6
Procedures

We now have a clearer understanding of the properties of physical symbols. In the represented system, $\hat{D}$, the entities can be anything, real or imagined, but in the representing system, $\hat{G}$, the symbols must be physically instantiated and distinguishable forms. We now need to understand the properties of the functions in $\hat{P}$ that play a role in physically realized representational systems.

As our understanding of symbols in $\hat{S}$ changed when we considered desirable physical properties, so must our understanding of the functions in $\hat{P}$. In this chapter we explore the properties of computable and physically realizable functions in a representational system. Computing functions is what allows functions to be put to productive use, just as distinguishing symbols and establishing referents for them is what allows symbols to be put to productive use.

Algorithms

As discussed in Chapter 3, function definitions may establish that a mapping exists from members of the domain to members of the codomain without necessarily giving a method or process to determine the output for a given input.

A clear and unambiguous method or process that allows one to determine robotically the input/output mappings of a function is called an *algorithm*. For example, the long multiplication method that grade school children are taught is an algorithm to determine the symbol for the product of two numbers. We saw some geometric algorithms in Chapter 4, such as the algorithm for determining a line segment that is $1/n$ of another line segment, where n is any integer (Figure 4.1).

To take a common example, let's say you have an endless supply of coins, denominated 25 cents, 10 cents, 5 cents, and 1 cent. Let's define a "change-giving" function that maps an amount of money 99 cents or less to the minimal group of coins that equal the given amount. Forty-three cents would map to one of the 25-cent coins, two of the 10-cent coins, one of the 5-cent coins and three of the 1-cent coins. Forty-three 1-cent coins would give the right amount of change, but not the minimal number of coins. This function is what cashiers must routinely determine (if they don't want to burden you with piles of pennies) when giving back change

for a purchase. These days, the function is generally implemented by a computer in the cash register, which then dispenses the change from a coin holder. While well defined, the definition for this function does not give us a method for determining it. Here is an algorithm that determines this function for an amount owed:

1 Take the largest coin of n cents where $n \leq$ the amount owed.
2 Reduce the amount owed by n cents.
3 If the amount owed is 0 cents, return all coins taken and stop.
4 Go back to State (line) 1.

The basic operating principle is to start in State 1 (line 1), and do each state in order (unless told otherwise). *State* is a general term that refers to a discernible stage that a process (procedure) is in during which it will act in some specified way. Each numbered line above describes a state. The reader is invited to try a few examples of applying this algorithm. In general, we suggest going through the process of trying any algorithms shown to convince yourself that they determine the function they are supposed to. This will also give you a feel for the mechanical and mindless nature of the processes – what ultimately allows them to be instantiated by bio-molecular and bio-physical processes.

In this change-giving algorithm we see a number of themes that we will see again in various incarnations: There are a series of distinct states that the algorithm proceeds through. The state determines what we do at a certain moment in time. There is the composition of functions, because later steps depend on the results of earlier steps. There are a number of functions that are embedded within the larger algorithm such as subtraction (reduction) and the $\leq$ relation. The algorithm returns to a previously visited state (go back to State 1). The algorithm eventually stops and returns an answer. The algorithm makes "decisions" (if) based on the current situation.

The fact that this algorithm depends on other functions for which there is no algorithm presented should give us pause. How do we determine "if the amount owed is 0 cents" or "the largest coin of n cents where $n \leq$ amount owed?" Ultimately, if we are going to flesh out algorithms in the detail that will be needed to understand how the brain might determine such functions, we will have to flesh out all of the pieces of such algorithms with processes that leave no room for interpretation, and demand no need for understanding the English language. In the next chapter, we will show a formalism, the Turing machine, which can be used to express algorithms in a form that leaves nothing open to interpretation.

Any particular algorithm determines one function; however any particular function can be determined by many possible algorithms. Another algorithm that determines the change-giving function is a look-up table. It has 99 entries. Table 6.1 shows a few of them. To determine the function, one simply looks up the answer (output) in the table. If using this algorithm, the cashier would have a little card on which was written the coins that should be returned for each possible change amount. The cashier then simply finds the required change on the card and returns the coins associated with it.

Table 6.1 Partial look-up table for change making

Change owed (in cents)	*Minimal group of coins (in cents)*
3	(1, 1, 1)
26	(25, 1)
37	(25, 10, 1, 1)
66	(25, 25, 10, 5, 1)
80	(25, 25, 25, 5)

Procedures, Computation, and Symbols

The algorithms we are interested in will describe the process by which the output of a function in $\hat{P}$ is determined given the arguments to the function. All of the functions in $\hat{P}$ must have algorithms for them, as one cannot make productive use of a function unless one can determine the output for any permissible input. Since every function in $\hat{P}$ maps physical symbols to physical symbols, the algorithms will be applied to physical symbols as input and produce physical symbols for their output. Ultimately, the algorithms will be instantiated as a physical system/process that acts on the symbols and produces other symbols. We will call symbol processing algorithms *effective procedures*, or just *procedures* for short. We can think of the procedures in $\hat{P}$ as being a set of physically realized functions, and continue to use the functional terminology.

We call the process of putting into action a procedure a *computation*. We also say that the procedure *computes* the output from the given inputs and computes the function. Just as it was the case for *defining* functions, there is no universally sanctioned way to describe a procedure that can compute a function. That said, a small set of procedural primitives used compositionally appears to suffice to determine any function. We saw a glimpse of this possibility in the themes from our change-giving example.

As we saw in the last chapter, if we are to create numerous symbols in $\hat{G}$ that can be put to productive use by procedures, we will have to build them out of component pieces (data). Additionally, we saw that using concatenation with a combinatorial syntax, we could get much more bang for our buck in terms of how many symbols could be constructed per physical unit. This reasoning applies to both nominal and compact symbols. Using combinatorial syntax, one can produce d^n symbols from d atomic symbols and strings of length n.

