1
Enumerability

An *enumerable* set is one whose members can be enumerated: arranged in a single list with a first entry, a second entry, etc., so that every member of the set appears sooner or later in the list. Examples: the set P of positive integers is enumerated by the list

$$1, 2, 3, 4, \ldots$$

and the set E of even positive integers is enumerated by the list

$$2, 4, 6, 8, \ldots$$

Of course, the entries in these lists are not integers, but names of integers. In listing the members of a set (say, the members of the United States Senate) you manipulate names, not the things named. (You don't have the Senators form a queue; rather, you arrange their *names* in a list, perhaps alphabetically.)

By courtesy, we regard as enumerable the empty set, $\varnothing$, which has no members. (*The* empty set: there is only one. The terminology is a bit misleading: it suggests a comparison of empty sets with empty containers. But sets are more aptly compared with contents, and it should be considered that all empty containers have the same, null content.)

A list that enumerates a set may be finite or unending. An infinite set which is enumerable is said to be *enumerably infinite* or *denumerable.* Let us get clear about what things count as infinite lists, and what things do not. The positive integers can be arranged in a single infinite list as indicated above, but the following is not acceptable as a list of the positive integers:

$$1, 3, 5, 7, \ldots, 2, 4, 6, 8, \ldots$$

Here, all the odd positive integers are listed, and then all the even ones. This will not do. In an acceptable list, each item must appear sooner or later as the nth entry, for some *finite* n. But in the unacceptable arrangement above, none of the even positive integers are represented in this way. Rather, they appear (so to speak) as entries number $\infty + 1, \infty + 2$, etc.

To make this point perfectly clear we might define an enumeration of a set not as a listing, but as an arrangement in which each member of the set is *associated with* one of the positive integers 1, 2, 3, ... Actually, a list *is* such an arrangement. The thing named by the first entry in the list is associated with the positive integer 1, the thing named by the second entry is associated with the positive integer 2, and in general, the thing named by the nth entry is associated with the positive integer n.

In enumerating a set by listing its members, it is perfectly all right if a member of the set shows up more than once in the list. The requirement is rather that each member show up *at least once*. It does not matter if the list is redundant; all we require is that it be complete. Indeed, a redundant list can always be thinned out to get an irredundant list. (Examine each entry in turn, comparing it with the finite segment of the list that precedes it. Erase the entry if it already appears in that finite segment.)

In mathematical parlance, an infinite list determines a *function* (call it 'f') which takes positive integers as *arguments* and takes members of the set as *values*. The value of the function f for the argument n is denoted by '$f(n)$'. This value is simply the thing denoted by the nth entry in the list. Thus, the list

$$2, 4, 6, 8, \ldots$$

which enumerates the set E of even positive integers determines the function f for which we have

$$f(1) = 2, \quad f(2) = 4, \quad f(3) = 6, \quad f(4) = 8, \ldots$$

And conversely, the function f determines the list, except for notation. (The same list would look like this, in binary notation: 10, 100, 110, 1000, ...) Thus, we might have defined the function f first, by saying that for any positive integer n, the value of f is $f(n) = 2n$; and then we could have described the list by saying that for each positive integer n, its nth entry is the decimal representation of the number $f(n)$, i.e., of the number $2n$.

Then we may speak of sets as being enumerated by functions, as well as by lists. Instead of enumerating the odd positive integers by the list 1, 3, 5, 7, ..., we may enumerate them by the function which assigns to each positive integer n the value $2n-1$. And instead of enumerating the set P of all positive integers by the list 1, 2, 3, 4, ..., we may enumerate P by the function which assigns to each positive integer n the value n itself. (This is the *identity function*. If we call it 'i', we have $i(n) = n$ for each positive integer n.)

If one function enumerates a nonempty set, so does some other; and so, in fact, do infinitely many others. Thus, the set of positive integers

is enumerated not only by the function i, but also by the function (call it 'g') which is determined by the following list:

$$2, 1, 4, 3, 6, 5, \ldots$$

This list is obtained from the list 1, 2, 3, 4, 5, 6, ... by interchanging entries in pairs: 1 with 2, 3 with 4, 5 with 6, and so on. This list is a strange but perfectly acceptable enumeration of the set P: every positive integer shows up in it, sooner or later. The corresponding function, g, can be defined as follows:

$$g(n) = \begin{cases} n+1 & \text{if } n \text{ is odd,} \\ n-1 & \text{if } n \text{ is even.} \end{cases}$$

This definition is not as neat as the definitions $f(n) = 2n$ and $i(n) = n$ of the functions f and i, but it does the job: it does indeed associate one and only one member of P with each positive integer n. And the function g so defined does indeed enumerate P: for each member m of P there is a positive integer n for which we have $g(n) = m$.

We have noticed that it is perfectly all right if a list that enumerates a set is redundant, for if we wished, we could always thin it out so as to remove repetitions. It is also perfectly all right if a list has gaps in it, since one could go through and close up the gaps in turn. Thus, a flawless enumeration of the positive integers is given by the following gappy list:

$$1, -, 2, -, 3, -, 4, -, 5, -, 6, -, \ldots$$

The corresponding function (call it 'h') assigns values corresponding to the first, third, fifth, ... entries, but assigns no values corresponding to the gaps (second, fourth, sixth, ... entries). Thus we have $h(1) = 1$, but $h(2)$ is nothing at all, for the function h is undefined for the argument 2; $h(3) = 2$, but $h(4)$ is undefined; $h(5) = 3$, but $h(6)$ is undefined. And so on: h is a *partial function* of positive integers, i.e., it is defined only for positive integer arguments, but not for all such arguments. Explicitly, we might define the partial function h as follows:

$$h(n) = (n+1)/2 \quad \text{if } n \text{ is odd.}$$

Or, to make it clear that we haven't simply forgotten to say what values h assigns to even positive integers, we might put the definition as follows:

$$h(n) = \begin{cases} (n+1)/2 & \text{if } n \text{ is odd,} \\ \text{undefined otherwise.} \end{cases}$$

Now the partial function h is a queer but perfectly acceptable enumeration of the set P of positive integers.

It would be perverse to choose h instead of the simple function i as an enumeration of P; but other sets are most naturally enumerated by partial functions. Thus, the set E of even positive integers is conveniently enumerated by the partial function (say, 'j') that agrees with i for even arguments, and is undefined for odd arguments:

$$j(n) = \begin{cases} n & \text{if } n \text{ is even,} \\ \text{undefined otherwise.} \end{cases}$$

The corresponding gappy list (in decimal notation) is

$$-, 2, -, 4, -, 6, -, 8, \ldots$$

Of course, the function f, defined by

$$f(n) = 2n$$

for all positive integers n, is an equally acceptable enumeration of E, corresponding to the gapless list,

$$2, 4, 6, 8, \ldots$$

Any set S of positive integers is enumerated quite simply by a partial function, s, which is defined as follows:

$$s(n) = \begin{cases} n & \text{if } n \text{ is in the set } S, \\ \text{undefined otherwise.} \end{cases}$$

Thus, the set $\{1, 2, 5\}$ which consists of the numbers 1, 2, and 5 is enumerated by the partial function k which is defined

$$k(n) = \begin{cases} n & \text{if } n = 1 \quad \text{or} \quad n = 2 \quad \text{or} \quad n = 5, \\ \text{undefined otherwise.} \end{cases}$$

