# 3

# Turing Computability

*A function is* effectively computable *if there are definite, explicit rules by following which one could in principle compute its value for any given arguments. This notion will be further explained below, but even after further explanation it remains an intuitive notion. In this chapter we pursue the analysis of computability by introducing a rigorously defined notion of a* Turing-computable *function. It will be obvious from the definition that Turing-computable functions are effectively computable. The hypothesis that, conversely, every effectively computable function is Turing computable is known as* Turing's thesis. *This thesis is not obvious, nor can it be rigorously proved (since the notion of effective computability is an intuitive and not a rigorously defined one), but an enormous amount of evidence has been accumulated for it. A small part of that evidence will be presented in this chapter, with more in chapters to come. We first introduce the notion of Turing machine, give examples, and then present the official definition of what it is for a function to be computable by a Turing machine, or Turing computable.*

A superhuman being, like Zeus of the preceding chapter, could perhaps write out the whole table of values of a one-place function on positive integers, by writing each entry twice as fast as the one before; but for a human being, completing an infinite process of this kind is impossible in principle. Fortunately, for human purposes we generally do not need the whole table of values of a function $f$, but only need the values one at a time, so to speak: given some argument $n$, we need the value $f(n)$. If it is possible to produce the value $f(n)$ of the function $f$ for argument $n$ whenever such a value is needed, then that is almost as good as having the whole table of values written out in advance.

A function $f$ from positive integers to positive integers is called *effectively computable* if a list of instructions can be given that in principle make it possible to determine the value $f(n)$ for any argument $n$. (This notion extends in an obvious way to two-place and many-place functions.) The instructions must be completely definite and explicit. They should tell you at each step what to do, not tell you to go ask someone else what to do, or to figure out for yourself what to do: the instructions should require no external sources of information, and should require no ingenuity to execute, so that one might hope to automate the process of applying the rules, and have it performed by some mechanical device.

There remains the fact that for all but a finite number of values of $n$, it will be infeasible in practice for any human being, or any mechanical device, actually to carry out the computation: in principle it could be completed in a finite amount of time if we stayed in good health so long, or the machine stayed in working order so long; but in practice we will die, or the machine will collapse, long before the process is complete. (There is also a worry about finding enough space to store the intermediate results of the computation, and even a worry about finding enough matter to use in writing down those results: there's only a finite amount of paper in the world, so you'd have to writer smaller and smaller without limit; to get an infinite number of symbols down on paper, eventually you'd be trying to write on molecules, on atoms, on electrons.) But our present study will ignore these practical limitations, and work with an idealized notion of computability that goes beyond what actual people or actual machines can be sure of doing. Our eventual goal will be to prove that certain functions are *not* computable, *even if* practical limitations on time, speed, and amount of material could somehow be overcome, and for this purpose the essential requirement is that our notion of computability not be too *narrow*.

So far we have been sliding over a significant point. When we are given as argument a number $n$ or pair of numbers $(m, n)$, what we in fact are directly given is a *numeral* for $n$ or an ordered pair of *numerals* for $m$ and $n$. Likewise, if the value of the function we are trying to compute is a number, what our computations in fact end with is a *numeral* for that number. Now in the course of human history a great many systems of numeration have been developed, from the primitive *monadic* or *tally* notation, in which the number $n$ is represented by a sequence of $n$ strokes, through systems like Roman numerals, in which bunches of five, ten, fifty, one-hundred, and so forth strokes are abbreviated by special symbols, to the *Hindu–Arabic* or *decimal* notation in common use today. Does it make a difference in a definition of computability which of these many systems we adopt?

Certainly computations can be *harder in practice* with some notations than with others. For instance, multiplying numbers given in decimal numerals (expressing the product in the same form) is easier in practice than multiplying numbers given in something like Roman numerals. Suppose we are given two numbers, expressed in Roman numerals, say XXXIX and XLVIII, and are asked to obtain the product, also expressed in Roman numerals. Probably for most us the easiest way to do this would be first to translate from Roman to Hindu–Arabic—the rules for doing this are, or at least used to be, taught in primary school, and in any case can be looked up in reference works—obtaining 39 and 48. Next one would carry out the multiplication in our own more convenient numeral system, obtaining 1872. Finally, one would translate the result back into the inconvenient system, obtaining MDCCCLXXII. Doing all this is, of course, harder than simply performing a multiplication on numbers given by decimal numerals to begin with.

But the example shows that when a computation can be done in one notation, it is *possible in principle* to do in any other notation, simply by translating the data from the difficult notation into an easier one, performing the operation using the easier notation, and then translating the result back from the easier to the difficult notation. If a function is effectively computable when numbers are represented in one system of numerals, it will also be so when numbers are represented in any other system of numerals, provided only that translation between the systems can itself be carried out according to explicit rules, which is the case for any historical system of numeration that we have been able to decipher. (To say we have been able to decipher it amounts to saying that there are rules for translating back and forth between it and the system now in common use.) For purposes of framing a rigorously defined notion of computability, it is convenient to use monadic or tally notation.

A *Turing machine* is a specific kind of idealized machine for carrying out computations, especially computations on positive integers represented in monadic notation. We suppose that the computation takes place on a tape, marked into squares, which is unending in both directions—either because it is actually infinite or because there is someone stationed at each end to add extra blank squares as needed. Each square either is *blank*, or has a *stroke* printed on it. (We represent the blank by $S_0$ or 0 or most often $B$, and the stroke by $S_1$ or | or most often 1, depending on the context.) And with at most a finite number of exceptions, all squares are blank, both initially and at each subsequent stage of the computation.

At each stage of the computation, the computer (that is, the human or mechanical agent doing the computation) is *scanning* some one square of the tape. The computer is capable of erasing a stroke in the scanned square if there is one there, or of printing a stroke if the scanned square is blank. And he, she, or it is capable of movement: one square to the right or one square to the left at a time. If you like, think of the machine quite crudely, as a box on wheels which, at any stage of the computation, is over some square of the tape. The tape is like a railroad track; the ties mark the boundaries of the squares; and the machine is like a very short car, capable of moving along the track in either direction, as in Figure 3-1.

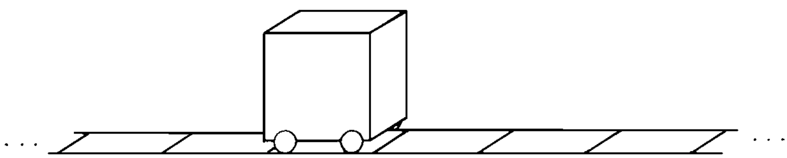

Figure 3-1. A Turing machine.

At the bottom of the car there is a device that can read what's written between the ties, and erase or print a stroke. The machine is designed in such a way that at each stage of the computation it is in one of a finite number of internal *states*, $q_1, \ldots, q_m$. Being in one state or another might be a matter of having one or another cog of a certain gear uppermost, or of having the voltage at a certain terminal inside the machine at one or another of $m$ different levels, or what have you: we are not concerned with the mechanics or the electronics of the matter. Perhaps the simplest way to picture the thing is quite crudely: inside the box there is a little man, who does all the reading and writing and erasing and moving. (The box has no bottom: the poor mug just walks along between the ties, pulling the box along.) This operator inside the machine has a list of $m$ instructions written down on a piece of paper and *is in state $q_i$ when carrying out instruction number $i$.*

Each of the instructions has conditional form: it tells what to do, depending on whether the symbol being scanned (the symbol in the scanned square) is the blank or

**3.2 Example** (Doubling the number of strokes). The machine starts off scanning the leftmost of a block of strokes on an otherwise blank tape, and winds up scanning the leftmost of a block of twice that many strokes on an otherwise blank tape. The flow chart is shown in Figure 3-5.

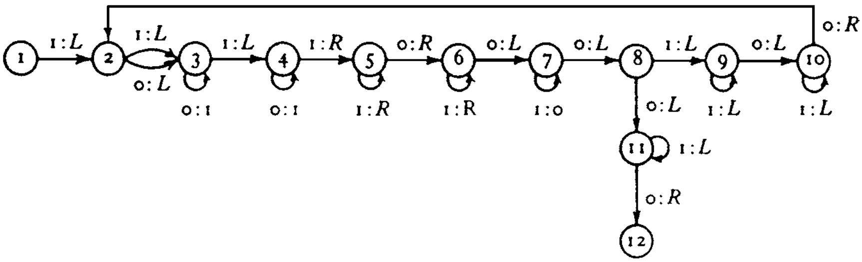


Figure 3-5. Doubling the number of strokes.