The input symbols must be distinguished by detecting their syntax, that is, their form and their spatial or temporal context. Their referents cannot come into play, because the machinery that implements a function (procedure) only encounters the symbols themselves, not their referents. This gives symbols an important property that is not shared by the entities they represent: symbols share a common atomic structure (a common representational currency), which is used to compute functions

of those symbols. This makes the description of procedures in $\hat{P}$ potentially much more uniform and capable of being formalized than are the functions for the entities that the procedures represent. Functions and entities in the represented system take a bewildering variety of physical and non-physical forms, but the machinery in the representing system that implements the corresponding functions on the symbols for those entities is most often constructed from a modest set of basic components.

An elementary neurobiological manifestation of this important principle is seen in the fact that spike trains are the universal currency by which information is transmitted from place to place within neural tissue. A spike is a spike, whether it transmits visual information or auditory information or tactile information, etc. This common signal currency enables visual signals carrying information about location to be combined with auditory signals carrying information about location. This combination determines the activity of neurons in the deep layers of the superior colliculus. We similarly imagine that there must be a common currency for carrying information forward in time, regardless of the character of that information. We return to this question in Chapter 16.

Similarly, the fact that the system of *real numbers* can be used to represent both discrete quantities, such as the number of earthquakes in Los Angeles in a given amount of time, and continuous quantities, such as the given amount of time, makes it possible to obtain a symbol that represents the rate of earthquake occurrence. The symbol is the real number obtained by dividing the integer representing the number of earthquakes by the real number representing the amount of time. If the integers were not part of the system of real numbers (the system of symbols on which the arithmetic functions are defined), this would not be possible (Leslie, Gelman, & Gallistel, 2008). Indeed, the emergence of the algebraic representation of geometry created a common-currency crisis at the foundations of mathematics, because there were simple geometric proportions, such as the proportion between the side of a square and its diagonal or the proportion between the circumference of a circle and its diameter, that could not be represented by the so-called *rational* numbers. As their name suggests, these are the numbers that suggest themselves to untutored reason. The algebraic representation of geometry, however, requires the so-called *irrational* numbers. The Greeks already knew this. It was a major reason for their drawing a strong boundary between geometry and arithmetic, a boundary that was breached when Descartes and Fermat showed how to represent geometry algebraically. In pursuit of what Descartes and Fermat started, mathematicians in the nineteenth century sought to rigorously define irrational numbers and prove that they behaved like rational numbers. They could then be added to the representational currency of arithmetic (the entities on which arithmetic functions operate), creating the so-called *real* numbers.

It is not always easy to find procedures for a given function. There exist well-defined functions for which no one has devised a procedure that computes them. Given a well-defined notion of what it means to be able to compute a function (we will discuss this is the next chapter), there are functions where it has been proven that no such procedures exist. Functions that cannot be computed under this notion are referred to as *uncomputable*. Other functions have procedures that in theory

compute them, but the physical resources required render the procedures impractical. Other procedures are rendered impractical by the amount of time that it would take the procedure to compute the function. Functions for which all possible procedures that compute them have such spatial or temporal resource problems are called *intractable* functions.[1] There is a large set of computationally important functions for which the only known procedures to solve them are not practical, and yet it is not known whether these functions are intractable.[2]

The lack of efficient procedures for finding the prime factors of large numbers is the basis for many of the encryption systems that protect your information as it moves back and forth across the internet. These schemes take advantage of the fact that while it is trivial to compute the product of any two prime numbers, no matter how large they may be, the known procedures for computing the inverse function, which specifies for every non-prime number its prime factors, become unusable as the inputs become large. Given a large number that is not itself prime, we know that there exists a unique set of prime factors and we know procedures that are guaranteed to find that set if allowed to run long enough, but for very large numbers "long enough" is greater than the age of the universe, even when implemented on a super computer. Functions whose inverses cannot be efficiently computed are called trap-door functions, because they allow one to go one way but not the other.

In short, there is a disconnect between the definition of a function and the ability to determine the output when given the input. The disconnect runs both ways. We may have a system that gives us input/output pairs and yet we do not have a definition/understanding of the function. Science is often in this position. Scientists perform experiments on natural systems that can be considered inputs and the natural system reacts with what may be called the outputs. However, the scientists may not have an independent means of determining and thereby predicting the input–output relationship. Scientists say in these cases that they don't have a model of the system. We would say that we don't have a representational system for the natural system.

Coding and Procedures

As we mentioned previously, the use of concatenated symbols that share a set of data elements (e.g., '0' and '1') as their constructive base suggests that procedures in $\hat{P}$ may be able to take advantage of this common symbol-building currency. This

[1] Typically, the line between tractable and intractable functions is drawn when the procedures needed to implement them need an amount of spatial or temporal resources that grows exponentially in the number of bits needed to compactly encode the input.

[2] This is the class of (often quite useful) functions that are grouped together under the heading NP-Complete. If any one of these functions turns out to have a feasible procedure, then all of them do. The tantalizing nature of this makes the question of whether there is a feasible such procedure one of the most famous open questions in computer science, called the P = NP problem. The general consensus is that these functions are intractable, yet no one has been able to prove this.

does not imply, however, that the particular encoding used for the symbols is irrelevant with respect to the procedures. In fact, there is a very tight bond between the procedures in $\hat{P}$ and the encoding of the symbols in $\hat{S}$.

If the code used is nominal, the nominal symbols cannot be put to productive use by taking advantage of aspects of their form. They can only be used productively in a mapping by distinguishing the entire data string (string of atomic elements) that constitutes the symbol and then using a look-up table to return the result. This is because the string used to form a nominal symbol is arbitrary – any string can be substituted for any other without loss or gain of referential efficacy. When a large number of nominal symbols is used, it becomes impractical to distinguish them. Encoding symbols, however, can be distinguished efficiently using what we will call *compact procedures*.