We shall see in the next chapter that although every set *of positive integers* is enumerable, there are sets of other sorts which are not enumerable. To say that a set A is enumerable is to say that there is a function all of whose arguments are positive integers and all of whose values are members of A, and that each member of A is a value of this function: for each member a of A there is at least one positive integer n to which the function assigns a as its value. Notice that nothing in this definition requires A to be a set of positive integers. Instead, A might be a set of people (members of the United States Senate, perhaps); it might be a set of strings of symbols (perhaps, the set of all grammatically correct English sentences, where we count the space between adjacent words as a symbol); or the members of A might themselves be sets, as when A is the set $\{P, E, \varnothing\}$. Here, A is a set with three members, each of which is itself a set. One member of A is the infinite set P of all positive integers; another member of A is the infinite set E of all even positive integers; and the third is the empty set $\varnothing$. The set A is certainly enumerable, e.g. by the following finite list:

$$P, \ E, \ \varnothing.$$

Each entry in this list names a member of A, and every member of A is named sooner or later in the list. The list determines a function – call it 'f' – which can be defined by the three statements

$$f(1) = P, \quad f(2) = E, \quad f(3) = \varnothing.$$

To be precise, f is a *partial function* of positive integers, being undefined for arguments greater than 3. Moral: a (partial or total) function of positive integers may have its values in any set whatever. Its arguments must all be positive integers, but its values need not be positive integers.

The set of all finite strings of letters of the alphabet provides an example of an enumerably infinite set which is not a set of positive integers. This set is enumerable because its members can be arranged in a single list, which we shall describe but not exhibit. The first 26 entries in the list are the 26 letters of the alphabet, in their usual order. The next 676 ($= 26^2$) entries are the two-letter sequences, in dictionary order. The following 17576 ($= 26^3$) entries are the three-letter strings, again in dictionary order. And so on, without end. From this description, with the aid of paper, pencil, and patience, you can determine the position in the list of any finite string that interests you. Then in effect we have defined a function of positive integers which enumerates the set in question. The definition does not have the form of a mathematical equation, but it is a perfectly acceptable definition for all that: it does its job, which is to associate a member of the set, as value, with each positive integer, as argument. No matter that the definition was given in English instead of mathematical notation, as long as it is clear and unambiguous.

Since the members of the set of all finite strings of letters are inscriptions, we can plausibly speak of arranging those members themselves in a list. But we might also speak of the entries in the list as *names of themselves* so as to be able to continue to insist that in enumerating a set, it is *names* of members of the set that are arranged in a list.

In conclusion, let us straighten out our terminology.

A *function* is an assignment of *values* to *arguments*. The set of all those arguments to which the function assigns values is called the *domain* of the function. The set of all those values which the function assigns to its arguments is called the *range* of the function. In the case of functions whose arguments are positive integers, we distinguish between *total* functions and *partial* functions. A total function of positive integers is one whose domain is the whole set P of positive integers. A partial function of positive integers is one whose domain is something less than the whole set P. From now on, when we speak simply of a *function of positive integers,* we should be understood as leaving it open, whether the function is total or partial. This is a departure from the usual terminology, in which 'function' (of positive integers) always means *total function.* Also, from now on we shall regard as a genuine partial function of positive integers the weird function e whose domain is the empty set $\varnothing$. *This function is undefined for all arguments!* Now a set is *enumerable* if and only if it is *the range of some function of positive integers.* The empty set is enumerable because it is the range of e.

Example. The set of positive rational numbers is enumerable. A positive rational number is a number that can be expressed as a ratio of positive integers: in the form m/n where m and n are positive integers. As a preliminary to enumerating them, we organize them in a rectangular array as in Table 1-1(a).

Table 1-1

(a)

$\frac{1}{1}$	$\frac{2}{1}$	$\frac{3}{1}$	$\frac{4}{1}$	$\frac{5}{1}$	…
$\frac{1}{2}$	$\frac{2}{2}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	…
$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$	$\frac{4}{3}$	$\frac{5}{3}$	…
$\frac{1}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\frac{4}{4}$	$\frac{5}{4}$	…
$\frac{1}{5}$	$\frac{2}{5}$	$\frac{3}{5}$	$\frac{4}{5}$	$\frac{5}{5}$	…
⋮	⋮	⋮	⋮	⋮	

(b)

1	2	5	10	17	…
4	3	6	11	18	…
9	8	7	12	19	…
16	15	14	13	20	…
25	24	23	22	21	…
⋮	⋮	⋮	⋮	⋮	

Every positive rational number is represented somewhere in this array. Thus, the number m/n is represented in the mth column, nth row. In fact, every such number is represented infinitely often, for m/n, $2m/2n$, $3m/3n$, … all represent the same number. In particular, the number 1 is represented by every entry in the main diagonal of Table 1-1(a): first row, first column; second row, second column; etc.

To show that the positive rationals are enumerable, we must arrange Table 1-1(a) in a single list. This can be done in various ways, e.g. in the pattern shown in Table 1-1(b), where the number at each position gives the position in the list of the corresponding entry in Table 1-1(a). Then the list, in which each positive rational is represented infinitely often, is this:

$$\tfrac{1}{1},\quad \tfrac{2}{1},\quad \tfrac{2}{2},\quad \tfrac{1}{2},\quad \tfrac{3}{1},\ldots$$

The two arrays in Table 1-1 also give a perfectly satisfactory definition of a function of positive integers (call it 'r') which enumerates the positive rationals. Thus we have

$$r(1) = \tfrac{1}{1},\quad r(2) = \tfrac{2}{1},\quad r(3) = \tfrac{2}{2},\quad r(4) = \tfrac{1}{2},\quad r(5) = \tfrac{3}{1},\ldots$$

or, simplifying,

$$r(1) = 1,\quad r(2) = 2,\quad r(3) = 1,\quad r(4) = \tfrac{1}{2},\quad r(5) = 3,\ldots$$

The function r is quite adequately defined by the two arrays: given any positive integer n, it is perfectly straightforward to find the value $r(n)$ by building the two arrays up to the point where the number n is reached in the second. The value $r(n)$ will then be given by the corresponding entry in the first array. It is amusing, but by no means necessary, to give a more mathematical-looking definition of r. If you want to try it, here is a hint: every positive integer, n, is somewhere in the interval from one perfect square to the next, i.e., for each n there is exactly one m for which we have $(m-1)^2 < n \leqslant m^2$. Let us use the symbol 'sq' for the function which assigns the value m to the argument n as above. Your problem is to replace the question marks in the following definition by mathematical expressions involving n and the function sq:

$$r(n) = \begin{cases} \mathrm{sq}(n)/? & \text{if } ? \\ ?/\mathrm{sq}(n) & \text{otherwise.} \end{cases}$$

But if this sort of thing paralyzes you, don't bother with it. If you do want to try it, you can check your answer against the solution to Exercise 1.1 below.

Exercises. (Solutions follow)

1.1 Define the function r in the form shown above, where $\mathrm{sq}(n)$ is the smallest number whose square is n or greater. *Hint*: begin by writing out the first few entries in the mth rows and mth columns of Table 1-1.

1.2 Show that the integers …, -2, -1, 0, 1, 2, … are enumerable, by arranging them in a single infinite list with a beginning but no end; and write out a mathematical-looking definition of the function which corresponds to your list.

1.3 Show that the set of all the rationals (positive, negative, and zero) is enumerable, by describing a single infinite list (with a beginning but no end) which enumerates them.