How does it work? In general, by writing double strokes at the left and erasing single strokes at the right. In particular, suppose the initial configuration is $1_111$, so that we start in state 1, scanning the leftmost of a block of three strokes on an otherwise blank tape. The next few configurations are as follows:

$$0_2111 \qquad 0_30111 \qquad 1_30111 \qquad 0_410111 \qquad 1_410111.$$

So we have written our first double stroke at the left—separated from the original block 111 by a blank. Next we go right, past the blank to the right-hand end of the original block, and erase the rightmost stroke. Here is how that works, in two phases. Phase 1:

$$11_50111 \qquad 110_5111 \qquad 1101_611 \qquad 11011_61 \qquad 110111_6 \qquad 1101110_6.$$

Now we know that we have passed the last of the original block of strokes, so (phase 2) we back up, erase one of them, and move one more square left:

$$110111_7 \qquad 110110_7 \qquad 11011_80.$$

Now we hop back left, over what is left of the original block of strokes, over the blank separating the original block from the additional strokes we have printed, and over those additional strokes, until we find the blank beyond the leftmost stroke:

$$1101_91 \qquad 110_911 \qquad 11_{10}011 \qquad 1_{10}1011 \qquad 0_{10}11011.$$

Now we will print another two new strokes, much as before:

$$01_21011 \qquad 0_311011 \qquad 1_311011 \qquad 0_4111011 \qquad 1_4111011.$$

We are now back on the leftmost of the block of newly printed strokes, and the process that led to finding and erasing the rightmost stroke will be repeated, until we arrive at the following:

$$1111011_7 \qquad 1111010_7 \qquad 111101_80.$$

Another round of this will lead first to writing another pair of strokes:

$$1_41111101.$$

It will then lead to erasing the last of the original block of strokes:

$$1111110 1_7 \qquad 1111110 0_7 \qquad 1111110_8 0.$$

And now the endgame begins, for we have what we want on the tape, and need only move back to halt on the leftmost stroke:

$$111111_{11} \qquad 11111_{11}1 \qquad 1111_{11}11 \qquad 111_{11}111 \qquad 11_{11}1111 \qquad 1_{11}11111$$
$$0_{11}111111 \qquad 1_{12}11111.$$

Now we are in state 12, scanning a stroke. Since there is no arrow from that node telling us what to do in such a case, we halt. The machine performs as advertised.

(*Note*: The fact that the machine doubles the number of strokes when the original number is three is not a proof that the machine performs as advertised. But our examination of the special case in which there are three strokes initially made no essential use of the fact that the initial number was three: it is readily converted into a proof that the machine doubles the number of strokes no matter how long the original block may be.)

Readers may wish, in the remaining examples, to try to design their own machines before reading our designs; and for this reason we give the statements of all the examples first, and collect all the proofs afterward.

**3.3 Example** (Determining the parity of the length of a block of strokes). There is a Turing machine that, started scanning the leftmost of an unbroken block of strokes on an otherwise blank tape, eventually halts, scanning a square on an otherwise blank tape, where the square contains a blank or a stroke depending on whether there were an even or an odd number of strokes in the original block.

**3.4 Example** (Adding in monadic (tally) notation). There is a Turing machine that does the following. Initially, the tape is blank except for two solid blocks of strokes, say a left block of $p$ strokes and a right block of $q$ strokes, separated by a single blank. Started on the leftmost blank of the left block, the machine eventually halts, scanning the leftmost stroke in a solid block of $p + q$ stokes on an otherwise blank tape.

**3.5 Example** (Multiplying in monadic (tally) notation). There is a Turing machine that does the same thing as the one in the preceding example, but with $p \cdot q$ in place of $p + q$.

**Proofs**

*Example 3.3.* A flow chart for such a machine is shown in Figure 3-6.

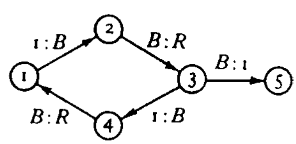


Figure 3-6. Parity machine.

If there were 0 or 2 or 4 or ... strokes to begin with, this machine halts in state 1, scanning a blank on a blank tape; if there were 1 or 3 or 5 or ..., it halts in state 5, scanning a stroke on an otherwise blank tape.

*Example 3.4.* The object is to erase the leftmost stroke, fill the gap between the two blocks of strokes, and halt scanning the leftmost stroke that remains on the tape. Here is one way of doing it, in quadruple notation: $q_1 S_1 S_0 q_1$; $q_1 S_0 R q_2$; $q_2 S_1 R q_2$; $q_2 S_0 S_1 q_3$; $q_3 S_1 L q_3$; $q_3 S_0 R q_4$.

*Example 3.5.* A flow chart for a machine is shown in Figure 3-7.

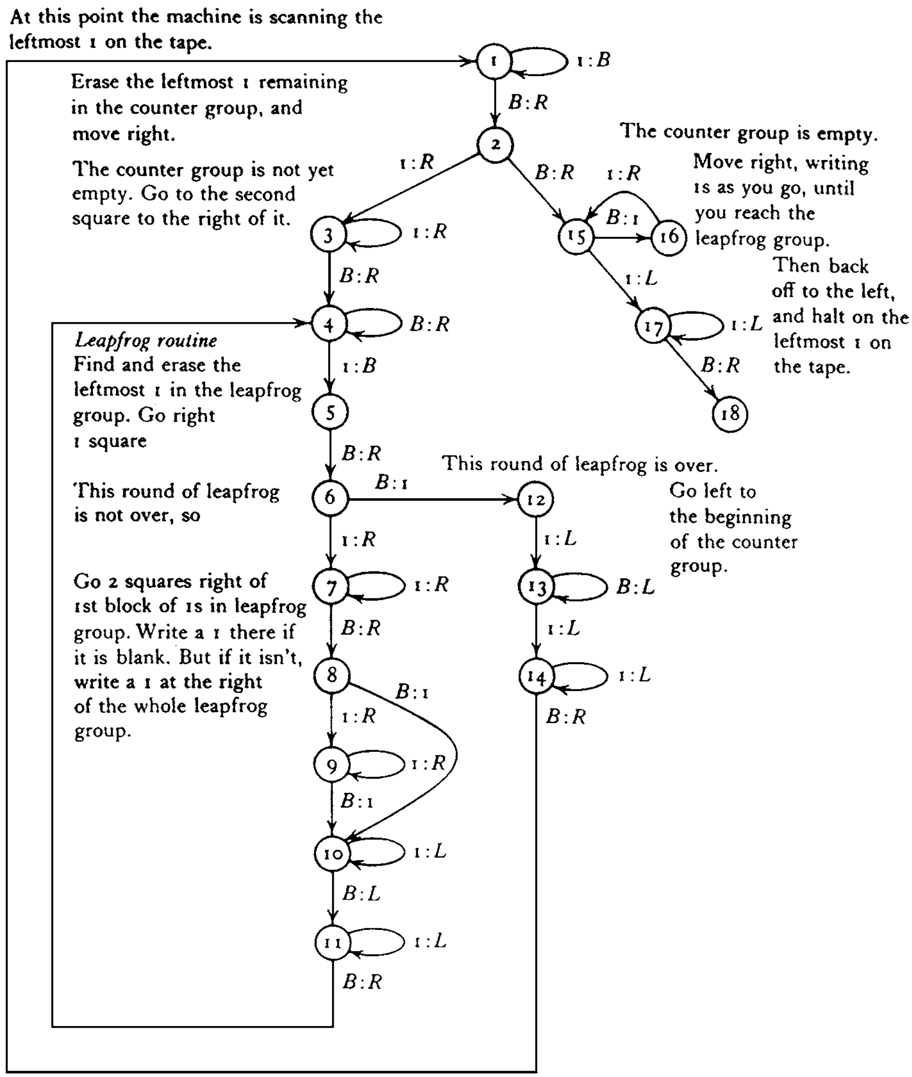


Figure 3-7. Multiplication machine.

Here is how the machine works. The first block, of $p$ strokes, is used as a *counter*, to keep track of how many times the machine has added $q$ strokes to the group at the right. To start, the machine erases the leftmost of the $p$ strokes and sees if there are any strokes left in the counter group. If not, $pq = q$, and all the machine has to do is position itself over the leftmost stroke on the tape, and halt.