To see the force of this consideration, we examine potential representational systems involving the integers, $N = \{1, 2, \ldots, 10^{30}\}$, using nominal and encoding systems, and two arithmetic functions, the parity function f_{is_even}: $N \rightarrow \{false, true\}$, and the addition function f_+: $N \times N \rightarrow N$. Each datum (element in a symbol string) will come from the set {'0', '1'}, and we call each datum a bit. The symbols for *true* and *false* will be '1' and '0', respectively. Our procedures then will be of the form f_{is_even}: $D^{\circledast} \rightarrow \{0, 1\}$, where $D^{\circledast}$, the input, is a string of bits and the output is a single bit, and f_+: $D^{\circledast} \times D^{\circledast} \rightarrow D^{\circledast}$, where the inputs are two strings of bits and the output is a string of bits.

Procedures using non-compact symbols

Preparatory to considering procedures for f_{is_even} and f_+ that use compact symbols, we consider briefly the possibility of using non-compact encodings. One simple way to represent integers is the hash-mark or unary code in which the number of '1' bits is equal in numerosity to the number it encodes. We can reject this out of hand, because we must be able to operate on a number as large as 10^{30} and there are only on the order of 10^{25} atoms in the human brain. Even if we devoted every atom in the brain to constructing a unary symbol for 10^{30}, we would not have the physical resources. Unary symbols are not compact.

An analog symbol would have the same problem. Analog symbols share a property with non-combinatorial digital system in that they both produce an increase in the physical mass of the symbols that is linearly proportional to their representational capacity. Like the unary symbols, the most efficient use of physical resources for analog symbols would be one whereby each symbol is distinguished by the number of atoms that compose it. We could then use weight to distinguish the symbol for one integer from the symbol for another integer (assuming that all the atoms were atoms of the same substance). The problem with using analog symbols to represent our integers is twofold. First, such symbols are not compact, so there are not enough atoms in the brain to represent what may need to be represented. Second, no weighing procedure could distinguish the weight of 10^{29} atoms from the weight of $10^{29} + 1$ atoms. In short, non-compact symbols (digital or analog) are not practical as the basis for a representational system whose symbols must each be capable of representing a large number of different possible states of

Table 6.2 The parity function f_{is_even} with a nominal binary code for the integers 0–7

Integer (base 10)	*Nominal binary code*	*Parity*
0	001	1
1	100	0
2	110	1
3	010	0
4	011	1
5	111	0
6	000	1
7	101	0

the represented system; their demand on resources for symbol realization is too high, and the symbols soon become indistinguishable one from the next.

With a combinatorial syntax for (compact) symbol formation, the form of the symbol is varied by varying the sequence of atomic data (for example, '0's and '1's), with each different sequence representing a different integer. If we form our symbols for the integers in this way, we need a symbol string consisting of only 100 bits to represent any number in our very large set of integers (because $2^{100} > 10^{30}$). The demands on the physical resources required to compose the symbol for a single number go from the preposterous to the trivial.

Procedures for f_{is_even} using compact nominal symbols

Before discussing procedures for f_{is_even}, we must consider how we will encode our large (10^{30}) set of integers, because procedures are code-specific. The decision to use compact symbols does not speak to the question of how we encode number into these symbols – how we map from the different numbers to the different bit patterns that refer to them. We consider first the procedural options when we use a nominal mapping for the integers, one in which there are no encoding principles governing which bit patterns represent which numbers. Table 6.2 shows f_{is_even} defined for one possible nominal mapping for the integers 0–7. The bit patterns shown in the second column (the input symbols), can be shuffled at will. The only constraint is that the mapping be one-to-one: each integer must map to only one bit pattern and each bit pattern must represent only one integer. The question is, what does a procedure f_{is_even} look like if we use this kind of nominal coding of the integers? Part of what gives this question force is that the codings assumed by neurobiologists are commonly nominal codings: the firing of "this" neuron represents "that" state of the world.

Whatever procedure we use for determining f_{is_even}, it will have to distinguish every one of the 100 data elements that comprise the input symbol. In addition, since there is no principled way to determine the correct output from the input, we have no choice but to use a look-up table. That is, we must directly implement Table 6.2. We first consider a procedure that distinguishes the elements sequentially, moving

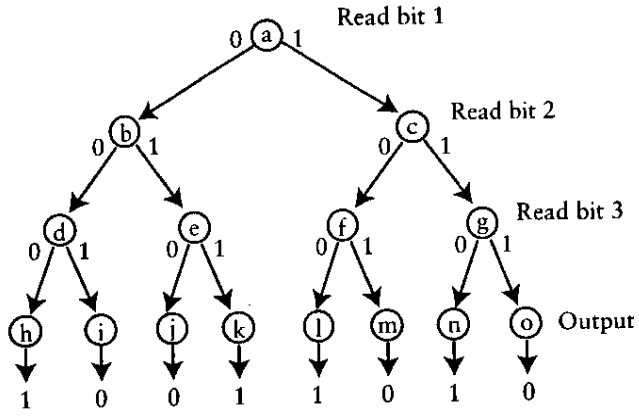

Figure 6.1 Binary search tree for determining the parity of the first eight integers using the nominal binary encoding in Table 6.2.

through the input string from right to left. As we move through the input, we will change from state to state within the procedure. Each state then will provide a memory of what we have seen so far. When we reach the last symbol, we will know what the output should be based on the state we have landed in. In effect, we will be moving through a tree of states (a binary search tree) that branches after each symbol encountered. Figure 6.1 shows the tree that corresponds to the given nominal mapping. Different input strings will direct us along different paths of the tree, in the end generating the pre-specified output for the given input (the symbol '1' or the symbol '0', depending on whether the integer coded for is even or odd).