1.4 Show that the set of all *finite* strings of pluses and minuses is enumerable, by describing a single list that enumerates them.

1.5 Then show that the set of all *finite* sets of positive integers is enumerable, by describing a way in which finite strings of pluses and minuses can be used as names of finite sets of positive integers.

1.6 Let D be a certain set of sets of positive integers, as follows: a set of positive integers belongs to D if and only if it is definable by a finite number of words in English. Thus, the following sets belong to D because they are defined here by finite numbers of words in English.

E:	the set of all even positive integers (7 words).
$\varnothing$:	the set which has no members at all (8 words).
$\{1, 2\}$:	the set whose only members are the numbers one and two (11 words).
$\{1, \ldots, 999\}$:	the set of all positive integers less than one thousand (10 words).

And so on. *Show that the set D is enumerably infinite.*

Solutions

1.1 Bearing in mind that $m = \mathrm{sq}(n)$, the mth rows and columns of Table 1-1 are as follows:

(*a*)
$$\begin{array}{lllll}
 & & & \mathrm{sq}(n)/1 & \\
 & & & \mathrm{sq}(n)/2 & \\
 & & & \mathrm{sq}(n)/3 & \\
 & & & \vdots & \\
1/\mathrm{sq}(n) & 2/\mathrm{sq}(n) & 3/\mathrm{sq}(n) \ldots & \mathrm{sq}(n)/\mathrm{sq}(n) & \ldots \\
 & & & \vdots &
\end{array}$$

(*b*)
$$\begin{array}{lllll}
 & & & (\mathrm{sq}(n)-1)^2+1 & \\
 & & & (\mathrm{sq}(n)-1)^2+2 & \\
 & & & (\mathrm{sq}(n)-1)^2+3 & \\
 & & & \vdots & \\
(\mathrm{sq}(n))^2-0 & (\mathrm{sq}(n))^2-1 & (\mathrm{sq}(n))^2-2 \ldots & (\mathrm{sq}(n)-1)^2+m & \ldots \\
 & & & \vdots &
\end{array}$$

The required definition then emerges as

$$r(n) = \begin{cases} \dfrac{\mathrm{sq}(n)}{n-(\mathrm{sq}(n)-1)^2} & \text{if} \quad n \leqslant (\mathrm{sq}(n)-1)^2+\mathrm{sq}(n), \\[2ex] \dfrac{(\mathrm{sq}(n))^2+1-n}{\mathrm{sq}(n)} & \text{otherwise.} \end{cases}$$

This is cute, but no more illuminating than the definition of r in terms of the arrays in Table 1-1.

1.2 The simplest list is: $0, 1, -1, 2, -2, 3, -3, \ldots$. Then if the corresponding function is called 'f' we have $f(1) = 0$, $f(2) = 1$, $f(3) = -1$, $f(4) = 2$, etc. Here is a mathematical-looking definition of f:

$$f(n) = \begin{cases} -\dfrac{n-1}{2} & \text{if } n \text{ is odd,} \\[2ex] \dfrac{n}{2} & \text{if } n \text{ is even.} \end{cases}$$

1.3 You already know how to arrange the positive rationals in a single infinite list. Write '0' in front of this list and then write the positive rationals, backwards, with minus signs before them, in front of that. You now have this:

$$\ldots, -3, -\tfrac{1}{2}, -1, -2, -1, 0, 1, 2, 1, \tfrac{1}{2}, 3, \ldots$$

Finally, use the method of Exercise 1.2 to turn this into a proper list:

$$0, 1, -1, 2, -2, 1, -1, \tfrac{1}{2}, -\tfrac{1}{2}, 3, -3, \ldots$$

1.4 Here is an enumeration:

$$+, -, ++, +-, -+, --, +++, ++-, +-+,$$
$$+--, -++, \ldots$$

Description of the list: first list the strings of length 1, then the strings of length 2, then the strings of length 3, and so on; within each group, arrange the strings in dictionary order, thinking of '+' as coming before '−' in the alphabet. (Thus, the next six entries are $-+-$, $--+$, $---$, $++++$, $+++-$, $++-+$.)

1.5 Interpret a finite string as a name of the set to which the number n belongs if there is an nth symbol in the string and it is a plus; and to which the number n does not belong if there are fewer than n symbols in the string, or if there is an nth symbol but it is a minus. Every finite set of positive integers now has a name. In fact, it has infinitely many names, e.g. the empty set $\varnothing$ is named by '−' and by '−−' and by '−−−' etc., and the set $\{1, 2, 5\}$ is named by '++−−+' and by '++−−+−' and by '++−−+−−' etc.

1.6 Certainly the set D is infinite, for among its members are all of the sets in the following infinite collection:

$\{1\}$:	the set whose only member is the number one
$\{2\}$:	the set whose only member is the number two
$\vdots$	

Since every positive integer can be named in English, this subset of D is infinite, and therefore D itself is infinite. The fact that D is enumerable follows from the fact that every definition in English of a set of positive integers is a finite string of symbols, each of which is either one of the 26 letters of the alphabet (never mind about capitals or punctuation) or a space (to separate adjacent words), which we can treat as a 27th letter of the alphabet. (If you want to use capitals and punctuation, just add the appropriate symbols to this alphabet. You will still have a finite but large alphabet.) Now the finite strings of signs in the enlarged alphabet can be enumerated: first list the strings of length 1, then the strings of length 2, and so on; and within each group, arrange the strings in dictionary order (after arbitrarily deciding on some dictionary order for the enlarged alphabet). In this enumeration, most of the entries will be gibberish, or will make sense but will fail to be definitions of sets of positive integers. Erase all such entries. This leaves a gappy list which enumerates all the English definitions of sets of positive integers.

2 Diagonalization

Not all sets are enumerable: some are too big. An example is the set of *all sets of positive integers*. This set (call it 'P^*') contains, as a member, each finite and each infinite set of positive integers: the empty set $\varnothing$, the set P of all positive integers, and every set between these two extremes.

To show (following Georg Cantor) that P^* is not enumerable, we give a method which can be applied to *any* list L of sets of positive integers in order to discover a set $\bar{D}(L)$ of positive integers which is not named in the list. If you then try to repair the defect by adding $\bar{D}(L)$ to the list as a new first member, the same method, applied to the augmented list L', will yield a different set $\bar{D}(L')$ which is not in the augmented list!

The method is this. Confronted with any infinite list

$$S_1, S_2, S_3, \ldots \qquad (L)$$

of sets of positive integers, we define a set $\bar{D}(L)$ as follows:

For each positive integer n, n is in $\bar{D}(L)$ if and only if n is *not* in S_n. (*)

As the notation '$\bar{D}(L)$' indicates, the composition of the set $\bar{D}(L)$ depends on the composition of the list L so that different lists L may yield different sets $\bar{D}(L)$. It should be clear that the statement (*) genuinely defines a set $\bar{D}(L)$ for, given any positive integer, n, we can tell whether n is in $\bar{D}(L)$ if we can tell whether n is in the nth set in list L. Thus, if S_3 happens to be the set E of all even positive integers, the number 3 is not in S_3 and therefore it *is* in $\bar{D}(L)$.

