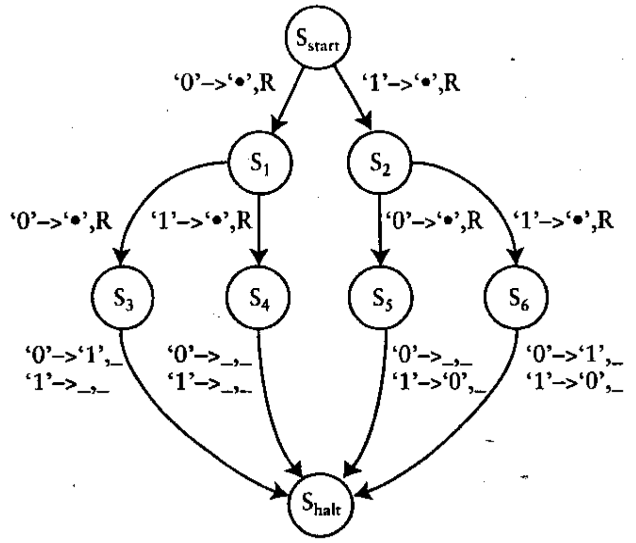

Figure 7.3 State diagram for parity-determining machine operating on nominally encoded integers. When the same state transition is caused by the reading of either bit, it is doubly annotated.

machines. Problems that may be solved without writing to symbolic memory are called *regular* problems. There are many such problems; however, there are many others that are routinely solved by animals and that *do* demand all the components of the Turing machine. They cannot be solved by a physically realizable finite-state machine, which cannot write to symbolic memory.

Next we will consider f_{is_even} as implemented on the nominal encoding from Chapter 6. Figure 7.3 shows the state diagram. It implements a look-up table, a non-compact procedure operating on a compact but nominal encoding. The procedure shown can handle an input of three bits; however, the number of states needed grows exponentially in the number of input bits supported. This Turing machine is once again emulating a weaker finite-state machine. Stepping through each datum, it uses its states to "remember" what it has seen. The structure of the Turing machine (as laid out by the transition table or by the state diagram) directly reflects the binary tree that it is implementing. Notationally, the transition arrows that go to the halt state have two labels each, indicating that these two inputs produce the same state change – but not necessarily the same actions. If comparing this tree to that in Figure 6.1, take note that here we are reading the bits from left to right.

Finally, we implement the compact procedure for f_{is_even} that operates on the compact binary encoding of the integers ('0' for 0, '1' for 1, '10' for 2, and so on). It maps them to '0' or '1' according to whether the final bit is '1' (hence, the integer is odd) or '0' (hence, the integer is even). The state diagram is shown in Figure 7.4. The procedure is easy to state in English: "Output the opposite (*not* or *inverse*) of the final bit." The Turing machine that implements this function reflects this simplicity. Its structure is similar to the procedure above that operates on the unary

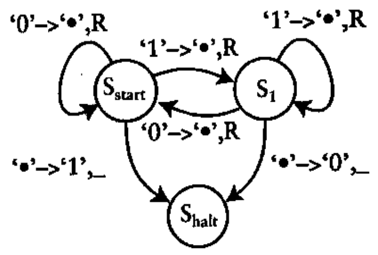

Figure 7.4 State diagram for a machine operating on binary encodings of the integers, erasing the data string as it goes. If it is in S_{start}, the last bit read was '0'; if in S_1, the last bit read was '1'. If it reads a '•' when in S_1, it writes a '0' (the symbol for odd parity) and enters S_{halt}. If it reads a '•' when in S_{start}, it writes a '1' (the symbol for even parity) and enters S_{halt}.

encoding. This machine also uses two states as memory. In this case, however, each state "knows" the *bit* it has last seen, not the *parity of the bit string* that it has seen so far. Because each bit is erased as the machine moves along, when it falls off the end of the string, it has lost its symbolic knowledge of what the last bit was. It must remember this information in non-symbolic (state memory) form.