Here is the procedure that implements the search tree shown in Figure 6.1:

1 Read bit 1. If it is a '0', go to state 2. If it is a '1', go to state 3.
2 Read bit 2. If it is a '0', go to state 4. If it is a '1', go to state 5.
3 Read bit 2. If it is a '0', go to state 6. If it is a '1', go to state 7.
4 Read bit 3. If it is a '0', output '1'. If it is a '1', output '0'. Halt.
5 Read bit 3. If it is a '0', output '0'. If it is a '1', output '1'. Halt.
6 Read bit 3. If it is a '0', output '1'. If it is a '1', output '0'. Halt.
7 Read bit 3. If it is a '0', output '1'. If it is a '1', output '0'. Halt.

We see a number of the same themes here that we saw in the change-making algorithm. There are two related properties and problems of look-up table procedures that are of great importance.

Combinatorial explosion. The search tree procedure has the unfortunate property that the size of the tree grows exponentially with the length of the binary strings to be processed, hence linearly with the number of different possible inputs. As we see graphically in Figure 6.1, each node in the tree is represented by a different state that must be physically distinct from each other state. Therefore, the number of states required grows exponentially with the length of the symbols to be processed. We used combinatorial syntax to avoid the problem of linear growth in

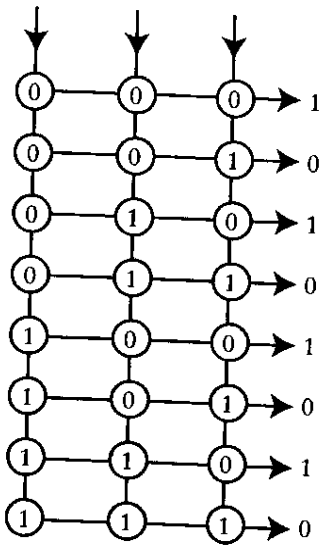

Figure 6.2 A content-addressable memory makes possible parallel search. The string to be processed is fed in at the top. All three bits go simultaneously to the three input nodes at each memory location. Each input node is activated only if the input bit it sees matches it. If all of the nodes at a given location are activated, they generate the output bit stored at that location, the bit that specifies the parity of the number represented by that location in the content-addressable memory. (The ordering of the binary numbers in this illustration plays no role in the procedure.)

required symbol size. However, because we have used a nominal encoding of the numbers, the problem has come back to haunt us in our attempts to create a procedure that operates productively on the symbols.

Pre-specification. Intimately related to, but distinct from the problem of combinatorial explosion is the problem of *pre-specification* in procedures based on a look-up table approach. What this means is that in the procedure, for every possible input (which corresponds to the number of potential symbols used for the input) it was necessary to embed in the structure of a state the corresponding output symbol that will be returned. This implies that all of the possible input/output pairs are determined *a priori* in the representing system. While one can certainly figure out whether any hundred-bit number is even, it is not reasonable to require the creator of the procedure to determine this in advance for all 2^{100} numbers. The problem with procedures based on look-up tables is that they are not productive; they only give back what has been already put in.

One may wonder whether the problem of processing a large number of nominal symbols goes away if one uses a parallel search procedure rather than a sequential procedure. We raise this question in part because, first, it is often argued that the brain is so powerful because it engages in massive parallel processing, and, second, many neural network models for implementing function are based on learned content-addressable memory nets. Content-addressable memories are look-up tables in which the different possible results are accessed through parallel search (see Figure 6.2).

As is apparent in Figure 6.2, a parallel search procedure, using a content-addressable memory table in lieu of a binary search tree, does not avoid either the combinatorial explosion problem or the problem of pre-specification. If anything, it makes the situation worse. The content-addressable memory requires a physically distinct memory location for every possible input string (leading to a combinatorial explosion). The hardware necessary to implement a comparison between an input bit and a bit stored at every location must also be replicated as many times as the length of the input. In addition, we have to store at that location the output bit specifying the parity of the integer to which that location is "tuned" (the pre-specification problem).

State memory

The sequential procedure illustrates a concept of central importance in our understanding of the possible architectures of computing machines, the concept of state memory. A state of a computing device is a hard-wired capacity to execute a particular function given a certain context. Each state essentially implements a mini look-up table of its own, and by moving from state to state, a machine can map a pre-specified set of inputs to a pre-specified set of outputs (implementing a larger look-up table). At any one time it is in one and only one of its possible states. Which state it is in at any given time depends on the input history, because different inputs lead to different state transitions. Take, for example, a typewriter (or, these days, a keyboard, which is not as favorable for our purposes, because the physical basis for its state changes are not apparent). When the Shift key has not been depressed (previous input), it is in one state. In that state it maps presses on the keys to their lower-case symbols. When the Shift key has been depressed, it maps the same inputs to their upper-case outputs, because the depressing of the Shift key has shifted the machine into a different state. In an old-fashioned typewriter, depressing the Shift key raised the entire set of striking levers so that the bottom portion of the strike end of each lever struck the paper, rather than the top portion. The fact that the machine was in a different state when the Shift key was depressed was, therefore, physically transparent.

In the tree search procedure for determining parity, the procedure travels down one branch of the tree, and then another and another. The branch it will travel down next depends on the state it has reached, that is, on where it is in the tree. The tree must be physically realized for this procedure to work because it is the tree that keeps track of where the procedure has got to. Each time the procedure advances to a new node, the state of the processing machinery changes.

The state that the procedural component of a computing machine is in reflects that history of the inputs and determines what its response to any given input will be. It is therefore tempting to use the state of the processing machinery as the memory of the machine. Indeed, an idea that has dominated learning theory for a century – and hence the attempt to understand learning and memory neurobiologically – is that all learning is in essence procedural and state based. Experience puts neural machines in enduringly different states (rewires them to implement a new

look-up table), which is why they respond differently to inputs after they have had state-changing (rewiring) experiences. Most contemporary connectionist modeling is devoted to the development of this idea. A recurring problem with this approach is, like the look-up tables they result in, these nets lack productivity: in this case one only gets back what *experience* has put in. (For an extensive illustration of what this problem leads to, see Chapter 14.)