But if there are any strokes left in the counter, the machine goes into a *leapfrog routine*: in effect, it moves the block of $q$ strokes (the *leapfrog group*) $q$ places to the right along the tape. For example, with $p = 2$ and $q = 3$ the tape looks like this initially:

$$11B111$$

and looks like this after going through the leapfrog routine:

$$B1BBBB111.$$

The machine will then note that there is only one 1 left in the counter, and will finish up by erasing that 1, moving right two squares, and changing all $B$s to strokes until it comes to a stroke, at which point it continues to the leftmost 1 and halts.

The general picture of how the leapfrog routine works is shown in Figure 3-8.

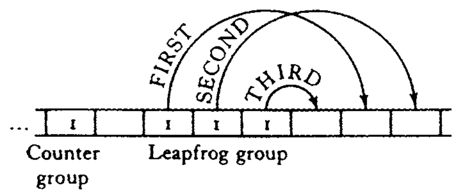


Figure 3-8. Leapfrog.

In general, the leapfrog group consists of a block of 0 or 1 or ... or $q$ strokes, followed by a blank, followed by the remainder of the $q$ strokes. The blank is there to tell the machine when the leapfrog game is over: without it the group of $q$ strokes would keep moving right along the tape forever. (In playing leapfrog, the portion of the $q$ strokes to the left of the blank in the leapfrog group functions as a counter: it controls the process of adding strokes to the portion of the leapfrog group to the right of the blank. That is why there are two big loops in the flow chart: one for each counter-controlled subroutine.)

We have not yet given an official definition of what it is for a numerical function to be computable by a Turing machine, specifying how inputs or arguments are to be represented on the machine, and how outputs or values represented. Our specifications for a $k$-place function from positive integers to positive integers are as follows:

**(a)** The arguments $m_1, \ldots, m_k$ of the function will be represented in monadic notation by blocks of those numbers of strokes, each block separated from the next by a single blank, on an otherwise blank tape. Thus, at the beginning of the computation of, say, $3 + 2$, the tape will look like this: $111B11$.

**(b)** Initially, the machine will be scanning the leftmost 1 on the tape, and will be in its initial state, state 1. Thus in the computation of $3 + 2$, the initial configuration will be $1_1 11B11$. A configuration as described by (a) and (b) is called a *standard initial configuration* (or *position*).

**(c)** If the function that is to be computed assigns a value $n$ to the arguments that are represented initially on the tape, then the machine will eventually halt on a tape containing a block of that number of strokes, and otherwise blank. Thus in the computation of $3 + 2$, the tape will look like this: 11111.

**(d)** In this case, the machine will halt scanning the leftmost 1 on the tape. Thus in the computation of $3 + 2$, the final configuration will be $1_n 1111$, where $n$th state is one for which there is no instruction what to do if scanning a stroke, so that in this configuration the machine will be halted. A configuration as described by (c) and (d) is called a *standard final configuration* (or *position*).

**(e)** If the function that is to be computed assigns no value to the arguments that are represented initially on the tape, then the machine either will never halt, or will halt in some nonstandard configuration such as $B_n 11111$ or $B11_n 111$ or $B11111_n$.

The restriction above to the standard position (scanning the leftmost 1) for starting and halting is inessential, but *some* specifications or other have to be made about initial and final positions of the machine, and the above assumptions seem especially simple.

With these specifications, any Turing machine can be seen to compute a function of one argument, a function of two arguments, and, in general, a function of $k$ arguments for each positive integer $k$. Thus consider the machine specified by the single quadruple $q_1 1 1 q_2$. Started in a standard initial configuration, it immediately halts, leaving the tape unaltered. If there was only a single block of strokes on the tape initially, its final configuration will be standard, and thus this machine computes the identity function id of one argument: $\text{id}(m) = m$ for each positive integer $m$. Thus the machine computes a certain total function of one argument. But if there were two or more blocks of strokes on the tape initially, the final configuration will not be standard. Accordingly, the machine computes the extreme partial function of two arguments that is undefined for all pairs of arguments: the empty function $e_2$ of two arguments. And in general, for $k$ arguments, this machine computes the empty function $e_k$ of $k$ arguments.

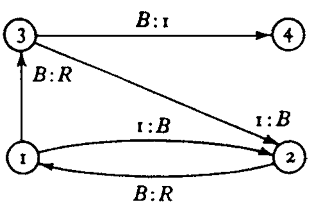


Figure 3-9. A machine computing the value 1 for all arguments.

By contrast, consider the machine whose flow chart is shown in Figure 3-9. This machine computes for each $k$ the total function that assigns the same value, namely 1, to each $k$-tuple. Started in initial state 1 in a standard initial configuration, this machine erases the first block of strokes (cycling between states 1 and 2 to do so) and goes to state 3, scanning the second square to the right of the first block. If it sees a blank there, it knows it has erased the whole tape, and so prints a single 1 and halts in state 4, in a standard configuration. If it sees a stroke there, it re-enters the cycle between states 1 and 2, erasing the second block of strokes and inquiring again, in state 3, whether the whole tape is blank, or whether there are still more blocks to be dealt with.

A numerical function of $k$ arguments is *Turing computable* if there is some Turing machine that computes it in the sense we have just been specifying. Now computation in the Turing-machine sense is certainly *one* kind of computation in the intuitive sense, so all Turing-computable functions are effectively computable. *Turing's thesis* is that, conversely, any effectively computable function is Turing computable, so that computation in the precise technical sense we have been developing coincides with effective computability in the intuitive sense.

It is easy to imagine liberalizations of the notion of the Turing machine. One could allow machines using more symbols than just the blank and the stroke. One could allow machines operating on a rectangular grid, able to move up or down a square as well as left or right. Turing's thesis implies that no liberalization of the notion of Turing machine will enlarge the class of functions computable, because all functions that are effectively computable in any way at all are already computable by a Turing machine of the restricted kind we have been considering. Turing's thesis is thus a bold claim.

It *is* possible to give a heuristic argument for it. After all, effective computation consists of moving around and writing and perhaps erasing symbols, according to definite, explicit rules; and surely writing and erasing symbols can be done stroke by stroke, and moving from one place to another can be done step by step. But the main argument will be the accumulation of examples of effectively computable functions that we succeed in showing are Turing computable. So far, however, we have had just a few examples of Turing machines computing numerical functions, that is, of effectively computable functions that we have proved to be Turing computable: addition and multiplication in the preceding section, and just now the identity function, the empty function, and the function with constant value 1.

Now addition and multiplication are just the first two of a series of arithmetic operations all of which are effectively computable. The next item in the series is exponentiation. Just as multiplication is repeated addition, so exponentiation is repeated multiplication. (Then repeated exponentiation gives a kind of super-exponentiation, and so on. We will investigate this general process of defining new functions from old in a later chapter.) If Turing's thesis is correct, there must be a Turing machine for each of these functions, computing it. Designing a multiplier was already difficult enough to suggest that designing an exponentiator would be quite a challenge, and in any case, the direct approach of designing a machine for each operation would take us forever, since there are infinitely many operations in the series. Moreover, there are many other effectively computable numerical functions besides the ones in this series. When we return, in the chapter after next, to the task of showing various effectively computable numerical functions to be Turing computable, and thus accumulating evidence for Turing's thesis, a less direct approach will be adopted, and all the operations in the series that begins with addition and multiplication will be shown to be Turing computable in one go.

For the moment, we set aside the positive task of showing functions to be Turing computable and instead turn to examples of numerical functions of one argument that are Turing *un*computable (and so, if Turing's thesis is correct, effectively uncomputable).

## Problems

**3.1** Consider a tape containing a block of $n$ strokes, followed by a space, followed by a block of $m$ strokes, followed by a space, followed by a block of $k$ strokes, and otherwise blank. Design a Turing machine that when started on the leftmost stroke will eventually halt, having neither printed nor erased anything . . .

**(a)** . . . on the leftmost stroke of the second block.

**(b)** . . . on the leftmost stroke of the third block.

**3.2** Continuing the preceding problem, design a Turing machine that when started on the leftmost stroke will eventually halt, having neither printed nor erased anything . . .

**(a)** . . . on the rightmost stroke of the second block.

**(b)** . . . on the rightmost stroke of the third block.

**3.3** Design a Turing machine that, starting with the tape as in the preceding problems, will eventually halt on the leftmost stroke on the tape, which is now to contain a block of $n$ strokes, followed by a blank, followed by a block of $m + 1$ strokes, followed by a blank, followed by a block of $k$ strokes.