Different procedures may be used to determine the same function, operating on the same encoding. Figure 7.5 shows the state diagram of a procedure that again uses the compact binary encoding of the integers. This machine leaves its tape (symbolic) memory intact. In S_{start}, it moves to the right along the sequence of '1's and '0's until it reads a blank ('•'), indicating that it has reached the end of the symbol string. Reading a '•' while in S_{start} causes it to move backward one step on the tape and enter S_1. In S_1 then, the read/write head is reading the last datum. It has gained access to this information in a state-independent manner. Both states S_{start} and S_1 read this last bit. The knowledge of this bit, therefore, is not tied to the state of the machine. Having carried the information forward in time in a computationally accessible form in symbolic memory, the machine (that is, the processor) is not dependent upon its own state to carry the information. Information from the past is informing the behavior of the present. By contrast, the knowledge that the machine is now reading the last bit is embodied in the processor's state; if it is in S_1, it's reading the last bit. In this state, it inverts the bit, and enters S_2. S_2 clears the tape, stepping backward when it reads a bit, through the input, erasing as it goes. Note that here we have enhanced our notation to allow for multiple symbols on the left side of the arrow. This is simply a shorthand for multiple read symbols that lead to the same actions (write, move, and state change). When it finally reads a '•', it enters S_3. In this state, it steps back to the right through the blanks left by the erasing done by the previous state, until it reads a bit. This bit is the answer to the parity question – which, again, it has remembered in symbolic form. Seeing the answer, it enters S_{halt}.

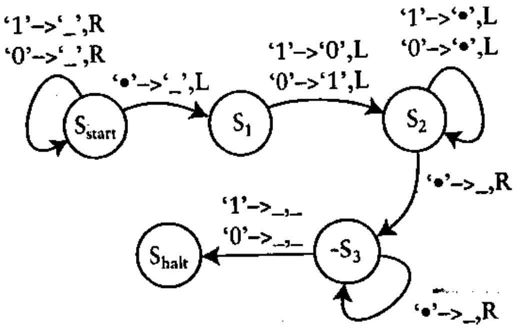

Figure 7.5 State diagram for parity-determining machine operating on binary encodings of the integers without erasing the data string.

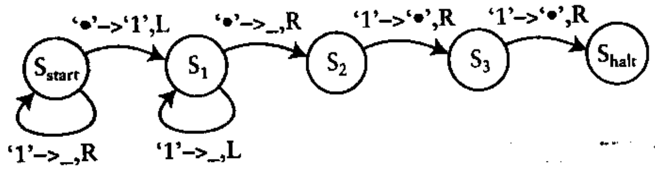

Figure 7.6 State diagram for an adding machine operating on a unary encoding of the integers ('1' for 0, '11' for 1, and so on).

Turing Machines for f_+

We now turn our attention to creating two procedures implemented on Turing machines for f_+. First, we solve the problem for a unary encoding of the integers. We again use the encoding in which the integer n is encoded by $n + 1$ '1's.

We stipulate that the symbols for the two numbers to be added will be separated by a single '•', the blank symbol. Thus, if we want to add 2 and 3, the initial state of the tape will be: . . . • • $\boxed{1}$ 1 1 • 1 1 1 1 • • . . . The first string of three '1's encodes for 2, then comes the '•' that serves to separate the two symbols, and then comes the second string of (four) '1's, which encodes for 3. As usual, the reading head is at the start of the first symbol when the computation begins. Our adding machine has five states (Figure 7.6). At the end of the computation, the tape should be: . . . • • $\boxed{1}$ 1 1 1 1 1 • • . . .

This machine starts by skipping over all of the '1's in the first addend while remaining in S_{start}, until it finds the '•'. It then converts this punctuation mark into '1', thereby forming one continuous data string, and then it transitions from S_{start} to S_1. At this point, we almost have the result we want. There are just two extra '1's. The first extra '1' came when we filled in the gap. The second came because of our encoding system. Each number n is encoded with $n + 1$ '1's and therefore the numbers x and y will have a total of $x + 1 + y + 1 = (x + y) + 2$ total '1's – the other extra '1'. The task remaining is to remove the two '1's. S_1 steps back through the '1's until it reads a '•', which causes it to step back once to the right and transition to S_2. S_2 reads and deletes the first '1' and transitions to S_3, which reads and deletes the second '1' and transitions to S_{halt}.