While the use of look-up tables is in many cases either preposterously inefficient or (as in the present case) physically impossible, look-up tables are nonetheless an important component of many computing procedures. As the content-addressable example (Figure 6.2) suggests, a look-up-table procedure can involve accessing memory only once, whereas a procedure implemented by composing many elementary functions, with each composition requiring reading from and writing to memory, may take much longer.

Take, for example, the use of sine and cosine tables in a video game, where speed of the procedures is a top priority. Computing the sine and cosine of an angle is time intensive. For example, one formula for $\sin(x)$ where x is in radians is given by $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$. While one can limit the number of terms used, it may still take a fair amount of time to compute and the denominators become very large integers very quickly.

What is often done to overcome this speed problem is to make a look-up table for enough angles to give a reasonably good resolution for the game. For example, one might have a look-up table procedure that determines the sine function for each integral degree from 1 to 360. Then one only need find the closest integral angle to the input that is stored in the look-up table and use this for an approximation. One can take the two closest integral angles and then use linear interpolation to get a better and still easily computed estimate.

Procedures for f_{is_even} using compact encoding symbols

When symbols encode their referents in some systematic way, it is often possible to implement functions on those symbols using what we call a *compact procedure*. A compact procedure is one in which the number of bits required to communicate the procedure that implements a function (the bits required to encode the algorithm itself) is many orders of magnitude less than the number of bits required to communicate the look-up table for the function. And usually, the number of bits required to communicate a compact procedure is independent of the size of the usable domain and codomain, whereas in a look-up table procedure, the number of bits required grows exponentially with the size of the usable codomain and domain. We have begun our analysis of procedures with the parity function because it is a striking and simple illustration of the generally quite radical difference between the information needed to specify a compact procedure and the information needed to specify the look-up table procedure. As most readers will have long since appreciated, the parity-determining procedure for the conventional binary encoding of the integers is absurdly simple:

Read bit 1. If it is a '0', output '1'. If it is a '1', output '0'. Halt.

This function of flipping a bit (turning 0 to 1 and 1 to 0) is the function f_{not} in Boolean algebra ($f_{not}(0) = 1$, $f_{not}(1) = 0$). It is one of the primitive operations built into computers.[3] When communicating with a device that has a few basic functions built in, the procedure could be communicated as the composition $f_{output}(f_{not}(f_{first\_bit}(x)))$. This expression only has four symbols, representing the three needed functions and the input. Because these would all be high-frequency referents (in the device's representation of itself), the symbols for them would all themselves be composed of only a few bits, in a rationally constructed device. Thus, the number of bits required to communicate the procedure for computing the parity function to a (suitably equipped) device is considerably smaller than 10^2. By contrast, to communicate the parity look-up table procedure for the integers from 0 to 10^{30}, we would need to use at least 100 bits for *each* number. Thus, we would need about $(10^2)(10^{30}) = 10^{32}$ bits to communicate the corresponding look-up table.

That is more than 30 orders of magnitude greater than the number of bits needed to communicate the compact procedure. And that huge number – 30 orders of magnitude – is predicated on the completely arbitrary assumption that we limit the integers in our parity table to those less than 10^{30}. The compact procedure does not stop working when the input symbols represent integers greater than 10^{30}. Because we are using a compact encoding of the integers, the size of the symbols (the lengths of the data strings) present no problem, and it is no harder to look at the first bit of a string 1,000,000 bits long than it is to look at the first bit of a string 2 bits long. In short, there is no comparison between the number of bits required to communicate the compact procedure and the number of bits required to communicate the look-up table procedure that the compact procedure can in principle compute. The latter number can become arbitrarily large (like, say $10^{10,000,000}$) without putting any strain on the physical implementation of the compact procedure. Of course, a device equipped with this procedure never computes any substantial portion of the table. It doesn't have to. With the procedure, it can find the output for any input, which is every bit as good (indeed, much better, if you take the pun) than incorporating some realized portion of the table into its structure.

The effectiveness of a compact procedure depends on the symbol type on which it operates. When we use nominal symbols to represent integers, then there is no compact procedure that implements the parity function, or any other useful function. Nominal symbolization does not rest on an analytic decomposition of the referents. An encoding symbolization does. When the form of a symbol derives from an analytic decomposition of the encoded entity, then the decomposition is explicitly represented by the substructure of the symbol itself. The binary encoding of the integers rests on the decomposition of an integer into a sum of successively higher powers of 2 (for example, $13 = 1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) = 8 + 4 + 0 + 1$). In this decomposition, the parity of an integer is explicitly represented by the rightmost

[3] We conjecture that it is also a computational primitive in neural tissue, a conjecture echoed by the reflections expressed in the T-shirt slogan, "What part of *no* don't you understand?"

bit in the symbol for it. That makes possible the highly compact procedure for realizing the parity function.