**3.4** Design a Turing machine that, starting with the tape as in the preceding problems, will eventually halt on the leftmost stroke on the tape, which is now to contain a block of $n$ strokes, followed by a blank, followed by a block of $m - 1$ strokes, followed by a blank, followed by a block of $k$ strokes.

**3.5** Design a Turing machine to compute the function $\min(x, y) =$ the smaller of $x$ and $y$.

**3.6** Design a Turing machine to compute the function $\max(x, y) =$ the larger of $x$ and $y$.

# 4

# Uncomputability

*In the preceding chapter we introduced the notion of* Turing computability. *In the present short chapter we give examples of Turing-uncomputable functions: the* halting function *in section 4.1, and the* productivity function *in the optional section 4.2. If Turing's thesis is correct, these are actually examples of effectively uncomputable functions.*

## 4.1 The Halting Problem

There are too many functions from positive integers to positive integers for them all to be Turing computable. For on the one hand, as we have seen in problem 2.2, the set of all such functions is nonenumerable. And on the other hand, the set of Turing machines, and therefore of Turing-computable functions, is enumerable, since the representation of a Turing machine in the form of quadruples amounts to a representation of it by a finite string of symbols from a finite alphabet; and we have seen in Chapter 1 that the set of such strings is enumerable. These considerations show us that there must exist functions that are not Turing computable, but they do not provide an explicit example of such a function. To provide explicit examples is the task of this chapter. We begin simply by examining the argument just given in slow motion, with careful attention to details, so as to extract a specific example of a Turing-uncomputable function from it.

To begin with, we have suggested that we can enumerate the Turing-computable functions of one argument by enumerating the Turing machines, and that we can enumerate the Turing machines using their quadruple representations. As we turn to details, it will be convenient to modify the quadruple representation used so far somewhat. To indicate the nature of the modifications, consider the machine in Figure 3-9 in the preceding chapter. Its quadruple representation would be

$$q_1 S_0 R q_3,\ q_1 S_1 S_0 q_2,\ q_2 S_0 R q_1,\ q_3 S_0 S_1 q_4,\ q_3 S_1 S_0 q_2.$$

We have already been taking the lowest-numbered state $q_1$ to be the initial state. We now want to assume that the highest-numbered state is a halted state, for which there are no instructions and no quadruples. This is already the case in our example, and if it were not already so in some other example, we could make it so by adding one additional state.

We now also want to assume that for *every* state $q_i$ except this highest-numbered halted state, and for *each* of the two symbols $S_j$ we are allowing ourselves to use, namely $S_0 = B$ and $S_1 = 1$, there is a quadruple beginning $q_i S_j$. This is not so in our example as it stands, where there is no instruction for $q_2 S_1$. We have been interpreting the absence of an instruction for $q_i S_j$ as an instruction to halt, but the same effect could be achieved by giving an explicit instruction to keep the same symbol and then go to the highest-numbered state. When we modify the representation by adding this instruction, the representation becomes

$$q_1 S_0 R q_3,\ q_1 S_1 S_0 q_2,\ q_2 S_0 R q_1,\ q_2 S_1 S_1 q_4,\ q_3 S_0 S_1 q_4,\ q_3 S_1 S_0 q_2.$$

Now taking the quadruples beginning $q_1 S_0$, $q_1 S_1$, $q_2 S_0, \ldots$ in that order, as we have done, the first two symbols of each quadruple are predictable and therefore do not need to be written. So we may simply write

$$R q_3,\ S_0 q_2,\ R q_1,\ S_1 q_4,\ S_1 q_4,\ S_0 q_2.$$

Representing $q_i$ by $i$, and $S_j$ by $j + 1$ (so as to avoid 0), and $L$ and $R$ by 3 and 4, we can write still more simply

$$4, 3, 1, 2, 4, 1, 2, 4, 2, 4, 1, 2.$$

Thus the Turing machine can be completely represented by a finite sequence of positive integers—and even, if desired, by a single positive integer, say using the method of coding based on prime decomposition:

$$2^4 \cdot 3^3 \cdot 5 \cdot 7^2 \cdot 11^4 \cdot 13 \cdot 17^2 \cdot 19^4 \cdot 23^2 \cdot 29^4 \cdot 31 \cdot 37^2.$$

Not every positive integer will represent a Turing machine: whether a given positive integer does so or not depends on what the sequence of exponents in its prime decomposition is, and not every finite sequence represents a Turing machine. Those that do must have length some multiple $4n$ of 4, and have among their odd-numbered entries only numbers 1 to 4 (representing $B$, 1, $L$, $R$) and among their even-numbered entries only numbers 1 to $n + 1$ (representing the initial state $q_1$, various other states $q_i$, and the halted state $q_{n+1}$). But no matter: from the above representation we at least get a gappy listing of all Turing machines, in which each Turing machine is listed at least once, and on filling in the gaps we get a gapless list of all Turing machines, $M_1, M_2, M_3, \ldots$, and from this a similar list of all Turing-computable functions of one argument, $f_1, f_2, f_3, \ldots$, where $f_i$ is the total or partial function computed by $M_i$.

To give a trivial example, consider the machine represented by (1, 1, 1, 1), or $2 \cdot 3 \cdot 5 \cdot 7 = 210$. Started scanning a stroke, it erases it, then leaves the resulting blank alone and remains in the same initial state, never going to the halted state, which would be state 2. Or consider the machine represented by (2, 1, 1, 1) or $2^2 \cdot 3 \cdot 5 \cdot 7 = 420$. Started scanning a stroke, it erases it, then prints it back again, then erases it, then prints it back again, and so on, again never halting. Or consider the machine represented by (1, 2, 1, 1), or $2 \cdot 3^2 \cdot 5 \cdot 7 = 630$. Started scanning a stroke, it erases it, then goes to the halted state 2 when it scans the resulting blank, which means halting in a nonstandard final configuration. A little thought shows that 210, 420, 630 are the smallest numbers that represent Turing machines, so the three machines just described will be $M_1$, $M_2$, $M_3$, and we have $f_1 = f_2 = f_3 =$ the empty function.

We have now indicated an explicit enumeration of the Turing-computable functions of one argument, obtained by enumerating the machines that compute them. The fact that such an enumeration is possible shows, as we remarked at the outset, that there must exist Turing-uncomputable functions of a single argument. The point of actually specifying one such enumeration is to be able to exhibit a particular such function. To do so, we define a *diagonal function* $d$ as follows:

$$d(n) = \begin{cases} 2 & \text{if } f_n(n) \text{ is defined and } = 1 \\ 1 & \text{otherwise.} \end{cases} \tag{1}$$

Now $d$ is a perfectly genuine total function of one argument, but it is not Turing computable, that is, $d$ is neither $f_1$ nor $f_2$ nor $f_3$, and so on. *Proof*: Suppose that $d$ is one of the Turing computable functions—the $m$th, let us say. Then for each positive integer $n$, either $d(n)$ and $f_m(n)$ are both defined and equal, or neither of them is defined. But consider the case $n = m$:

$$f_m(m) = d(m) = \begin{cases} 2 & \text{if } f_m(m) \text{ is defined and } = 1 \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

Then whether $f_m(m)$ is or is not defined, we have a contradiction: Either $f_m(m)$ is undefined, in which case (2) tells us that it is defined and has value 1; or $f_m(m)$ is defined and has a value $\neq 1$, in which case (2) tells us it has value 1; or $f_m(m)$ is defined and has value 1, in which case (2) tells us it has value 2. Since we have derived a contradiction from the assumption that $d$ appears somewhere in the list $f_1, f_2, \ldots, f_m, \ldots$, we may conclude that the supposition is false. We have proved:

**4.1 Theorem.** The diagonal function $d$ is not Turing computable.

According to Turing's thesis, since $d$ is not Turing computable, $d$ cannot be effectively computable. Why not? After all, although no Turing machine computes the function $d$, *we* were able compute at least its first few values. For since, as we have noted, $f_1 = f_2 = f_3 =$ the empty function we have $d(1) = d(2) = d(3) = 1$. And it may seem that we can actually compute $d(n)$ for any positive integer $n$—if we don't run out of time.

Certainly it is straightforward to discover which quadruples determine $M_n$ for $n = 1, 2, 3$, and so on. (This is straightforward in principle, though eventually humanly infeasible in practice because the duration of the trivial calculations, for large $n$, exceeds the lifetime of a human being and, in all probability, the lifetime of the human race. But in our idealized notion of computability, we ignore the fact that human life is limited.)