Changing the punctuation mark to a '1' may seem like a trick. The procedure takes advantage of the way the numbers are placed on the tape – the way they happened to be symbolized. This highlights the difference between reality and representation, and the importance of the encoding scheme that is chosen in representational systems. In a sense, all computation is a "trick." After all, we are representing (typically) real entities using sequences of symbols. Then, by performing manipulations on the symbols themselves, we end up with more symbols that themselves map back to entities in the real world. For these symbolic processes to yield results that reflect actual relationships in the real world may seem too much to hope for. Yet, such tricks have transformed the world. The tricks that allow the algebra to yield results that reflect accurately on geometry have been put to productive use for centuries. Such tricks allow us to determine how far away stars are. Such tricks have created the Internet. Perhaps what is most astonishing is that such tricks work for complex and indirect encoding schemes such as the binary encoding of integers. That integers can be encoded into unary (analog) form, manipulated by essentially "adding" them together, and then converted back to the integer may seem ho-hum. This coding is direct, and the procedure itself is direct. Yet, as we have stressed, this approach rapidly becomes untenable as the integers that one wants to deal with grow large. There isn't enough paper in the universe to determine the sum of $10^{56} + 10^{92}$. Yet using a compact-encoding (the decimal exponential system, or any other base for that matter) and then a compact procedure (addition as we learned as children) makes this (almost) child's play. It is no overstatement to say that the modern world would not be possible if this were not the case. Perhaps the success of such "tricks" is due to the likelihood that they are not tricks at all. Perhaps the symbolic representation of a messy reality reflects deep and simple truths about the reality that is encoded.

We come finally to consider the most complex procedure whose implementation on a Turing machine we detail – a compact procedure for f_+ operating on the binary encoding for integers. In Chapter 6 we described this procedure in what is often called *pseudo-code* – descriptions that are informal and intended for easy human consumption, but give one enough detail to go off and implement the procedure in actual computer code. Now, we use this procedure in its broadest strokes; however, we make some changes that aid us in implementing the procedure on a Turing machine.

The biggest change we make is that we augment our stock of three symbol elements ('•', '0', and '1') with two new elements, 'X' and 'Y'. We do this to minimize the number of different states in the machine. A necessary part of the addition procedure is keeping track of how far it has progressed. As always, this information can be carried forward in time in two different ways, either in symbolic memory (on the tape), or by means of state memory. If we were to do it by state memory, we would need to replicate a group of the states over and over again. Each replication would do the same thing, but to the next bits in the two strings of bits being processed. The purpose served by the replications would be keeping track of the position in the bit string to which the procedure has progressed. If we used state memory to do this, then the number of states would be proportional to the length of the strings that could be processed. By adding to our symbolic resources the two additional symbols that enable us to carry this information forward on the tape, we create a machine in which the number of states required is independent of the length of the strings to be processed.

There are other ways of achieving a procedure whose states do not scale with the length of the strings to be processed. For example, instead of enriching our set of symbol elements first with '•', and then with 'X' and 'Y', we could use only the minimal set ('0' and '1') and create a sub-procedure that functions to divide the tape into 3-cell words. The remainder of the procedure would then treat each word as an 8-symbol element ('000', '001', 010', etc.). Except for the multi-state sub-procedures that read and wrote those words, that procedure would look much like the one we here describe, because it would then be operating on 8 virtual symbol elements. Our approach serves to remind the reader that a Turing machine can have as many symbol elements as one likes (two is simply the absolute minimum), and, it keeps the state diagram (relatively) simple.

Unlike the other procedures described in Chapter 6, the procedure there described for f_+ may seem to be of a different kind. The symbols were placed in a two-dimensional arrangement and the pseudo-code procedure used phrases such as "add the two top numbers," and "at the top of the column to the left of the current column." Can we recode the symbols to be amendable to the one-dimensional world of a Turing machine? The Church-Turing hypothesis says that we can. And, indeed, it is not difficult: We put the four rows (Carry, Addend1, Addend2, and Sum) end to end, using the blank symbol as a punctuation mark to separate them. We handle the carries as they occur, essentially rippling them through the sum. That is to say, each power of two is added in its entirety as it is encountered. Rather than create the symbol string for the sum in a separate location on the tape, we transform the first addend into the sum and erase the second as we go.[3] The state diagram is in Figure 7.7.

In our example, we add the integer 6 to the integer 7 to compute the resulting integer, 13. The tape initially contains . . . • • $\boxed{1}$ 1 0 • 1 1 1 • • . . . , and it ends up containing . . . • • $\boxed{1}$ 1 0 1 • • . . .

Walking through the procedure at the conceptual level, we see the following:

. . . • • 1 1 0 •	Start with 6, the first addend (which will become the sum):
. . . • • 1 1 1 •	Add the 1 in the ones place to get 7.
. . . • 1 0 0 1 •	Add the 1 in the twos place to get 9 (the carry ripples through).
. . . • 1 1 0 1 •	Add the 1 in the fours place to get 13.

Conceptually, the machine keeps track of its progress by "marking" the bits of the first addend to keep track of which columns (powers of two) have been processed.

[3] Thanks to David Eck for these modifications and the associated Turing code.