To show that the set $\bar{D}(L)$ which this method yields is never in the given list L, we argue by *reductio ad absurdum*: we suppose that $\bar{D}(L)$ does appear somewhere in list L, say as entry number m, and deduce a contradiction, thus showing that the supposition must be false. Here we go. *Supposition*: For some positive integer m,

$$S_m = \bar{D}(L).$$

(Thus, if 127 is such an m, we are supposing that '$\bar{D}(L)$' and 'S_{127}' are two names of one and the same set: we are supposing that a positive integer belongs to $\bar{D}(L)$ if and only if it belongs to the 127th set in list L.) To deduce a contradiction from this supposition we apply definition (*) to the particular positive integer m: with $n = m$, (*) tells us that

$$m \text{ is in } \bar{D}(L) \text{ if and only if } m \text{ is } \textit{not} \text{ in } S_m.$$

Now a contradiction follows from our supposition: if S_m and $\bar{D}(L)$ are one and the same set we have

$$m \text{ is in } \bar{D}(L) \text{ if and only if } m \text{ is } \textit{not} \text{ in } \bar{D}(L).$$

Since this is a flat self-contradiction, our supposition must be false. For no positive integer m do we have $S_m = \bar{D}(L)$. In other words, the set $\bar{D}(L)$ is named nowhere in list L.

So the method works. Applied to any list of sets of positive integers it yields a set of positive integers which was not in the list. Then no list enumerates all sets of positive integers: the set P^* of all such sets is not enumerable.

The method ('diagonalization') is so important that it will be well to look at it again from a slightly different point of view, which allows the entries in list L to be more readily visualized. Accordingly, we think of the sets $S_1, S_2, \ldots$ as represented by functions $s_1, s_2, \ldots$ of positive integers which take the numbers 0 and 1 as values. The relationship between the set S_n and the corresponding function s_n is simply this: for each positive integer p we have

$$s_n(p) = \begin{cases} 1 & \text{if } p \text{ is in } S_n, \\ 0 & \text{if } p \text{ is not in } S_n. \end{cases}$$

Then the list L can be visualized as an infinite rectangular array of zeros and ones as in Table 2-1.

TABLE 2-1

	1	2	3	4	...
s_1	$s_1(1)$	$s_1(2)$	$s_1(3)$	$s_1(4)$	...
s_2	$s_2(1)$	$s_2(2)$	$s_2(3)$	$s_2(4)$	...
s_3	$s_3(1)$	$s_3(2)$	$s_3(3)$	$s_3(4)$	...
s_4	$s_4(1)$	$s_4(2)$	$s_4(3)$	$s_4(4)$	...
⋮	⋮	⋮	⋮	⋮	

The nth row of this array represents the function s_n and thus represents the set S_n: that row

$$s_n(1)\, s_n(2)\, s_n(3)\, s_n(4) \ldots$$

is a sequence of zeros and ones in which the pth entry, $s_n(p)$, is 1 or 0 accordingly as the number p is or is not in the set S_n. The entries in the diagonal of the array (upper left to lower right) form a sequence of zeros and ones:

$$s_1(1)\, s_2(2)\, s_3(3)\, s_4(4) \ldots$$

This sequence of zeros and ones ('the diagonal sequence') determines a set of positive integers ('the diagonal set') *which may well be among those listed in L*. In other words, there may well be a positive integer d such that the set S_d is none other than our diagonal set. The sequence of zeros and ones in the dth row of Table 2-1 would then agree with the diagonal sequence, entry by entry:

$$s_d(1) = s_1(1), \quad s_d(2) = s_2(2), \quad s_d(3) = s_3(3), \ldots$$

That is as may be: the diagonal set may or may not appear in the list L, depending on the detailed makeup of that list. What we want is a set we can rely upon *not* to appear in L, no matter how L is composed. Such a set lies near to hand: it is *the antidiagonal set* which consists of the positive integers not in the diagonal set. The corresponding *antidiagonal sequence* is obtained by changing zeros to ones and ones to zeros in the diagonal sequence. We may think of this transformation as a matter of subtracting each member of the diagonal sequence from 1: we write the antidiagonal sequence as

$$1 - s_1(1), \quad 1 - s_2(2), \quad 1 - s_3(3), \quad 1 - s_4(4), \ldots$$

This sequence can be relied upon not to appear as a row in Table 2-1 for if it did so appear – say, as the mth row – we should have

$$s_m(1) = 1 - s_1(1), \quad s_m(2) = 1 - s_2(2), \quad \ldots \quad s_m(m) = 1 - s_m(m), \ldots$$

But the mth of these equations cannot hold. (*Proof.* $s_m(m)$ must be zero or one. If zero, the mth equation says that $0 = 1$. If one, the mth equation says that $1 = 0$.) Then the antidiagonal sequence differs from every row of our array, and so the antidiagonal set differs from every set in our list L. This is no news, for the antidiagonal set is simply the set $\bar{D}(L)$. We have merely repeated with the aid of a diagram – Table 2-1 – our proof that $\bar{D}(L)$ appears nowhere in the list L.

Of course, it is rather queer to say that the members of an infinite set 'can be arranged' in a single infinite list. By whom? Certainly not by any human being, for nobody has that much time or paper; and similar restrictions apply to machines. In fact, to call a set enumerable is simply to say that it is the range of some function of positive integers. Thus, the set E of even positive integers is enumerable because there are functions of positive integers which have E as their range, e.g. the function f for which we have $f(n) = 2n$ for each positive integer n. Any such function can then be thought of as a program which a superhuman enumerator can follow in order to arrange the members of the set in a single list. More explicitly, the program (the set of instructions) is this:

> Start counting from 1, and never stop. As you reach each number n, write a name of $f(n)$ in your list. (Where $f(n)$ is undefined, leave the nth position blank.)

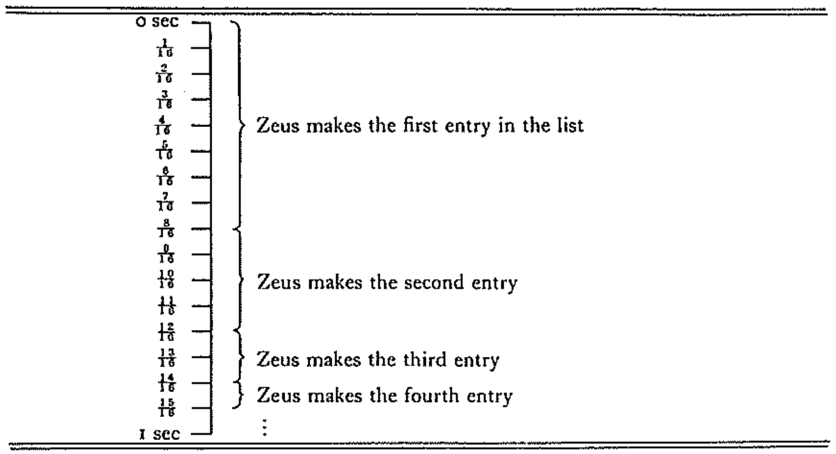
But there is no need to refer to the list, or to a superhuman enumerator: anything we need to say about enumerability can be said in terms of the functions themselves, e.g. to say that the set P^* is not enumerable is simply to deny the existence of any function of positive integers which has P^* as its range.

Vivid talk of lists and superhuman enumerators may still aid the imagination, but in such terms the theory of enumerability and diagonalization appears as a chapter in mathematical theology. To avoid treading on living toes we might put the whole thing in a classical Greek setting: Cantor proved that there are sets which even Zeus cannot enumerate, no matter how fast he works, or how long (even, infinitely long).