Because this point is of quite fundamental importance, we illustrate it next with a biological example: If the nucleotide-sequence symbols for proteins in the DNA of a chromosome were nominal symbols, then we would not be able to deduce from those sequences the linear structures of the proteins they represent. In fact, however, the genetic code uses encoded symbols. The encoding of protein structures by nucleotide sequences rests on an analytic decomposition of the protein into a linear sequence of amino acids. Within the nucleotide sequences of the double-helical DNA molecule of a chromosome, different nucleotide triplets (codons) represent different amino acids, and the sequence of codons represents the sequence of the amino acids within the protein. The decomposition of proteins into their amino acid sequences and the explicit representation of this sequence within the genetic symbol (gene) for a protein makes possible a relatively simple molecular procedure for assembling the protein, using a transcription of the symbol for it as an input to the procedure. A small part of the genetic blueprint specifies the structure of a machine (the ribosome) that can construct an arbitrarily large number of different proteins from encoded symbols composed of only four atomic data (the A, G, T, and C nucleotides). The combinatorial syntax of these symbols – the fact that, like bits, they can be made into strings with infinitely various sequences – makes them capable of representing an arbitrarily large number of different proteins. Whatever the codon sequence, the transcription procedure can map it to an actual protein, just as the parity function and the about to be described addition function can map the binary encoding of any integer or any pair of integers to the symbol for the parity or the symbol for the sum. Biological coding mechanisms and the procedures based on them, like computer coding mechanisms and the procedures based on them, are informed by the logic of coding and compact procedures. This logic is as deep a constraint on the realization of effective molecular processes as is any chemical constraint.

Procedures for f_+

When working with symbols that refer to a quantity, the addition function is exceedingly useful because under many circumstances quantities combine additively (or linearly, as additive combination is often called). As always, the procedure depends on the encoding. If we use a unary (analog) code – symbols that have as many '1's as the integer they encode – the adding procedure is highly compact: you get the symbol for the sum simply by concatenating (stringing together) the addends (the symbols for the integers to be added). Or, if one thinks of the two numbers as being contained in two "bags," then one would get the sum by simply pouring the two bags into a new bag. The problem with this approach is not in the procedure – it could not be more compact – but rather in the symbolization (the encoding). As we have already seen, it leads to exponential growth in use of resources, so it is physically impractical. If we attempt to use a nominal coding, the symbols will be compact but we will be forced into constructing a look-up table procedure which will succumb to combinatorial explosion.

0110	*carry*	
00011	$addend_1$	3
01011	$addend_2$	11
1110	*sum*	14

Figure 6.3 The results of computing f_+ on the input 3 and 11 (represented in binary). All of the data created during the computation are shown in italics.

As we did with parity, we will pursue the use of encoding symbols by employing the binary number system for integers. So what would be a compact procedure for determining f_+? One approach is essentially the method that we learn in grade school. The numbers (as usually represented in the decimal number system) are placed one on top of the other, right justified so that the numerals line up in columns.[4] Starting from the right, each column is added using another addition procedure, f_{++}, to produce a number. This is not an infinite regress, as the sub-procedure is itself a look-up table. Since there will be carries, we may have to handle adding three numbers in each column. To make this process uniform (and thereby simplify our procedure), we place a '0' at the top of the first column, and then for every column thereafter either a '0' if there is no carry, or a '1' if there is a carry. We also add a '0' to the left end of both addends so that we can handle one last carry in a uniform fashion. To add the three numbers in each column, we use functional composition with $f_{++} - f_{++}(f_{++}(carry\_bit, addend_1\_bit), addend_2\_bit)$ – to add the first two digits and then add this result to the third digit.

Below is a procedure that computes the addition function (f_+). Figure 6.3 shows the results of this computation on the inputs '0011' (3) and '1011' (11).

1 Place a '0' at the top of the first (rightmost) column and the left end of both $addend_1$ and $addend_2$.
2 Start with the rightmost column.
3 Add the top two numbers in the current column using f_{++}.
4 Add the result from State 3 to the bottom number in the current column using f_{++}. (Here we are using functional composition of f_{++} with itself.)
5 Place the first (rightmost) bit of the result from State 4 in the bottom row of the current column.
6 Place the second bit of the result from State 4 at the top of the column to the left of the current column. If there isn't a second bit, place a '0' there.
7 Move one column to the left.
8 If there are numbers in the current column, go back to state 3.
9 Output the bottom row. Halt.

[4] Note that relative spatial positioning allows one to discern the top number as the *first* addend and the bottom as the *second* addend. Addition being commutative, this distinction is not relevant; however for subtraction, for example, distinguishing between the minuend (the top number) and the subtrahend (the bottom number) is critical.

Table 6.3 Look-up table for f_{++}

a	b	$f_{++}(a, b)$
0	0	0
0	1	1
1	0	1
1	1	10
10	0	10
10	1	11

The procedure for f_{++}, which is used within f_{+}, can be implemented as a look-up table (Table 6.3). Note that this does *not* make f_{+} non-compact. The embedded look-up table does *not grow* as a function of the input; f_{++} only needs to deal with the addition of three bits. Our addends can increase without bound without changing how we deal with each column. Here, state memory is not only useful, it is necessary. All procedures require some state memory, just as they require some structure that is not a result of experience. This is a direct reflection of the fact that if any device is to receive information, it must have an *a priori* representation of the possible messages that it might receive.

The compact procedure f_{+} allows for the efficient addition of arbitrarily large integers. Like the compact procedure for f_{parity}, it does this by using compact symbols and taking advantage of the analytic decomposition of the referents. Whereas look-up-table approaches are agnostic as regards the encoding, compact procedures only function appropriately with appropriately encoded symbols. There is a tight bond between the encoding procedure that generates the symbols and the procedures that act on them.

When you get your nose down into the details of the procedures required to implement even something as simple as addition operating on compact symbols, it is somewhat surprising how complex they are. There is no question that the brain has a procedure (or possibly procedures) for adding symbols for simple quantities, like distance and duration. We review a small part of the relevant behavioral evidence in Chapters 11 through 13. Animals – even insects – can infer the distance and direction of one known location from another known location (Gallistel, 1990; 1998; Menzel et al., 2005). There is no way to do this without performing operations on vector-like symbols formally equivalent to vector subtraction (that is, addition with signed integers). Or, somewhat more cautiously, if the brain of the insect can compute the range and bearing of one known location from another known location without doing something homomorphic to vector addition, the discovery of how it does it will have profound mathematical implications.