FEB-05-2015 12:19 P.015

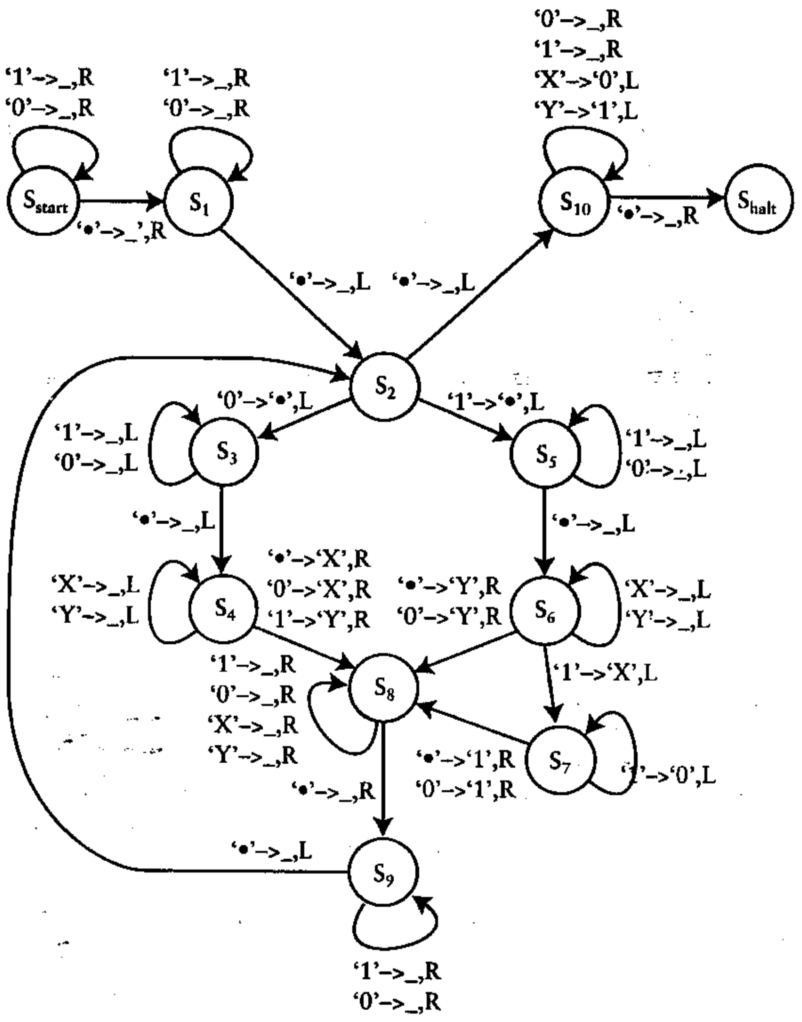

Figure 7.7 State diagram for the machine that does addition on binary encoded integers, using an augmented set of symbol elements to keep symbolic track of the progress of the procedure. S_{start} and S_1 move the head to the last bit of Addend2. The circle composed of S_2, S_3, S_4, S_5, S_6, and S_8 forms the look-up table for the sum of two binary digits (0 + 0, 0 + 1, 1 + 0, and 1 + 1). S_7 ripples the carry through, however successive '1's lie immediately to the left in the first addend until it gets to the first '0', where it deposits the carry bit. S_{10} converts the 'X's and 'Y's back to '0's and '1's.

The marking is accomplished by temporarily replacing processed '0's with 'X's and processed '1's with a 'Y's. When the machine has finished computing the sum, it goes back and restores the 'X's to '0's and the 'Y's to '1's.

Table 7.3 shows a trace of the tape as the Turing machine computes . . . • • $\boxed{1}$ 1 0 • 1 1 1 • • (6 + 7). S_3 and S_4 operate when the bit in the first addend is '0',

Table 7.3 Trace of the procedure when adding 6 (binary 110) and 7 (binary 111) to get 13 (binary 1001)