If a set *is* enumerable, Zeus can enumerate it in one second by writing out an infinite list faster and faster. He spends $\frac{1}{2}$ second writing the first entry in the list; $\frac{1}{4}$ second writing the second entry; $\frac{1}{8}$ second writing the third; and in general, he writes each entry in half the time he spent on its predecessor. At no point *during* the one second interval has he written out the whole list, but when one second has passed, the list is complete! On a time scale in which the marked divisions are sixteenths of seconds, the process can be represented as in Table 2-2.

To speak of writing out an infinite list (e.g. of all the positive integers, in decimal notation) is to speak of such an enumerator either working faster and faster as above, or taking all of infinite time to complete the list (making one entry per second, perhaps). Indeed, Zeus could write out an infinite sequence of infinite lists if he chose to, taking only one second to complete the job. He could simply allocate the first half second to the business of writing out the first infinite list ($\frac{1}{4}$ second for the first entry, $\frac{1}{8}$ second for the next, and so on); he could then write out the whole second list in the following quarter second ($\frac{1}{8}$ second for the first entry, $\frac{1}{16}$ second

TABLE 2-2



for the next, and so on); and in general, he could write out each subsequent list in just half the time he spent on its predecessor, so that after one second had passed he would have written out every entry in every list, in order. But the result does not count as a single infinite list, in our sense of the term. In our sort of list, each entry must come some *finite* number of places after the first.

As we use the term 'list', Zeus has not produced a list by writing infinitely many infinite lists one after another. But he could perfectly well produce a genuine list which exhausts the entries in all the lists, by using some such device as we used in Chapter 1 to enumerate the positive rational numbers. He need only imagine an infinite rectangular array in which, for each positive integer n, the nth row is the nth of his lists. He can then pick out various entries from various lists in either of the orders indicated in Table 2-3 so as to get a single list in which every entry in each of the original lists appears at one or another *finite* number of places from the beginning.

TABLE 2-3

(*a*)

1	2	5	…
4	3	6	…
9	8	7	…
⋮	⋮	⋮	

(*b*)

1	2	4	7	…
3	5	8	·	
6	9	·		
10	·			
⋮				

But Cantor's *diagonal argument* shows that neither this nor any more ingenious device is available, even to a god, for arranging all the sets of positive integers into a single infinite list. Such a list would be as much an impossibility as a round square: the impossibility of enumerating all the sets of positive integers is as absolute as the impossibility of drawing a round square, even for Zeus.

Exercises

2.1 Given a list L of sets $S_1, S_2, \ldots$, define a set $D(L)$ as follows:

For each positive integer n, n is in $D(L)$ if and only if n is in S_n.

Suppose that L is the enumeration of the finite sets of positive integers that was given in the solution to Exercise 1.5. (*a*) What is $D(L)$? (*b*) Is $D(L)$ in L? (*c*) What is $\overline{D}(L)$?

2.2 Adapt the diagonal argument (see Table 2-1) so as to prove that the set of real numbers in the interval from 0 to 1 is not enumerable. *Hint*: each such number is represented in one or two ways in decimal notation by a decimal point followed by an infinite string of digits, e.g. $0.5 = 0.5000\ldots = 0.4999\ldots$ and $0.123 = 0.123000\ldots = 0.122999\ldots$, but if no zeros or nines appear in it, the decimal representation (as an *infinite* string of digits) is unique.

2.3 Prove by a diagonal argument that the set of all total and partial functions of positive integers that take only positive integers as values is not enumerable.

2.4 Define a *word* as any finite string of letters from a certain enumerably infinite alphabet $a_1, a_2, \ldots$. Prove that each of the following sets is enumerable:

(*a*) the two-letter words,
(*b*) the three-letter words,
(*c*) the n-letter words, for arbitrary but fixed n,
(*d*) all words, irrespective of (*finite*) length.

Solutions

2.1 The successive entries in L are $S_1 = \{1\}$, $S_2 = \varnothing$, $S_3 = \{1, 2\}$, $S_4 = \{1\}$, etc. Only for $n = 1$ does n belong to S_n, for the subscripts n grow much faster than the largest members of S_n (which are never greater than the lengths of the corresponding strings of pluses and minuses). Then

(*a*) $D(L) = \{1\}$.
(*b*) $D(L)$ is in L, e.g. $D(L) = S_1$ and $D(L) = S_4$.
(*c*) $\overline{D}(L) = \{2, 3, 4, \ldots\}$ which is infinite and hence not in L – as we knew anyway, by the diagonal argument.

2.2 Let L be any list in which each entry is a decimal point followed by an unending string of decimal digits. In particular, suppose that the successive digits in the nth entry are $s_n(1), s_n(2), \ldots$, so that the entries in L are the rows of Table 2-1 (where, now, each position in the table can represent any one of the ten digits '0', …, '9'). Where d is any one of the ten digits, define

$$d' = \begin{cases} 1 & \text{if } d = 2, \\ 2 & \text{otherwise.} \end{cases}$$

(The essential feature of this definition is that $0 \neq d' \neq d$.) Now let the successive digits of $\overline{D}(L)$ be $s_1(1)', s_2(2)', \ldots$. These are not the same as the successive digits of any row of Table 2-1 – say, the nth – for if they were, we should have $s_n(n)' = s_n(n)$, which is impossible by definition of the operation $'$. Then since $\overline{D}(L)$ contains no zeros or nines, the number which it represents is represented by no entry in the list L. Thus, no list L of real numbers in the unit interval can be exhaustive.

2.3 Let $L = f_1, f_2, \ldots$ be an enumeration of such functions. Define the antidiagonal function u ($= \overline{D}(L)$) as follows:

$$u(n) = \begin{cases} 1 & \text{if } f_n(n) \text{ is undefined,} \\ f_n(n) + 1 & \text{otherwise.} \end{cases}$$

Then u is a well defined total function of positive integers which takes only positive integers as values. But u cannot be in L: if we had $u = f_m$ (say) the definition would yield

$$f_m(m) = u(m) = \begin{cases} 1 & \text{if } f_m(m) \text{ is undefined,} \\ f_m(m) + 1 & \text{otherwise.} \end{cases}$$

Then whether or not f_m assigns a value to the argument m, we have a contradiction: if $f_m(m)$ is undefined it *is* defined (and equal to 1), while if $f_m(m)$ is defined and equal to n (say) we have $n = n + 1$.

2.4 The straightforward solution uses the same trick (again and again) that was used in enumerating the positive rational numbers.

(*a*) Form a rectangular array of all two-letter words as in Table 2-4(*a*) and weave your way through them, e.g. in the triangular pattern of Table 2-3(*b*) to get the single list

$$a_1a_1,\ a_1a_2,\ a_2a_1,\ a_1a_3,\ a_2a_2,\ a_3a_1, \ldots$$

TABLE 2-4

(*a*)

	a_1	a_2	a_3	...
a_1	a_1a_1	a_1a_2	a_1a_3	...
a_2	a_2a_1	a_2a_2	a_2a_3	...
a_3	a_3a_1	a_3a_2	a_3a_3	...
⋮	⋮	⋮	⋮	

(*b*)

	a_1a_1	a_1a_2	a_2a_1	a_1a_3	...
a_1	$a_1a_1a_1$	$a_1a_1a_2$	$a_1a_2a_1$	$a_1a_1a_3$	...
a_2	$a_2a_1a_1$	$a_2a_1a_2$	$a_2a_2a_1$	$a_2a_1a_3$	...
a_3	$a_3a_1a_1$	$a_3a_1a_2$	$a_3a_2a_1$	$a_3a_1a_3$	...
⋮	⋮	⋮	⋮	⋮	

(*b*) To get the three-letter words, notice that each of these is obtainable by writing a two-letter word after a one-letter word. Form the array of Table 2-4(*b*) and weave in the triangular pattern of Table 2-3(*b*) to get the list

$$a_1a_1a_1,\ a_1a_1a_2,\ a_2a_1a_1,\ a_1a_2a_1,\ a_2a_1a_2, \ldots$$

(*c*) To list the four-letter words, weave through an array which has the three-letter words on top and the one-letter words down the side. And so on. By iterating the process sufficiently often, you can list the n-letter words for any particular n.