It cannot be stressed too strongly that the procedure by which the brain of the insect does vector addition will depend on the form of the neurobiological symbols on which the procedure operates and the encoding function that maps from distances and directions to the forms of those symbols. If the form is unary – or, what is nearly the same thing, if addition is done on analog symbols – then the procedure

can be very simple. However, then we must understand how the brain can represent distances ranging from millimeters to kilometers using those unary symbols. To see the problem, one has simply to ponder why no symbolization of quantity that has any appreciable power uses unary symbols (hash marks). The Roman system starts out that way (I, II, III), but gives up after only three symbols. Adding (concatenating) hash marks is extremely simple but it does not appeal when one contemplates adding the hash-mark symbol for 511 to the hash-mark symbol for 10,324. Thus, the question of how the brain symbolizes simple quantities and its procedures/mechanisms for performing arithmetic operations on those quantities is a profoundly important and deeply interesting question, to which at this time neuroscience has no answer.

Two Senses of Knowing

In tracing our way through the details of the procedures for both f_{is_even} and f_{++}, we came upon a distinction between knowing in the symbolic sense and the "knowing" that is implicit in a stage (state) of a procedure. This is in essence the distinction between straightforward, transparent symbolic knowledge, and the indirect, opaque "knowing" that is characteristic of finite-state machines, which lack a symbolic read/write memory. Symbolic knowing is transparent because the symbols carry information gleaned from experience forward in time in a manner that makes it accessible to computation. The information needed to inform behavior is either explicit in the symbols that carry it forward or may be made explicit by computations that take those symbols as inputs. Contrast this with the procedural "knowing" that occurs, for example, in the search tree implementation of f_{is_even}. State 5 "knows" that the first bit in the input was a '0' and the second bit was a '1', not because it has symbols carrying this information but instead because the procedure would never have entered that state were that not the case. We, who are gods outside the procedure, can deduce this by scrutinizing the procedure, but the procedure does not symbolize these facts. It does not make them accessible to some other procedure.

We see in our compact procedure for f_{+} both forms of knowing. The look-up table sub-procedure for f_{++}, implemented as state memory, would only "know" what the first bit it received was by virtue of the fact that it was in a particular state. On the other hand, consider the knowing that takes place within the main procedure when it begins to add a new column. It knows what the carry bit is because that information is carried forward by a symbol (the bit) placed at the top of the current column earlier during the computation. f_{+} can be in State 3 with a '0' in the carry position or a '1' in the carry position. This information is known explicitly.

We put the state-based form of knowing in quotation marks, because it does not correspond to what is ordinarily understood by knowing. We do not place the symbolic form of knowing in quotation marks, both because it corresponds to the ordinary sense, and because we believe that this symbolic sense of knowing is the correct sense when we say such things as "the rat knows where it is" or "the bee knows the location of the nectar source" or "the jay knows when and where it cached what" (see later chapters).

It is important to be clear about these different senses of knowing, because they are closely related to a long-standing controversy within cognitive science and related fields. The anti-representational tradition, which is seen in essentially all forms of behaviorism, whether in psychology or philosophy or linguistics or neuroscience, regards all forms of learning as the learning of procedures. For early and pure expressions of this line of thought, see Hull (1930) and Skinner (1938, 1957). At least in its strongest form (Skinner, 1990), this line of thinking about the processes underlying behavior explicitly and emphatically rejects the assumption that there are symbols in the brain that encode experienced facts about the world (such as where things are and how long it takes food of a given kind to rot). By contrast, the assumption that there are such symbols and that they are central players in the causation of behavior is central to the what might be called mainline cognitive science (Chomsky, 1975; Fodor, 1975; Fodor & Pylyshyn, 1988; Marcus, 2001; Marr, 1982; Newell, 1980).

The anti-representational behaviorism of an earlier era finds an echo in contemporary connectionist and dynamic-systems work (P. M. Churchland, 1989; Edelman & Gally, 2001; Hoeffner, McClelland, & Seidenberg, 1996; Rumelhart & McClelland, 1986; Smolensky, 1991). Roughly speaking, the more committed theorists are to building psychological theory on neurobiological foundations, the more skeptical they are about the hypothesis that there are symbols and symbol-processing operations in the brain. We will explore the reasons for this in subsequent chapters, but the basic reason is simple: the language and conceptual framework for symbolic processing is alien to contemporary neuroscience. Neuroscientists cannot clearly identify the material basis for symbols – that is, there is no consensus about what the basis might be – nor can they specify the machinery that implements any of the information-processing operations that would plausibly act on those symbols (operations such as vector addition). Thus, there is a conceptual chasm between mainline cognitive science and neuroscience. Our book is devoted to exploring that chasm and building the foundations for bridging it.

A Geometric Example

The tight connection between procedures and the encodings that generate the symbols on which they operate is a point of the utmost importance. We have illustrated it so far with purely numerical operations in which the symbols referred to integers. This may seem too abstract a referent. Do the brains of animals represent numbers? Traditionally, the answer to this question has been, no, but research on animal cognition in recent years has shown that rats, pigeons, monkeys, and apes do in fact represent number per se (Biro & Matsuzawa, 1999; Boysen & Berntson, 1989; Brannon & Terrace, 2002; Cantlon & Brannon, 2005, 2006; Gallistel, 1990; Hauser, Carey, & Hauser, 2000; Matsuzawa & Biro, 2001; Rumbaugh & Washburn, 1993). Nonetheless, a non-numerical illustration involving symbols for something arguably less abstract and something whose representation is clearly a foundation of animal behavior is desirable. In our final example, we turn to the processing of geometric symbols, symbols for locations. There is overwhelming