Tape	*State*	*Comment*
· · $\boxed{1}$ 1 0 · 1 1 1 · ·	Start	Move to the end of first Addend2
· · 1 $\boxed{1}$ 0 · 1 1 1 · ·	Start	
· · 1 1 $\boxed{0}$ · 1 1 1 · ·	Start	
· · 1 1 0 $\boxed{\cdot}$ 1 1 1 · ·	Start	
· · 1 1 0 · $\boxed{1}$ 1 1 · ·	1	
· · 1 1 0 · 1 $\boxed{1}$ 1 · ·	1	
· · 1 1 0 · 1 1 $\boxed{1}$ · ·	1	
· · 1 1 0 · 1 1 1 $\boxed{\cdot}$ ·	1	
· · 1 1 0 · 1 1 $\boxed{1}$ · ·	2	Working on next bit of Addend2 (ones place)
· · 1 1 0 · 1 $\boxed{1}$ · · ·	5	Found a 1, so of to state 5 for Addend1
· · 1 1 0 · $\boxed{1}$ 1 · · ·	5	
· · 1 1 0 $\boxed{\cdot}$ 1 1 · · ·	5	
· · 1 1 $\boxed{0}$ · 1 1 · · ·	6	Addend1 has a '0', sum = 1 + 0 = 1 = (y), no carry
· · 1 1 y $\boxed{\cdot}$ 1 1 · · ·	8	Back to find next bit of Addend2
· · 1 1 y · $\boxed{1}$ 1 · · ·	9	
· · 1 1 y · 1 $\boxed{1}$ · · ·	9	
· · 1 1 y · 1 1 $\boxed{\cdot}$ · ·	9	Found the end, step left for next bit
· · 1 1 y · 1 $\boxed{1}$ · · ·	2	Working on next bit of Addend2 (twos place)
· · 1 1 y · $\boxed{1}$ · · · ·	2	Found a '1', so off to state 5 for Addend1
· · 1 1 y $\boxed{\cdot}$ 1 · · · ·	5	
· · 1 1 $\boxed{y}$ · 1 · · · ·	6	Skip over y, already handled
· · 1 $\boxed{1}$ y · 1 · · · ·	6	Addend1 has a '1', sum = 1 + 1 = 0 = (x), with carry
· · $\boxed{1}$ x y · 1 · · · ·	7	Rippling carry, carry = 1 + 1 = 0 (x), with carry
· $\boxed{\cdot}$ 0 x y · 1 · · · ·	7	No more bits, placing the carry at end
· 1 $\boxed{0}$ x y · 1 · · · ·	8	Back to find next bit of Addend2
· 1 0 $\boxed{x}$ y · 1 · · · ·	8	Skip over x
· 1 0 x $\boxed{y}$ · 1 · · · ·	8	Skip over y
· 1 0 x y $\boxed{\cdot}$ 1 · · · ·	8	
· 1 0 x y · $\boxed{1}$ · · · ·	9	
· 1 0 x y · 1 $\boxed{\cdot}$ · · ·	9	Found end, left for next bit (fours place)
· 1 0 x y · $\boxed{1}$ · · · ·	2	Found a '1', so off to state 5 for Addend1
· 1 0 x y $\boxed{\cdot}$ · · · · · ·	5	
· 1 0 x $\boxed{y}$ · · · · · · ·	6	Skip over y
· 1 0 $\boxed{x}$ y · · · · · · ·	6	Skip over x
· 1 $\boxed{0}$ x y · · · · · · ·	6	Found a 0, sum = 1 + 0 = 1 (y), no carry
· 1 y $\boxed{x}$ y · · · · · · ·	8	Back to find next bit of Addend2
· 1 y x $\boxed{y}$ · · · · · · ·	8	Skip over y
· 1 y x y $\boxed{\cdot}$ · · · · · ·	8	
· 1 y x y · $\boxed{\cdot}$ · · · · ·	9	Found end, left for next bit (eights place)
· 1 y x y $\boxed{\cdot}$ · · · · · ·	2	There is no eights place, time to clean up
· 1 y x $\boxed{y}$ · · · · · · · ·	10	Change y back to 1
· 1 y $\boxed{x}$ 1 · · · · · · · ·	10	Change x back to 0
· 1 $\boxed{y}$ 0 1 · · · · · · · ·	10	Change y back to 1
· $\boxed{1}$ 1 0 1 · · · · · · · ·	10	Found first digit, all done

while S_5 and S_6 operate when it is '1'. They discern the corresponding bit in Addend2 and write the appropriate bit over the bit that is currently in that position in Addend1. This is written either as an 'X', for '0', or a 'Y' for '1', marking the current row (corresponding to the power of two being added) as done.

One may see the near duplication of S_3, S_4 and S_5, S_6 as wasteful. Can one do better? In this case, one cannot. The look-up table approach here is necessary as the two arguments are each just one bit. Therefore, there is no useful analytic decomposition of the arguments that would allow us to form a compact procedure. This imbedded look-up table, however, does not do our procedure any harm. Regardless of how big our addends grow, the look-up table still only needs to handle two bits.

Once this bit has been handled, the procedure moves back to the right to find the next bit to be added (S_8 and S_9). The machine then returns to S_2, being reborn with respect to its state memory. After all bits are processed, the machine transitions to S_{10} where it cleans up the tape (changes all of the 'X's back to '0's and 'Y's back to '1's) and halts.