(*d*) To list all words of whatever finite length, weave through an array in which the first row is the list of one-letter words $a_1, a_2, \ldots$ the second row is the list of two-letter words (*a*), the third row is the list of three-letter words (*b*), etc.

But there is a simpler, direct method for obtaining a gappy list of all words of whatever finite length on the alphabet $a_1, a_2, \ldots$. To determine the position in the list of any such word, replace each occurrence of a_1 in it by '12', replace each occurrence of a_2 in it by '122', and in general replace each occurrence of a_n in it by a *one* followed by n *twos*. The result is a string of digits which, read in the decimal notation, gives the position of the word in the list. (The list then starts with eleven gaps. The first entry is the one-letter word a_1, in position twelve. The next entry is the word a_2, in position 122. The third entry is the word a_1a_1, in position 1212. The fourth is a_3, in position 1222.) The list of n-letter words for any fixed n can be obtained similarly (and more simply).

3 Turing machines

No human being can write fast enough, or long enough, or small enough† to list all members of an enumerably infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain enumerably infinite sets: They can give explicit instructions fpr determining the nth member of the set, for arbitrary finite n. Such instructions are to be given quite explicitly, in a form in which they could be followed by a computing machine, or by a human who is capable of carrying out only very elementary operations on symbols. The problem will remain, that for all but a finite number of values of n it will be physically impossible for any human or any machine to actually carry out the computation, due to limitations on the time available for computation, and on the speed with which single steps of the computation can be carried out, and on the amount of matter in the universe which is available for forming symbols. But it will make for clarity and simplicity if we ignore these limitations, thus working with a notion of computability which goes beyond what actual men or machines can be sure of doing. The essential requirement is that our notion of computability not be too narrow, for shortly we shall use this notion to prove that certain functions are not computable, and that certain enumerable sets are not effectively (mechanically) enumerable – even if physical limitations on time, speed, and amount of material could somehow be overcome.

The notion of computation can be made precise in many different ways, corresponding to different possible answers to such questions as the following. 'Is the computation to be carried out on a linear tape, or on a rectangular grid, or what? If a linear tape is used, is the tape to have a beginning but no end, or is it to be endless in both directions? Are the squares into which the tape is divided to have addresses (like addresses of houses on a street) or are we to keep track of where we are by writing special symbols in suitable squares (as one might mark a trail in the woods by marking trees)?' And so on. Depending on how such questions are

† There is only a finite amount of paper in the world, so you'd have to write 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, ...

answered, computations will have one or another appearance, when examined in detail. But our object is not to give a faithful representation of the individual steps in computation, and in fact, *the very same set of functions turns out to be computable, no matter how these questions are answered.* We shall now answer these and other questions in a certain way, in order to get *a* characterization of the class of computable functions. A moderate amount of experience with this notion of computability will make it plausible that the net effect would have been the same, no matter in what plausible way the questions had been answered. Indeed, given any other plausible, precise characterization of computability that has been offered, one can prove by careful, laborious reasoning that our notion is equivalent to it in the sense that any function which is computable according to one characterization is also computable according to the other. But since there is no end to the possible variations in detailed characterizations of the notions of computability and effectiveness, one must finally accept or reject the *thesis* (which does not admit of mathematical proof) that the set of functions computable in our sense is identical with the set of functions that men or machines would be able to compute by whatever effective method, if limitations on time, speed, and material were overcome. Although this thesis ('Church's thesis') is unprovable, it *is* refutable, *if false.* For if it is false, there will be some function which is computable in the intuitive sense, but not in our sense. To show that such a function is computable in the intuitive sense, it would be sufficient to give instructions for computing its values for all (arbitrary) arguments, and to see that these instructions are indeed effective. (Presumably, we are capable of seeing *in particular cases*, that sets of instructions are effective.) To show that such a function is not computable in our sense, one would have to survey all possible Turing machines and verify (by giving a suitable proof) that none of them compute that function. (You will see examples of such proofs shortly, in Chapters 4 and 5; but the functions shown there not to be Turing computable have not been shown to be effectively computable in *any* intuitive sense, so *they* won't do as counterexamples to Church's thesis.) Then if Church's thesis is false, it is refutable by finding a counterexample; and the more experience of computation we have without finding a counterexample, the better confirmed the thesis becomes.

Now for the details. 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 a man stationed at each

end to add extra blank squares as needed. With at most a finite number of exceptions, all squares are blank, both initially and at each subsequent stage of the calculation. If a square is not blank, it has exactly one of a finite number of symbols $S_1, \ldots, S_n$ written in it. It is convenient to think of a blank square as having a certain symbol – the *blank* – written in it. We designate the blank by 'S_0'.

At each stage of the computation, the human or mechanical computer is *scanning* some one square of the tape. The computer is capable of telling what symbol is written in the scanned square. It is capable of erasing the symbol in the scanned square and writing a symbol there. And it is capable of movement: one square to the right, or one square to the left. 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. At the bottom of the car there is a device that can read what's written between the ties; erase such stuff; and write symbols there.

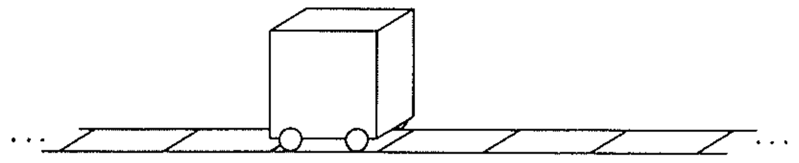


Figure 3-1

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 other 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 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 with him.) The man has a list of *m* instructions written down on a piece of paper. *He is in state q_i when he is carrying out instruction number i.*

Each of the instructions has conditional form: It tells the man what to do, depending on whether the symbol being scanned (the symbol in the scanned square) is S_0 or S_1 or ... or S_n. Namely, there are $n+4$ things he can do:

(1) Halt the computation.
(2) Move one square to the right.
(3) Move one square to the left.
(4) Write S_0 in place of whatever is in the scanned square.
(5) Write S_1 in place of whatever is in the scanned square.
⋮
($n+4$) Write S_n in place of whatever is in the scanned square.

So, depending on what instruction he is carrying out (= what state he is in) and on what symbol he is scanning, the man will perform one or another of these $n+4$ overt acts. Unless he has halted (overt act number 1), he will also perform a covert act, in the privacy of his box, *viz.*, he will determine what the next instruction (*next* state) is to be. Thus, the *present state* and the presently *scanned symbol* determine what overt *act* is to be performed, and what the *next state* is to be.