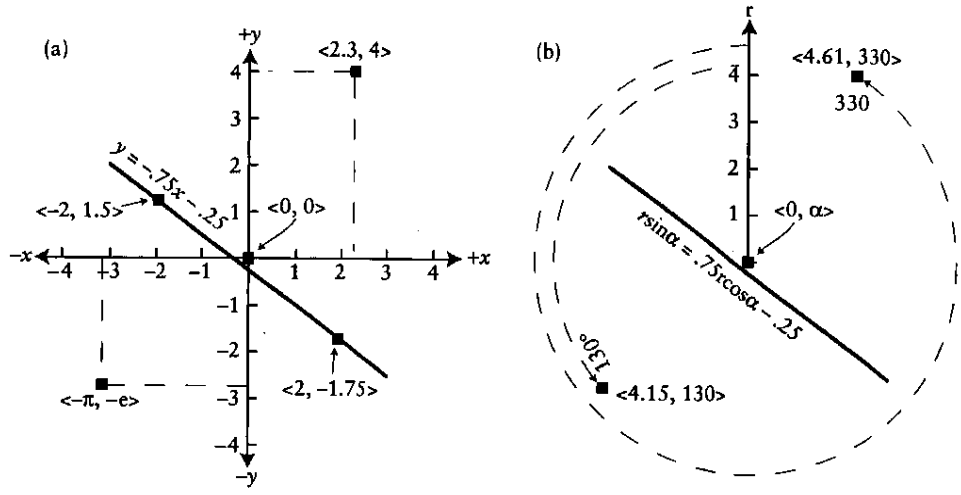

Figure 6.4 Two different ways of encoding locations and lines into vectors. (a) The Cartesian encoding. (b) Polar encoding.

behavioral evidence that animals represent locations, because the representation of locations is a *sine qua non* for effective navigation, and animals of all kinds, including most particularly insects, are gifted navigators (T. S. Collett, M. Collett, & Wehner, 2001; Gallistel, 1998; Gould, 1986; Menzel et al., 2005; Tautz et al., 2004; J. Wehner & Srinivasan, 2003; R. Wehner, Lehrer, & Harvey, 1996).

Like anything else, locations may be encoded in different ways. Whatever way they are encoded, the encoding will make some procedures simple and others complex. Which procedures are simple and which complex will depend on the encoding. The Cartesian encoding of locations (Figure 6.4a) decomposes a location into its signed (that is, directed) distances from two arbitrarily chosen orthogonal axes. An alternative is to decompose locations into a radial distance and an angular distance (Figure 6.4b). In navigation, this decomposition is called the range and bearing of a point from the origin. To effect this encoding of locations, we fix a point on the plane, called the origin or pole. The range is the distance of the location from this point. To be able to specify the bearings of locations, we draw a ray (line segment bounded at one end) from the pole running off arbitrarily far in some direction. This direction is often chosen with salient generally valid directional referents in mind, such as, for example, north, which is the point in the sky around which all of the heavenly bodies are seen to rotate, because it is the point toward which one end of the earth's axis of rotation points. This line is called the polar axis. The second coordinate in the polar encoding of location (the bearing of a location) is the angular distance through which we must rotate the polar axis in order for it to pass through that location. For the sake of familiarity, we will specify angular distances in degrees, even though radians would be preferred for computational purposes.

An awkwardness in this mapping is that there are an infinity of angular distances for any one location – thus, an infinite number of symbols that map to the same referent. First, we can rotate the polar axis either counterclockwise (as we do in Figure 6.4b) or clockwise. Either way, it will eventually pass through whatever location we are encoding, but the angular distance component of our vector symbol will have a different absolute value and a different sign, depending on which way we choose to rotate the polar axis. To forestall this, we may specify that the rotation must be, say, counterclockwise. This does not solve the problem of multiple symbols for the same referent because the polar axis will again pass through the point if we rotate it by an additional 360° or 720°, and so on. To prevent that, we must stipulate that only the smallest of the infinite set of angular distances is to be used as the second component of the symbol. Alternatively, we may use the sine and cosine of the bearing.

Another awkwardness of this encoding is that there is no specifiable angular distance for the polar point itself. Thus, the symbol for this point is different from the symbol for all other points. For some purposes, this is more than a little awkward. Nonetheless, for other purposes, this encoding is preferred because it makes the procedures for obtaining some very useful symbols extremely simple. Something that a navigator often wants to know is the distance and direction of a location (for example, the nest or the hive or the richest source of food or the nearest port when a storm threatens). Neither the distance nor the direction of a location from the origin (or from any other point) is explicit in the Cartesian encoding. They must be determined by means of a tolerably complex procedure applied to the vector that symbolizes the location. In Cartesian encoding, to determine the distance (range) of a point from the origin <0, 0> to the point <x, y>, we must compute $\sqrt{x^2 + y^2}$. To determine its direction (bearing), we must compute $\arcsin(y/x)$. By contrast, both quantities are explicit in the polar encoding of location. As in the case of determining parity from the binary encoding of an integer, we can read what we need directly from the symbol itself; the range (linear distance) of the location is represented by the first element of the vector symbol, the bearing (angular distance) by the second element. There are many other procedures that are simpler with the polar encoding than with the Cartesian encoding. On the other hand, there are many more procedures that are simpler with the Cartesian encoding than the polar encoding, which is why the Cartesian encoding is the default encoding.

The important point for our purpose is that if you change the encoding, then you must also change the procedures that are used to compute distance and everything else one wishes to compute. If one does not make suitable changes on the computational side, then the homomorphism breaks down; doing the computations no longer yields valid results. When you try to map from the computed symbols back to the entities to which they refer, it does not work. This point is of fundamental importance when considering proposed systems for establishing reference between activity in neurons or any other proposed neural symbol and the aspects of the world that the activity is supposed to represent. One must always ask, if that is the form that the symbols take, how does the computational side of the system work? What are the procedures that when applied to *those* symbols would extract behaviorally useful information?