It is hard to ignore the relative complexity of the Turing machine for f_+ that uses encoded compact symbols as compared to the one that uses the unary symbols. Both procedures are compact, and yet the procedure that operates on the compact symbols requires more states. We have traded some algorithmic simplicity for a procedure that can work on an arbitrarily large number of realistically symbolizable numbers – a trade that is mandatory for any machine that needs to work with many symbols.

The reader can verify that this machine will work for any initial pair of numbers. That is, if one puts in the binary numeral for the first number, a '•', and then the binary numeral for the second number, sets the machine to the initial state, and runs it – the machine will stop when it has converted what is written on the tape to the binary numeral for the sum of the two input numbers. Thus, this machine can add any two numbers we care to give it. It is not a look-up table; it is generative. It can give us the sums for pairs of numbers whose sums we do not know (have never computed).

Turing went on to show how to create more complicated machines that generated all computable numbers (real numbers for which there is a procedure by which one can determine any digit in its decimal expansion). He also showed how to implement all five of the elementary operations of arithmetic, from which all the other operations of arithmetic may be constructed. Also, how to implement basic text processing operations such as copying and concatenating. (These can all be shown to be equivalent to arithmetic operations.) His machines were never intended to be constructed. They were preposterously inefficient. They served a purely conceptual purpose; they rendered precise the notion of an effective procedure and linked it to the notion of what could be accomplished through the operation of a deterministic machine, a machine whose next action was determined by its current state and the symbol it was currently reading.

As one might imagine, there are variants on the Turing machine, but all the ones that have been suggested have been proved to be equivalent to the machine we have described, even stochastic (non-deterministic) variants. There are other approaches to specifying what is and is not computable, notably, the theory of recursive functions, but it has been proved that the Turing-computable functions are exactly the recursive functions.

Minimal Memory Structure

In a Turing machine, the head moves step by step along the tape to bring the symbol to be read to the head, which feeds the processor, the part of the machine whose state varies from step to step within a computational procedure. If a symbol is written to memory, it is always written to the cell that is currently under the head. A critical part of each processor state is the specification of how it moves the head. Only the symbol currently being read can contribute to this determination. Moving the head brings a different symbol into play. Thus, how the head moves determines which symbol in memory (on the tape) gains causal efficacy in the next step. The movements of the head are maximally simple: either one cell to the left or one cell to the right. One might suppose that a more complexly structured memory would be needed to achieve full computational power. It turns out, that it isn't. The sequential structure imposed by placing the data on a tape is all the structure that is needed.

This amount of structure is, however, critical. Memory in a Turing machine is not a disorderly basket into which symbols are tossed and which must then somehow be rummaged through whenever a particular symbol must again enter into some computational process. Memory is sequentially structured and that structure is a critical aspect of the procedures that Turing specified. As the example of the addition procedure illustrates, the arrangement of the symbols on the tape and the sequence in which the procedure brings them into the process by moving the reading head to them are the keys to the success or failure of the procedure.

Also critical is the question of how the symbols in memory and the machinery that operates on those symbols are brought together in space and time. In Turing's conceptual machine, the processor accessed the symbols by moving the head through memory. In modern, general-purpose computers, the symbols are brought to the processing machinery by a fetch or read operation and then exported back to memory by a put or write operation. It is widely assumed in the neural network literature that it is precisely in this particular that computation in nervous tissue departs most fundamentally from computation in modern computing machines. It is thought that in neural computation the data (whether they should be thought of as symbols or not is in dispute) and the machines that operate on the data are physically intertwined in such a way that there is no need to bring the data to the machinery that operates on it. However, the developments of this conception that we are familiar with generally avoid the question of how the combinatorial operations are to be realized – operations such as the arithmetic operations in which two different symbols must be brought together in space and time with machinery capable of generating from them a third symbol. The challenge posed by the necessity of implementing combinatorial operations is that of arranging for *whichever* two symbols need combining to come together in space and time with the machinery capable of combining them. It would seem that the only way of arranging this – other than bringing them both from memory to the combinatorial machinery – is to make a great many copies of the symbols that may have to enter into a combinatorial function and distribute these copies in pairs along with replications of the machinery capable of combining each such pair. This leads to a truly profligate use of physical resources. We will see in Chapter 14 that this is what at least some neural network models in fact suppose.