The overall *program* of instructions that the man is to follow can be specified in various ways, e.g. by a *machine table* or by a *flow graph* or by a *set of quadruples*. The three sorts of descriptions are illustrated in Figure 3-2 for the case of a machine which writes three symbols S_1 on a blank tape and then halts, scanning the leftmost of the three.

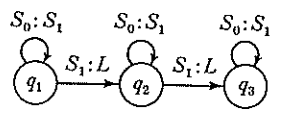
Example 3.1. Write $S_1 S_1 S_1$

The machine will write an S_1 in the square it's initially scanning, move left one square, write an S_1 there, move left one more square, write a third S_1 there, and halt. (It halts when it has no further instructions.) There will be three states – one for each of the symbols S_1 that are to be written. In Figure 3-2, the entries in the top row of the machine table (under the horizontal line) tell the man that when he's following instruction q_1 he is to (1) write S_1 and repeat instruction q_1, if the scanned symbol is S_0, but (2) move left and follow instruction q_2 next, if the scanned symbol is S_1. The same information is given in the flow graph by the two arrows that emerge from the node marked 'q_1'; and the same information is also given by the first two quadruples. The significance of a table entry, of an arrow in a flow graph, and of a quadruple, is shown in Figure 3-3.

Unless otherwise stated, it is to be understood that a machine starts in its

		Scanned symbol	
		S_0	S_1
Present state	q_1	$S_1 q_1$	$L q_2$
	q_2	$S_1 q_2$	$L q_3$
	q_3	$S_1 q_3$	

(*a*) *Machine table*



(*b*) *Flow graph*

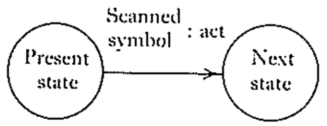
$q_1 S_0 S_1 q_1$, $q_1 S_1 L q_2$,
$q_2 S_0 S_1 q_2$, $q_2 S_1 L q_3$,
$q_3 S_0 S_1 q_3$

(*c*) *Set of quadruples*

Figure 3-2

Act Next state

(*a*) *Table entry*



(*b*) *Arrow in a flow graph*

Same sequence as for arrows

(*c*) *Quadruple*

Figure 3-3

lowest-numbered state. The machine we have been considering *halts* when it is in state q_3 scanning S_1, for there is no table entry or arrow or quadruple telling it what to do in such a case.

A virtue of the flow graph as a way of representing the machine program is that if the starting state is indicated somehow (e.g. if it is understood that the leftmost node represents the starting state unless there is an indication to the contrary) then we can dispense with the names of the states: It doesn't matter what you call them. Then the flow graph could be drawn as in Figure 3-4. We can indicate how such a machine operates by writing down its sequence of *configurations*. Each configuration shows what's on the tape at some stage of the computation, shows what state the machine is in at that stage, and shows which square is being scanned. We'll do this by writing out what's on the tape and writing the name of the present state under the symbol in the scanned square:

```
0000000  0000100  0000100  0001100  0001100  0011100
   1        1        2        2        3        3
```

Here we have written the symbols S_0 and S_1 simply as 0 and 1, and similarly we have written the states q_1, q_2, q_3 simply as 1, 2, 3 to save needless

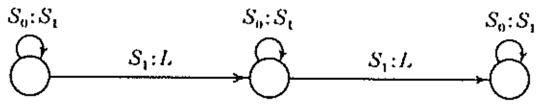


Figure 3-4. Write three S_1s.

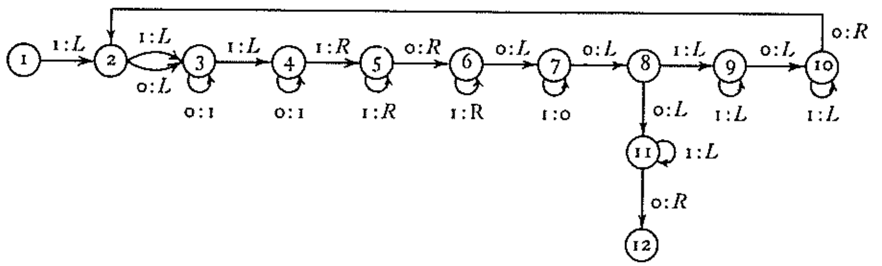
fuss. Thus, the last configuration is short for

$$S_0S_0\underset{q_3}{S_1}S_1S_1S_0S_0$$

Of course, the strings of S_0s at the beginning and end of the tape can be shortened or lengthened *ad lib.* without changing the significance: the tape is understood to have as many blanks as you please at each end. Now here is a more complex example.

Example 3.2. Double the number of 1s

This machine starts off scanning the leftmost of a string of 1s on an otherwise blank tape, and winds up scanning the leftmost of a string of twice that many 1s on an otherwise blank tape. Here is the flow graph:



How does it work? In general, by writing double 1s at the left, and erasing single 1s from the right. In particular, suppose that the initial configuration is $\underset{1}{1}11$, so that we start in state 1, scanning the leftmost of a string of three 1s on an otherwise blank tape. The next few configurations are

$$\underset{2}{0}111,\quad \underset{3}{0}0111,\quad \underset{3}{1}0111,\quad \underset{4}{0}10111,\quad \underset{4}{1}10111.$$

So we have written our first double 1 at the left – separated from the original string, 111, by a 0. Next we go right, past the 0 to the right-hand end of the original string, and erase the rightmost 1. Here is how that works, in two stages. Stage 1:

$$1\underset{5}{1}0111,\quad 11\underset{5}{0}111,\quad 110\underset{6}{1}11,\quad 1101\underset{6}{1}1,\quad 11011\underset{6}{1},\quad 110111\underset{6}{0}.$$

Now we know that we have passed the last of the original string of 1s, so (stage 2) we back up and erase one:

$$11011\underset{7}{1},\quad 11011\underset{7}{0}.$$

Now we hop back to our original position, scanning the leftmost 1 in what remains of the original string. When we have gone round the loop once more we shall be in this configuration: $111101\underset{7}{0}$. We'll then go through these:

$$11110\underset{8}{1},\quad 1111\underset{9}{0}1,\quad 111\underset{10}{1}01,\quad 11\underset{10}{1}101,\quad 1\underset{10}{1}1101,\quad \underset{10}{1}11101,\quad \underset{10}{0}111101,$$

$$\underset{2}{1}11101,\quad \underset{3}{0}111101,\quad \underset{3}{1}111101,\quad \underset{4}{0}1111101,\quad \underset{4}{1}1111101.$$

We now have our six 1s on the tape; but we want to erase the last of the original three and then halt, scanning the leftmost of the six that we wrote down. Here's how:

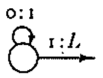
$$1\underset{5}{1}111101,\quad 11\underset{5}{1}11101,\quad \ldots,\quad 11111\underset{5}{1}01,\quad 111111\underset{5}{0}1,\quad 1111110\underset{6}{1},$$

$$11111101\underset{6}{0},\quad 1111110\underset{7}{1},\quad 1111110\underset{7}{0},\quad 111111\underset{8}{0},\quad 11111\underset{11}{1},$$

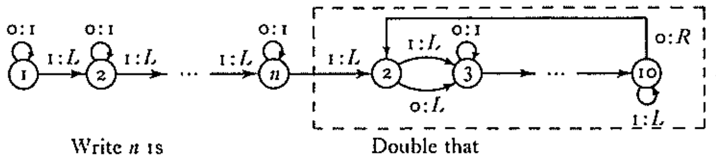
$$1111\underset{11}{1}1,\quad \ldots,\quad \underset{11}{1}11111,\quad \underset{11}{0}111111,\quad \underset{12}{1}11111$$

Now we are in state 12, scanning a 1. 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 1s 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 1s 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 1s no matter how long the original string may be.)