And certainly it is perfectly routine to follow the operations of $M_n$, once the initial configuration has been specified; and if $M_n$ does eventually halt, we must eventually get that information by following its operations. Thus if we start $M_n$ with input $n$ and it does halt with that input, then by following its operations until it halts, we can see whether it halts in nonstandard position, leaving $f_n(n)$ undefined, or halts in standard position with output $f_n(n) = 1$, or halts in standard position with output $f_n(n) \neq 1$. In the first or last cases, $d(n) = 1$, and in the middle case, $d(n) = 2$.

But there is yet another case where $d(n) = 1$; namely, the case where $M_n$ never halts at all. If $M_n$ is destined never to halt, given the initial configuration, can we find *that* out in a finite amount of time? This is the essential question: determining whether machine $M_n$, started scanning the leftmost of an unbroken block of $n$ strokes on an otherwise blank tape, does or does not eventually halt.

Is *this* perfectly routine? Must there be some point in the routine process of following its operations at which it becomes clear that it will never halt? In simple cases this is so, as we saw in the cases of $M_1$, $M_2$, and $M_3$ above. But for the function $d$ to be effectively computable, there would have to be a *uniform* mechanical procedure, applicable not just in these simple cases but also in more complicated cases, for discovering whether or not a given machine, started in a given configuration, will ever halt.

Thus consider the multiplier in Example 3.5. Its sequential representation would be a sequence of 68 numbers, each $\leq$18. It is routine to verify that it represents a Turing machine, and one can easily enough derive from it a flow chart like the one shown in Figure 3-7, but *without the annotations, and of course without the accompanying text*. Suppose one came upon such a sequence. It would be routine to check whether it represented a Turing machine and, if so, again to derive a flow chart *without annotations and accompanying text*. But is there a uniform method or mechanical routine that, in this and much more complicated cases, allows one to determine from inspecting the flow chart, *without any annotations or accompanying text*, whether the machine eventually halts, once the initial configuration has been specified?

If there is such a routine, Turing's thesis is erroneous: if Turing's thesis is correct, there can be no such routine. At present, several generations after the problem was first posed, no one has yet succeeded in describing any such routine—a fact that must be considered some kind of evidence in favor of the thesis.

Let us put the matter another way. A function closely related to $d$ is the *halting function h* of two arguments. Here $h(m, n) = 1$ or 2 according as machine $m$, started with input $n$, eventually halts or not. If $h$ were effectively computable, $d$ would be effectively computable. For given $n$, we could first compute $h(n, n)$. If we got $h(n, n) = 2$, we would know that $d(n) = 1$. If we got $h(n, n) = 1$, we would know that we could safely start machine $M_n$ in stardard initial configuration for input $n$, and that it would eventually halt. If it halted in nonstandard configuration, we would again have $d(n) = 1$. If it halted in standard final configuration giving an output $f_n(n)$, it would have $d(n) = 1$ or 2 according as $f_n(n) \neq 1$ or $= 1$.

This is an informal argument showing that if $h$ were effectively computable, then $d$ would be effectively computable. Since we have shown that $d$ is not Turing computable, assuming Turing's thesis it follows that $d$ is not effectively computable, and hence that $h$ is not effectively computable, and so not Turing computable. It is also possible to prove rigorously, though we do not at this point have the apparatus needed to do so, that if $h$ were Turing computable, then $d$ would be Turing computable, and since we have shown that $d$ is *not* Turing computable, this would show that $h$ is not

Turing computable. Finally, it is possible to prove rigorously in another way, not involving $d$, that $h$ is not Turing computable, and this we now do.

**4.2 Theorem.** The halting function $h$ is not Turing computable.

*Proof*: By way of background we need two special Turing machines. The first is a *copying machine* $C$, which works as follows. Given a tape containing a block of $n$ strokes, and otherwise blank, if the machine is started scanning the leftmost stroke on the tape, it will eventually halt with the tape containing two blocks of $n$ strokes separated by a blank, and otherwise blank, with the machine scanning the leftmost stroke on the tape. Thus if the machine is started with

$$\ldots BBB1111BBB\ldots$$

it will halt with

$$\ldots BBB1111B1111BBB\ldots$$

eventually. We ask you to design such a machine in the problems at the end of this chapter (and give you a pretty broad hint how to do it at the end of the book).

The second is a *dithering machine* $D$. Started on the leftmost of a block of $n$ strokes on an otherwise blank tape, $D$ eventually halts if $n > 1$, but never halts if $n = 1$. Such a machine is described by the sequence

$$1, 3, 4, 2, 3, 1, 3, 3.$$

Started on a stroke in state 1, it moves right and goes into state 2. If it finds itself on a stroke, it moves back left and halts, but if it finds itself on a blank, it moves back left and goes into state 1, starting an endless back-and-forth cycle.

Now suppose we had a machine $H$ that computed the function $h$. We could *combine* the machines $C$ and $H$ as follows: if the states of $C$ are numbered 1 through $p$, and the states of $H$ are numbered 1 through $q$, renumber the latter states $p + 1$ through $r = p + q$, and write these renumbered instructions after the instructions for $C$. Originally, $C$ tells us to halt by telling us to go into state $p + 1$, but in the new combined instructions, going into state $p + 1$ means not halting, but beginning the operations of machine $H$. So the new combined instructions will have us first go through the operations of $C$, and then, when $C$ would have halted, go through the operations of $H$. The result is thus a machine $G$ that computes the function $g(n) = h(n, n)$.

We now combine *this* machine $G$ with the dithering machine $D$, renumbering the states of the latter as $r + 1$ and $r + 2$, and writing its instructions after those for $G$. The result will be a machine $M$ that goes through the operations of $G$ and then the operations of $D$. Thus if machine number $n$ halts when started on its own number, that is, if $h(n, n) = g(n) = 1$, then the machine $M$ does *not* halt when started on that number $n$, whereas if machine number $n$ does *not* halt when started on its own number, that is, if $h(n, n) = g(n) = 2$, then machine $M$ *does* halt when started on $n$.

But of course there can be no such machine as $M$. For what would it do if started with input its own number $m$? It would halt if and only if machine number $m$, which is to say itself, does *not* halt when started with input the number $m$. This contradiction shows there can be no such machine as $H$.

The *halting problem* is to find an effective procedure that, given any Turing machine $M$, say represented by its number $m$, and given any number $n$, will enable us to determine whether or not that machine, given that number as input, ever halts. For the problem to be solvable by a Turing machine would require there to be a Turing machine that, given $m$ and $n$ as inputs, produces as its output the answer to the question whether machine number $m$ with input $n$ ever halts. Of course, a Turing machine of the kind we have been considering could not produce the output by printing the word 'yes' or 'no' on its tape, since we are considering machines that operate with just two symbols, the blank and the stroke. Rather, we take the affirmative answer to be presented by an output of 1 and the negative by an output of 2. With this understanding, the question whether the halting problem can be solved by a Turing machine amounts to the question whether the halting function $h$ is Turing computable, and we have just seen in Theorem 4.2 that it is not. That theorem, accordingly, is often quoted in the form: 'The halting problem is not solvable by a Turing machine.' Assuming Turing's thesis, it follows that it is not solvable at all.

Thus far we have two examples of functions that are not Turing computable—or problems that are not solvable by any Turing machine—and if Turing's thesis is correct, these functions are not effectively computable. A further example is given in the next section. Though working through the example will provide increased familiarity with the potential of Turing machines that will be desirable when we come to the next chapter, and in any case the example is a beautiful one, still none of the material connected with this example is strictly speaking indispensable for any of our further work; and therefore we have starred the section in which it appears as optional.

## 4.2* The Productivity Function

Consider a $k$-state Turing machine, that is, a machine with $k$ states (not counting the halted state). Start it with input $k$, that is, start it in its initial state on the leftmost of a block of $k$ strokes on an otherwise blank tape. If the machine never halts, or halts in nonstandard position, give it a score of zero. If it halts in standard position with output $n$, that is, on the leftmost of a block of $n$ strokes on an otherwise blank tape, give it a score of $n$. Now define $s(k)$ = the highest score achieved by any $k$-state Turing machine. This function can be shown to be Turing uncomputable.