General Purpose Computer

Perhaps Turing's most important result was to prove the existence of (i.e., mathematical possibility of) universal Turing machines. A universal Turing machine is a machine that, when given on its tape an encoding for the transition table for any other Turing machine, followed by the state of that machine's tape at the start of its computations (the input), leaves the output segment of its own tape in the same state as the state in which the other machine would leave its tape. In other words, the universal Turing machine can simulate or emulate any other Turing machine operating on any input appropriate to that other machine. (Remember that other Turing machines are computation-specific.) Thus, a universal Turing machine can do any Turing-computable computation, which is to say, given the current state of our understanding, any computation that can in principle be done. This is, in essence, an existence proof for a general purpose computer. The computers that most of us have on our desks are, for most practical purposes, realizations of such a machine. But their architecture is somewhat different, because these machines, unlike Turing's machines, have been designed with practical considerations very much in mind. They have been designed to make efficient use of time and memory.

The functional architecture of practical universal Turing machines reflects, however, the essentials in the functional architecture of Turing's computation-specific machines. First and foremost, they all have a read/write memory, which, like Turing's tape, carries symbolized information forward in time, making it accessible to computational operations. Turing's formalization has allowed others to investigate the consequences of removing this essential component (Hopcroft, 2000; Lewis, 1981). As we have already noted, a machine that cannot write to the tape – that cannot store the results of its computations in memory for use in subsequent computations – is called a *finite state machine*. It is provably less powerful than a Turing machine. There are things that a Turing machine can compute that a finite state machine cannot because it has no memory in which to store intermediate results. We consider the limitations this imposes in Chapter 8.

Putting the transition table in memory

The modern computer differs from the Turing machines we have so far described in a way that Turing himself foresaw. In the architecture we have so far considered, the processor with its different states are one functional component, and the tape is another. The states of the processor carry information about how to do the computation. They are a collection of suitably interconnected mini-machines. The symbols on the tape carry forward in time the information extracted by earlier stages of the computation. Turing realized that it was possible to put both kinds of information on the tape: the how-to information in the transition table could be symbolized in the same way, and by the same physical mechanisms, as the data on which the procedure operated. This is what allowed him to construct his universal Turing machine. This insight was a key step on the road to constructing the modern general purpose computer. A machine with a stored-program architecture is often called a von Neumann machine, but the basic ideas were already in Turing's seminal paper, which von Neumann knew well.

In the stored-program architecture, the processor is given some basic number of distinct states. When it is in one of those states, it performs a basic computational operation. It has proved efficient to make machines with many more elementary hard-wired actions than the three that Turing allowed – on the order of 100. Each of these actions could in principle be implemented by a sequence of his three basic actions, but it is more efficient to build them into the different states of the processing machinery.

The possible actions are themselves represented by nominal binary symbols (bit patterns, strings of '1's and '0's), which are in essence names for the various states of the machine (the equivalent of S_1, S_2, etc. in our diagrams). This allows us to store the transition table – the sequence of instructions, that is, states – in memory (on the tape). In this architecture, computations proceed as follows: the processing machinery calls an instruction from the sequence in memory. This instruction configures the processor to carry out one of its elementary hard-wired operations, that is, it puts the processor in the specified state. After placing itself in one of its possible states by calling in an instruction name from memory, the processor then loads one or two data symbols from memory. These correspond to the symbol being read or scanned by the head in Turing's bare-bones architecture. The operations particular to that state are then performed. The processor may for example add the two symbols to make a symbol for the sum of the numbers that they represent, or compare them and decide on the basis of the comparison what the next instruction to be called in must be. Finally, the resulting symbol is written to memory and/or the machine branches to a different location in the sequence of instructions. The processor then calls in from the new location in program memory the name of the next instruction in the list of instructions or the instruction decided on when it compared two values. And so on.

Storing the program in the memory to which the machine can write makes it possible for the machine to modify its own program. This gives the machine two distinct ways in which it can learn from experience. In the first way, experience supplies the data required by pre-specified programs. This is the only form of learning open to a machine whose program is not stored in memory but rather hard-wired into the machine. Machines with this structure have a read-only program memory. In the second way, experience modifies the program itself. A point that is sometimes overlooked is that this second form of learning requires that one part of the program – or, if one likes, a distinct program – treat another part of the program as data. This second, higher-level program establishes the procedure by which (and conditions under which) experience modifies the other program. An instance of this kind of learning is the back-propagation algorithm widely used in neural network simulations. It is not always made clear in such simulations that the back-propagation algorithm does not run "on" the net; it is the hand of an omniscient god that reaches into the net to make it a better net.