We first show that if the function $s$ were Turing computable, then so would be the function $t$ given by $t(k) = s(k) + 1$. For supposing we have a machine that computes $s$, we can modify it as follows to get a machine, having one more state than the original machine, that computes $t$. Where the instructions for the original machine would have it halt, the instructions for the new machine will have it go into the new, additional state. In this new state, if the machine is scanning a stroke, it is to move one square to the left, remaining in the new state; while if it is scanning a blank, it is to print a stroke and halt. A little thought shows that a computation of the new machine will

# 11

# The Undecidability of First-Order Logic

*This chapter connects our work on computability with questions of logic. Section 11.1 presupposes familiarity with the notions of logic from Chapter 9 and 10 and of Turing computability from Chapters 3–4, including the fact that the halting problem is not solvable by any Turing machine, and describes an effective procedure for producing, given any Turing machine M and input n, a set of sentences* $\Gamma$ *and a sentence D such that M given input n will eventually halt if and only if* $\Gamma$ *implies D. It follows that if there were an effective procedure for deciding when a finite set of sentences implies another sentence, then the halting problem would be solvable; whereas, by Turing's thesis, the latter problem is* not *solvable, since it is not solvable by a Turing machine. The upshot is, one gets an argument, based on Turing's thesis for (the* Turing–Büchi proof *of)* Church's theorem, *that the decision problem for implication is not effectively solvable. Section 11.2 presents a similar argument—the* Gödel-style proof *of Church's theorem—this time using not Turing machines and Turing's thesis, but primitive recursive and recursive functions and Church's thesis, as in Chapters 6–7. The constructions of the two sections, which are independent of each other, are both instructive; but an entirely different proof, not dependent on Turing's or Church's thesis, will be given in a later chapter, and in that sense the present chapter is optional. (After the present chapter we return to pure logic for the space of several chapters, to resume to the application of computability theory to logic with Chapter 15.)*

### 11.1 Logic and Turing Machines

We are going to show how, given the machine table or flow chart or other suitable presentation of a Turing machine, and any $n$, we can effectively write down a *finite* set of sentences $\Gamma$ and a sentence $D$ such that $\Gamma$ implies $D$ if and only if the machine in question does eventually halt when started with input $n$, that is, when started in its initial state scanning the leftmost of a block of $n$ strokes on an otherwise blank tape. It follows that if the *decision problem* for logical implication could be solved, that is, if an effective method could be devised that, applied to any finite set of sentences $\Gamma$ and sentence $D$, would in a finite amount of time tell us whether or not $\Gamma$ implies $D$, then the *halting problem* for Turing machines could be solved, or in other words, an effective method would exist that, applied to any suitably presented Turing machine and number $n$, would in a finite amount of time tell us whether or not that machine halts when started with input $n$. Since we have seen in Chapter 4 that, assuming Turing's thesis, the halting problem is *not* solvable, it follows that, again assuming Turing's thesis, the decision problem is unsolvable, or, as is said, that logic is undecidable.

In principle this section requires only the material of Chapters 3–4 and 9–10. In practice some facility at recognizing simple logical implications will be required: we are going to appeal freely to various facts about one sentence implying another, leaving the verification of these facts largely to the reader.

We begin by introducing simultaneously the language in which the sentences in $\Gamma$ and the sentence $D$ will be written, and its *standard interpretation* $\mathcal{M}$. The language interpretation will depend on what machine and what input $n$ we are considering. The domain of $\mathcal{M}$ will in all cases be the integers, positive and zero and negative. The nonnegative integers will be used to number the *times* when the machine is operating: the machine starts at time 0. The integers will also be used to number the squares on the tape: the machine starts at square 0, and the squares to the left and right are numbered as in Figure 11-1.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| … | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | … |

Figure 11-1. Numbering the squares of a Turing tape.

There will be a constant $\mathbf{0}$, whose denotation in the standard interpretation will be zero, and two-place predicates $\mathbf{S}$ and $\mathbf{<}$, whose denotations will be the successor relation (the relation an integer $n$ bears to $n + 1$ and nothing else) and the usual order relation, respectively. To save space, we write $\mathbf{S}uv$ rather than $\mathbf{S}(u, v)$, and similarly for other predicates. As to such other predicates, there will further be, for each of the (nonhalted) states of the machine, numbered let us say from 1 (the initial state) to $k$, a one-place predicate. In the standard interpretation, $\mathbf{Q}_i$ will denote the set of $t \geq 0$ such that at the time numbered $t$ the machine is in the state numbered $i$. Besides this we need two more two-place predicates **@** and $\mathbf{M}$. The denotation of the former will be the set of pairs of integers $t \geq 0$ and $x$ such that at the time number $t$, the machine is at the square numbered $x$. The denotation of the latter will be the set of $t \geq 0$ and $x$ such that at time $t$, square $x$ is 'marked', that is, contains a stroke rather than a blank. (We use $t$ as the variable when a time is intended, and $x$ and $y$ when squares are intended, as a reminder of the standard interpretation. Formally, the function of a variable is signalled by its position in the first or the second place of the predicate @ or $\mathbf{M}$.) It would be easy to adapt our construction to the case where more symbols than just the stroke and the blank are allowed, but for present purposes there is no reason to do so.

We must next describe the sentences that are to go into $\Gamma$ and the sentence $D$. The sentences in $\Gamma$ will fall into three groups. The first contains some 'background information' about $\mathbf{S}$ and $\mathbf{<}$ that would be the same for any machine and any input. The second consists of a single sentence specific to the input $n$ we are considering. The third consists of one sentence for each 'normal' instruction of the specific machine we are considering, that is, for each instruction except for those telling us to halt.

The 'background information' is provided by the following:

(1) $$\forall u \forall v \forall w(((\mathbf{S}uv \,\&\, \mathbf{S}uw) \to v = w) \,\&\, ((\mathbf{S}vu \,\&\, \mathbf{S}wu) \to v = w))$$

(2) $$\forall u \forall v(\mathbf{S}uv \to u < v) \,\&\, \forall u \forall v \forall w((u < v \,\&\, v < w) \to u < w)$$

(3) $$\forall u \forall v(u < v \to u \neq v).$$

These say that a number has only one successor and only one predecessor, that a number is less than its predecessor, and so on, and are all equally true in the standard interpretation.

It will be convenient to introduce abbreviations for the $m$th-successor relation, writing

$$\begin{array}{ll} \mathbf{S}_0uv & \text{for } u = v \\ \mathbf{S}_1uv & \text{for } \mathbf{S}uv \\ \mathbf{S}_2uv & \text{for } \exists y(\mathbf{S}uy \,\&\, \mathbf{S}yv) \\ \mathbf{S}_3uv & \text{for } \exists y_1 \exists y_2(\mathbf{S}uy_1 \,\&\, \mathbf{S}y_1y_2 \,\&\, \mathbf{S}y_2v) \end{array}$$

and so on. (In $\mathbf{S}_2$, $y$ may be any convenient variable distinct from $u$ and $v$; for definiteness let us say the first on our official list of variables. Similarly for $\mathbf{S}_3$.) The following are then true in the standard interpretation.

(4) $$\forall u \forall v \forall w(((\mathbf{S}_muv \,\&\, \mathbf{S}_muw) \to v = w) \,\&\, ((\mathbf{S}_mvu \,\&\, \mathbf{S}_mwu) \to v = w))$$

(5) $$\forall u \forall v(\mathbf{S}_muv \to u < v) \quad \text{if } m \neq 0$$

(6) $$\forall u \forall v(\mathbf{S}_muv \to u \neq v) \quad \text{if } m \neq 0$$

(7) $$\forall u \forall v \forall w((\mathbf{S}_mwu \,\&\, \mathbf{S}uv) \to \mathbf{S}_kwv) \quad \text{if } k = m + 1$$

(8) $$\forall u \forall v \forall w((\mathbf{S}_kwv \,\&\, \mathbf{S}uv) \to \mathbf{S}_mwu) \quad \text{if } m = k - 1.$$

Indeed, these are logical consequences of (1)–(3) and hence of $\Gamma$, true in *any* interpretation where $\Gamma$ is true: (4) follows on repeated application of (1); (5) follows on repeated application of (2); (6) follows from (3) and (5); (7) is immediate from the definitions; and (8) follows from (7) and (1). If we also write $\mathbf{S}_{-m}uv$ for $\mathbf{S}_mvu$, (4)–(8) still hold.

We need some further notational conventions before writing down the remaining sentences of $\Gamma$. Though officially our language contains only the numeral $\mathbf{0}$ and not numerals $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$, or $-\mathbf{1}$, $-\mathbf{2}$, $-\mathbf{3}$, it will be suggestive to write $y = \mathbf{1}$, $y = \mathbf{2}$, $y = -\mathbf{1}$, and the like for $\mathbf{S}_1(\mathbf{0}, y)$, $\mathbf{S}_2(\mathbf{0}, y)$, $\mathbf{S}_{-1}(\mathbf{0}, y)$, and so on, and to understand the application of a predicate to a numeral in the natural way, so that, for instance, $\mathbf{Q}_i\mathbf{2}$ and $\mathbf{S2}u$ abbreviate $\exists y(y = \mathbf{2} \,\&\, \mathrm{Q}_i y)$ and $\exists y(y = \mathbf{2} \,\&\, \mathbf{S}yu)$. A little thought shows that with these conventions (6)–(8) above (applied with $\mathbf{0}$ for $w$) give us the following wherein $\mathbf{p}$, $\mathbf{q}$, and so on, are the numerals for the numbers $p, q$, and so on:

(9) $$\mathbf{p} \neq \mathbf{q} \quad \text{if } p \neq q$$

(10) $$\forall v(\mathbf{Sm}v \to v = \mathbf{k}) \qquad \text{where} \quad k = m + 1$$

(11) $$\forall u(\mathbf{S}u\mathbf{k} \to u = \mathbf{m}) \qquad \text{where} \quad m = k - 1.$$

These abbreviatory conventions enable us to write down the remaining sentences of $\Gamma$ comparatively compactly.

The one member of $\Gamma$ pertaining to the input $n$ is a description of (the configuration at) time 0, as follows:

(12) $$\mathbf{Q}_0\mathbf{0} \ \& \ @\mathbf{00} \ \& \ \mathbf{M00} \ \& \ \mathbf{M01} \ \& \ \ldots \ \& \ \mathbf{M0n} \ \& \\ \forall x((x \neq \mathbf{0} \ \& \ x \neq \mathbf{1} \ \& \ \ldots \ \& \ x \neq \mathbf{n-1}) \rightarrow {\sim}\mathbf{M0}x).$$

This is true in the standard interpretation, since at time 0 the machine is in state 1, at square 0, with squares 0 through $n$ marked to represent the input $n$, and all other squares blank.

To complete the specification of $\Gamma$, there will be one sentence for each nonhalting instruction, that is, for each instruction of the following form, wherein $j$ is not the halted state:

(*) If you are in state $i$ and are scanning symbol $e$,
then —— and go into state $j$.

In writing down the corresponding sentence of $\Gamma$, we use one further notational convention, sometimes writing $\mathbf{M}$ as $\mathbf{M}_1$ and ${\sim}\mathbf{M}$ as $\mathbf{M}_0$. Thus $\mathbf{M}_e tx$ says, in the standard interpretation, that at time $t$, square $x$ contains symbol $e$ (where $e = 0$ means the blank, and $e = 1$ means the stroke). Then the sentence corresponding to (*) will have the form

(13) $$\forall t \forall x((\mathbf{Q}_i t \ \& @tx \ \& \ \mathbf{M}_e tx) \rightarrow \\ \exists u(\mathbf{S}tu \ \& \ \text{——} \ \& \ \mathrm{Q}_j u \ \& \\ \forall y((y \neq x \ \& \ \mathbf{M}_1 ty) \rightarrow \mathbf{M}_1 uy) \ \& \ \forall y((y \neq x \ \& \ \mathbf{M}_0 ty) \rightarrow \mathbf{M}_0 uy))).$$

The last two clauses just say that the marking of squares other than $x$ remains unchanged from one time $t$ to the next time $u$.

What goes into the space '——' in (13) depends on what goes into the corresponding space in (*). If the instruction is to (remain at the same square $x$ but) print the symbol $d$, the missing conjunct in (9) will be

$$@ux \ \& \ \mathbf{M}_d ux.$$

If the instruction is to move one square to the right or left (leaving the marking of the square $x$ as it was), it will instead be

$$\exists y(\mathbf{S}_{\pm 1} xy \ \& \ @uy \ \& \ (\mathbf{M}ux \leftrightarrow \mathbf{M}tx))$$

(with the minus sign for left and the plus sign for right). A little thought shows that when filled in after this fashion, (13) exactly corresponds to the instruction (*), and will be true in the standard interpretation.

This completes the specification of the set $\Gamma$. The next task is to describe the sentence $D$. To obtain $D$, consider a halting instruction, that is, an instruction of the type

(†) If you are in state $i$ and are scanning symbol $e$, then halt.

For each such instruction write down the sentence

(14) $$\exists t \exists x(\mathbf{Q}_i t \ \&\ @tx \ \&\ \mathbf{M}_e tx).$$

This will be true in the standard interpretation if and only if in the course of its operations the machine eventually comes to a configuration where the applicable instruction is (†), and halts for this reason. We let $D$ be the disjunction of all sentences of form (14) for all halting instructions (†). Since the machine will eventually halt if and only if it eventually comes to a configuration where the applicable instruction is some halting instruction or other, the machine will eventually halt if and only if $D$ is true in the standard interpretation.

We want to show that $\Gamma$ implies $D$ if and only if the given machine, started with the given input, eventually halts. The 'only if' part is easy. All sentences in $\Gamma$ are true in the standard interpretation, whereas $D$ is true only if the given machine started with the given input eventually halts. If the machine does not halt, we have an interpretation where all sentences in $\Gamma$ are true and $D$ isn't, so $\Gamma$ does not imply $D$.

For the 'if' part we need one more notion. If $a \geq 0$ is a time at which the machine has not (yet) halted, we mean by the *description of time* $a$ the sentence that does for $a$ what (12) does for 0, telling us what state the machine is in, where it is, and which squares are marked at time $a$. In other words, if at time $a$ the machine is in state $i$, at square $p$, and the marked squares are $q_1, q_2, \ldots, q_m$, then the description of time $a$ is the following sentence:

(15) $$\mathbf{Q}_i\mathbf{a} \ \&\ @\mathbf{ap} \ \&\ \mathbf{Maq}_1 \ \&\ \mathbf{Maq}_2 \ \&\ \ldots \ \&\ \mathbf{Maq}_m \ \&$$
$$\forall x((x \neq \mathbf{q}_1 \ \&\ x \neq \mathbf{q}_2 \ \&\ \ldots \ \&\ x \neq \mathbf{q}_m) \rightarrow \sim\mathbf{Ma}x).$$

It is important to note that (15) provides, directly or indirectly, the information whether the machine is scanning a blank or a stroke at time $a$. If the machine is scanning a stroke, then $p$ is one of the $q_r$ for $1 \leq r \leq m$, and $\mathbf{M}_1\mathbf{ap}$, which is to say **Map**, is actually a conjunct of (15). If the machine is scanning a blank, then $p$ is different from each of the various numbers $q$. In this case $\mathbf{M}_0\mathbf{ap}$, which is to say $\sim$**Map**, is implied by (15) and $\Gamma$. Briefly put, the reason is that (9) gives $\mathbf{p} \neq \mathbf{q}_r$ for each $q_r$, and then the last conjuct of (15) gives $\sim$**Map**.

[Less briefly but more accurately put, what the last conjunct of (15) *abbreviates* amounts to

$$\forall x((\sim\mathbf{S}_{q_1}\mathbf{0}x \ \&\ \ldots \sim\mathbf{S}_{q_m}\mathbf{0}x) \rightarrow \sim\exists t(\mathbf{S}_a\mathbf{0}t \ \&\ \mathbf{M}tx)).$$

What (9) applied to $p$ and $q_r$ *abbreviates* is

$$\sim\exists x((\mathbf{S}_p x \ \&\ \mathbf{S}_{q_r} x).$$

These together imply

$$\sim\exists t\exists x(\mathbf{S0}t \ \&\ \mathbf{S0}x \ \&\ \mathbf{M}tx)$$

which amounts to what $\sim$**Map** *abbreviates*.]

If the machine halts at time $b = a + 1$, that means that at time $a$ we had configuration for which the applicable instruction as to what to do next was a halting instruction of form (†). In that case, $\mathbf{Q}_i\mathbf{a}$ and **@ap** will be conjuncts of the description of time

$a$, and $\mathbf{M}_e\mathbf{ap}$ will be either a conjunct of the description also (if $e = 1$) or a logical implication of the description and $\Gamma$ (if $e = 0$). Hence (14) and therefore $D$ will be a logical implication of $\Gamma$ together with the description of time $a$. What if the machine does *not* halt at time $b = a + 1$?