Using the same memory mechanism to store both the data that must be processed and the transition table for processing them has an analog in the mechanism for the transmission and utilization of genetically coded information. In most presentations of the genetic code, what is stressed is that the sequence of triplets of base pairs (codons) in a gene specifies the sequence of amino acids in the protein whose structure is coded for by that gene. Less often emphasized is that there is another part of every gene, the promoter part, which is just as critical, but which does not code for the amino acid sequence of a protein. Promoters are sequences of base-pairs to which transcription factors bind. The transcription of a gene – whether its code is being read and used to make its protein or not – is governed by the binding of transcription factors to the promoters for that gene. Just as in a computer, the result of expressing many different genes depends on the sequence and conditions in which they are expressed, in other words, on the transition table or program. The sequence and conditions in which genes are expressed is determined by the system of promoters and transcription factors. The genetic program information (the transition table) is encoded by the same mechanism that encodes protein structure, namely base-pair sequences. So the genetic memory mechanism, like the memory mechanism in a modern computer, stores both the data and the program. DNA is the inherited-memory molecule in the molecular machinery of life. Its function is to carry heritable information forward in time. Unlike computer memory, however, this memory is read-only. There is, so far as we now know, no mechanism for writing to it the lessons of experience. We know from behavior, however, that the nervous system does have a memory to which it can write. The challenge for neurobiologists is to identify that mechanism.

Summary

We have reviewed and explicated key concepts underlying our current understanding of machines that compute – in the belief that the brain is one such machine. The Church-Turing thesis, which has withstood 70 years of empirical testing, is that a Turing machine can compute anything that can be computed by any physically realizable device. The essential functional components of a Turing machine are a read/write, sequentially structured symbolic memory and a symbol processor with several states. The processor's actions are determined by its current state and the symbol it is currently reading. Its actions have two components, one with respect to the symbolic memory (metaphorically, the tape) and one with respect to its own state. The memory-focused components are writing a symbol to the location currently being read and moving the head to one of the two memory locations that adjoin the currently read location. Moving the head brings new symbols stored in memory into the process. The other component is the change in the state of the processor. The machinery that determines the sequence of states (contingent on which symbols are encountered) is the program. The program may itself be stored in memory – as a sequence of symbols representing the possible states.

The Turing machine is a mathematical abstraction rooted in a physical conception. Its importance is twofold. First, it bridges the conceptual gulf between our intuitive conceptions of the physical world and our conception of computation. Intuitively, computation is a quintessentially mental operation, in the Cartesian dualist sense of something that is intrinsically not physical. Our ability to compute is the sort of thing that led to Descartes' famous assertion, "I think therefore I am." The "I" referred to here is the (supposed) non-physical soul, the seat of thought. In the modern materialist (non-dualist) metaphysics, which is taken more or less for granted by most cognitive scientists and neuroscientists, the material brain is the seat of thought, and its operations are computational in nature. Thus, it is essential to develop a firm physical understanding of computation, how it works physically speaking, how one builds machines that compute. (In the next chapter, we get more physical.)

Second, we believe that the concepts underlying the design of computing machines arise out of a kind of conceptual necessity. We believe that if one analyzes any computing machine that is powerful, fast, and efficient, one will find these concepts realized in its functional structure. That has been our motivation for calling attention to the way in which these concepts are implemented, not only in modern computers, but also in the best understood biological machinery that clearly involves a symbolic memory, namely, the genetic machinery. This well-understood molecular machinery carries heritable information from generation to generation and directs the construction of the living things that make up each successive generation of a species. In the years immediately following the discovery of the structure of the DNA molecule, biologists discovered that the genetic code was truly symbolic: there was no chemical necessity connecting the structure of a gene to the structure of the protein that it coded for. The divorcing of the code from what it codes for is the product of a complex multi-stage molecular mechanism for reading the code (transcribing it) and translating it into a protein structure. These discoveries made a conceptual revolution at the foundations of biochemistry, giving rise to a new discipline, molecular biology (Jacob, 1993; Judson, 1980). The new discipline had coding and information processing as its conceptual foundations. It studied their chemical implementation. The biochemistry of the previous era had no notion of coding, let alone reading a code, copying it, translating it, correcting errors, and so on – notions that are among the core concepts in molecular biology.

Thus, if one believes that the brain is an organ of computation – and we take that to be the core belief of cognitive scientists – then to understand the brain one must understand computation and how it may be physically implemented. To understand computation is to understand the codes by which information is represented in physical symbols and the operations performed on those symbols, the operations that give those symbols causal efficacy.