**11.1 Lemma.** If $a \geq 0$, and $b = a + 1$ is a time at which the machine has not (yet) halted, then $\Gamma$ together with the description of time $a$ implies the description of time $b$.

*Proof*: The proof is slightly different for each of the four types of instructions (print a blank, print a stroke, move left, move right). We do the case of printing a stroke, and leave the other cases to the reader. Actually, this case subdivides into the unusual case where there is already a stroke on the scanned square, so that the instruction is just to change state, and the more usual case where the scanned square is blank. We consider only the latter subcase.

So the description of time $a$ looks like this:

$$(16) \qquad \begin{gathered}\mathbf{Q}_i\mathbf{a} \,\&\, @\mathbf{ap} \,\&\, \mathbf{Maq}_1 \,\&\, \mathbf{Maq}_2 \,\&\, \ldots \,\&\, \mathbf{Maq}_m \,\&\\ \forall x((x \neq \mathbf{q}_1 \,\&\, x \neq \mathbf{q}_2 \,\&\, \ldots \,\&\, x \neq \mathbf{q}_m) \rightarrow \sim\mathbf{Ma}x)\end{gathered}$$

where $p \neq q_r$ for any $r$, so $\Gamma$ implies $\mathbf{p} \neq \mathbf{q}_r$ by (9), and, by the argument given earlier, $\Gamma$ and (16) together imply $\sim\mathbf{Map}$. The sentence in $\Gamma$ corresponding to the applicable instruction looks like this:

$$(17) \qquad \begin{gathered}\forall t \forall x((\mathbf{Q}_i t \,\&\, @tx \,\&\, \sim\mathbf{M}tx) \rightarrow\\ \exists u(\mathbf{S}tu \,\&\, @ux \,\&\, \mathbf{M}ux \,\&\, \mathbf{Q}_j u \,\&\, \forall y((y \neq x \,\&\, \mathbf{M}ty) \rightarrow \mathbf{M}uy)\\ \&\, \forall y((y \neq x \,\&\, \sim\mathbf{M}ty) \rightarrow \sim\mathbf{M}uy))).\end{gathered}$$

The description of time $b$ looks like this:

$$(18) \qquad \begin{gathered}\mathbf{Q}_j\mathbf{b} \,\&\, @\mathbf{bp} \,\&\, \mathbf{Mbp} \,\&\, \mathbf{Mbq}_1 \,\&\, \mathbf{Mbq}_2 \,\&\, \ldots \,\&\, \mathbf{Mbq}_m \,\&\\ \forall x((x \neq \mathbf{p} \,\&\, x \neq \mathbf{q}_1 \,\&\, x \neq \mathbf{q}_2 \,\&\, \ldots \,\&\, x \neq \mathbf{q}_m) \rightarrow \sim\mathbf{Mb}x).\end{gathered}$$

And, we submit, (18) is a consequence of (16), (17), and $\Gamma$.

[Briefly put, the reason is this. Putting $\mathbf{a}$ for $t$ and $\mathbf{p}$ for $x$ in (17), we get

$$\begin{gathered}(\mathbf{Q}_i\mathbf{a} \,\&\, @\mathbf{ap} \,\&\, \sim\mathbf{Map}) \rightarrow\\ \exists u(\mathbf{Sa}u \,\&\, @u\mathbf{p} \,\&\, \mathbf{M}u\mathbf{p} \,\&\, \mathbf{Q}_j u \,\&\\ \forall y((y \neq \mathbf{p} \,\&\, \mathbf{Ma}y) \rightarrow \mathbf{M}uy) \,\&\, \forall y((y \neq \mathbf{p} \,\&\, \sim\mathbf{Ma}y) \rightarrow \sim\mathbf{M}uy)).\end{gathered}$$

Since (16) and $\Gamma$ imply $\mathbf{Q}_i\mathbf{a} \,\&\, @\mathbf{ap} \,\&\, \sim\mathbf{Map}$, we get

$$\begin{gathered}\exists u(\mathbf{Sau} \,\&\, @u\mathbf{p} \,\&\, \mathbf{M}u\mathbf{p} \,\&\, \mathbf{Q}_j u \,\&\\ \forall y((y \neq \mathbf{p} \,\&\, \mathbf{Ma}y) \rightarrow \mathbf{M}uy) \,\&\, \forall y((y \neq \mathbf{p} \,\&\, \sim\mathbf{Ma}y) \rightarrow \sim\mathbf{M}uy)).\end{gathered}$$

By (10), $\mathbf{Sa}u$ gives $u = \mathbf{b}$, where $b = a + 1$, and we get

$$\begin{gathered}@\mathbf{bp} \,\&\, \mathbf{Mbp} \,\&\, \mathbf{Q}_j\mathbf{b} \,\&\\ \forall y((y \neq \mathbf{p} \,\&\, \mathbf{Ma}y) \rightarrow \mathbf{Mb}y) \,\&\, \forall y((y \neq \mathbf{p} \,\&\, \sim\mathbf{M}ay) \rightarrow \sim\mathbf{Mb}y).\end{gathered}$$

The first three conjuncts of this last are the same, except for order, as the first three conjuncts of (18). The fourth conjunct, together with $\mathbf{p} \neq \mathbf{q}_k$ from (9) and the conjunct

$\mathbf{Maq}_k$ of (16), gives the conjunct $\mathbf{Mbq}_k$ of (18). Finally, the fifth conjunct together with the last conjunct of (16) gives the last conjunct of (18). The reader will see now what we meant when we said at the outset, 'Some facility at recognizing simple logical implications will be required.']

Now the description of time 0 is one of the sentences in $\Gamma$. By the foregoing lemma, if the machine does not stop at time 1, the description of time 1 will be a consequence of $\Gamma$, and if the machine then does not stop at time 2, the description of time 2 will be a consequence of $\Gamma$ together with the description of time 1 (or, as we can more simply say, since the description of time 1 is a consequence of $\Gamma$, the description of time 2 will be a consequence of $\Gamma$), and so on until the last time $a$ before the machine halts, if it ever does. If it does halt at time $a + 1$, we have seen that the description of time $a$, which we now know to be a consequence of $\Gamma$, implies $D$. Hence if the machine ever halts, $\Gamma$ implies $D$.

Hence we have established that if the decision problem for logical implication were solvable, the halting problem would be solvable, which (assuming Turing's thesis) we know it is not. Hence we have established the following result, assuming Turing's thesis.

**11.2 Theorem** (Church's theorem). The decision problem for logical implication is unsolvable.

## 11.2 Logic and Primitive Recursive Functions

By the *nullity problem* for a two-place primitive recursive function $f$ we mean the problem of devising an effective procedure that, given any $m$, would in a finite amount of time tell us whether or not there is an $n$ such that $f(m, n) = 0$. We are going to show how, given $f$, to write down a certain finite set of sentences $\Gamma$ and a certain formula $D(x)$ in a language that contains the constants $\mathbf{0}$ and the successor symbol $'$ from the language of arithmetic, and therefore contains the numerals $\mathbf{0}'$, $\mathbf{0}''$, $\mathbf{0}'''$, ... or $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$, ... as we usually write them. And then we are going to show that for any $m$, $\Gamma$ implies $D(\mathbf{m})$ if and only if there is an $n$ such that $f(m, n) = 0$. It follows that if the decision problem for logical implication could be solved, and an effective method devised to tell whether or not a given finite set of sentences $\Gamma$ implies a sentence $D$, then the nullity problem for any $f$ could be solved. Since it is known that, assuming Church's thesis, there is an $f$ for which the nullity problem is *not* solvable, it follows, again assuming Church's thesis, that the decision problem for logical implication is unsolvable, or, as is said, that logic is undecidable. The proof of the fact just cited about the unsolvability of the nullity problem requires the apparatus of Chapter 8, but for the reader who is willing to take this fact on faith, this section otherwise presupposes only the material of Chapters 6–7 and 9–10.

To begin the construction, the function $f$, being primitive recursive, is built up from the basic functions (successor, zero, the identity functions) by the two processes of composition and primitive recursion. We can therefore make a finite list of primitive recursive functions $f_0, f_1, f_2, \ldots, f_r$, such that for each $i$ from 1 to $r$, $f_i$ is either the zero function or the successor function or one of the identity functions, or is obtained