Example 3.3. Write $2n$ 1s on a blank tape and halt, scanning the leftmost 1

There is a straightforward way, using $2n$ states. We made use of the method in Example 3.1. In general, the flow graph would be obtained by stringing together $2n$ replicas of this: 

and lopping off the last arrow. But there is another way, using $n+11$ states: write n 1s on the tape in the straightforward way, and then use the method of Example 3.2 to double *that*. Schematically, we put the two graphs together by identifying node 1 in Example 3.2 with node n in the graph of the machine that writes n 1s on a blank tape. If n is large, the straightforward way of writing $2n$ 1s will require many more states than the indirect way which we have just



Graph of Example 3.2

described. (If n is greater than 11, $2n$ is greater than $n+11$.) Of course, we could save even more states by repeating the doubling trick, e.g. to write 100 1s we could use 25 states to write 25 of them, use another 11 to double that, and use yet another 11 to get the full 100 by doubling *that*. This does it with 47 states instead of the 100 we would need by the direct method, or the 61 we would need if we used the doubling trick only once.

Exercises. (Solutions follow)

3.1 Design a Turing machine which, started scanning the leftmost of an unbroken string of 1s on an otherwise blank tape, eventually halts, scanning a square on an otherwise blank tape – where the scanned square contains a blank or a 1 depending on whether there were an even or an odd number of 1s in the original string.

3.2 Design a Turing machine which, started scanning the leftmost of an unbroken string of 1s and 2s on an otherwise blank tape (in any order, and perhaps with all 1s or all 2s) eventually halts – at which point the contents of the tape indicate (following some convention which you are free to stipulate) whether or not there were exactly as many 1s as 2s in the original string.

3.3 *Add* 1 *in decimal notation.* Initially the machine scans the rightmost of an unbroken string of decimal digits on an otherwise blank tape. When it halts, the tape holds the decimal representation of 1 + the original number. Design a simple machine which does this. (Give the machine table – a graph would be messy.)

3.4 *Multiply in monadic notation.* Initially, the tape is blank except for two solid blocks of 1s, separated by a single blank. Say there are p 1s in the first block and q in the second. Design a machine which, started scanning the leftmost 1 on the tape, eventually halts, scanning the leftmost of an unbroken block of pq 1s on an otherwise blank tape.

3.5 *Add in in monadic notation.* (Identical with Exercise 3.4 but with '$p+q$' in place of 'pq'.)

3.6 *Not all functions from positive integers to positive integers are Turing computable.* You know via an Exercise in Chapter 2 that there are more such functions than there are Turing machines, so there must be some that no Turing machines compute. Explain a bit more fully.

3.7 *Productivity.* Consider Turing machines which use only the symbol 1 in addition to the blank. Initially the tape is blank. If the machine eventually halts, scanning the leftmost of an unbroken string of 1s on an otherwise blank tape, its *productivity* is said to be the length of that string. But if the machine never halts, or halts in some other configuration, its productivity is said to be 0. Now define $p(n)$ = *the productivity of the most productive n-state Turing machines.* Prove that

(*a*) $p(1) = 1$;
(*b*) $p(47) \geqslant 100$;
(*c*) $p(n+1) > p(n)$;
(*d*) $p(n+11) \geqslant 2n$.

(For solutions, see Chapter 4.)

3.8 *The busy beaver problem* (after Tibor Rado, *Bell System Technical Journal*, May 1962). According to Exercise 3.6, uncomputable functions exist. But what would such a thing look like? Actually, you have just met one: the function p defined in Exercise 3.7. The proof that it is uncomputable turns on the fact that the 'busy beaver problem' is unsolvable. This is the problem of designing a Turing machine which uses only the symbol 1 in addition to the blank and which, started in state 1 scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape, eventually halts scanning the leftmost of an unbroken string of $p(n)$ 1s on an otherwise blank tape. In fact, no such machine can exist. To prove that, assume that there *is* such a machine (with k states, say) and deduce a contradiction: That takes some ingenuity. *Hint*: if there were such a machine, then by stringing two replicas of it onto an n-state machine which writes n 1s on a blank tape (right to left) you can get an $n+2k$ state machine which has productivity $p(p(n))$; and from this, together with the last two things you proved in Exercise 3.7, you can deduce a contradiction.

$$\begin{tikzcd}
\boxed{\text{Write } n \text{ 1s}} \arrow[r] & \boxed{\text{Busy beaver machine}} \arrow[r] & \boxed{\text{Busy beaver machine}}
\end{tikzcd}$$

(For solution, see Chapter 4.)

3.9 *Uncomputability via diagonalization.* Every Turing machine which uses only the symbol 1 in addition to the blank computes some function from positive integers to positive integers according to the following convention. To compute the value which the function assigns to the argument n, start the machine in its lowest-numbered state, scanning the leftmost of a string of n 1s on an otherwise blank tape. The function is defined for the argument n if and only if the machine eventually halts, scanning the leftmost of an unbroken string of 1s on an otherwise blank tape, in which case the value of the function is the length of the string. Since the Turing machines are enumerable (cf. Exercise 3.6), so are the functions they compute. Define an 'antidiagonal' function and prove that no Turing machine computes it. (For solution, see Chapter 5.)

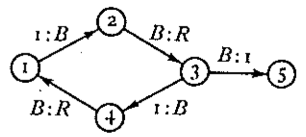
3.10 *The halting problem.* Let $M_1, M_2, \ldots$ be an enumeration of all Turing machines which use only the symbol 1 in addition to the blank. The *self-halting problem* is that of specifying a uniform effective procedure in some format or other for computing a total or partial function s for which we have $s(n) = 1$ if and only if M_n never halts, after being started in its initial state, scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape. The full *halting problem* is that of specifying a uniform effective procedure for computing some function h for which we have $h(m, n) = 1$ if and only if M_m never halts, after being started in its initial state, scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape.

(*a*) Prove that the self-halting problem is unsolvable.

(*b*) Prove the corollary: the halting problem is unsolvable.

Hint: Church's thesis must be used, sooner or later. Sooner *and* later: prove that one of the functions shown in Exercise 3.6 to be computable by no Turing machine is computable in an unusual format (using two Turing machines at once) if the self-halting problem is solvable.

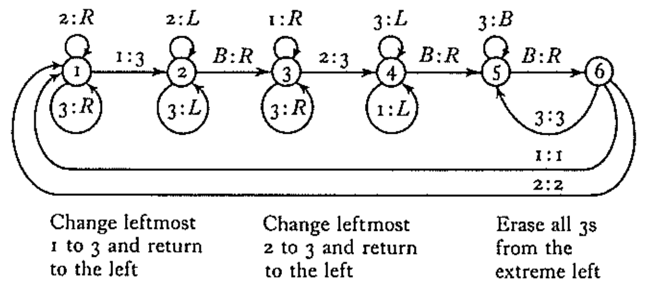
Solutions



(We have written 'B' for S_0, the blank.)

3.1 If there were 0 or 2 or 4 or ... 1s 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 1 on an otherwise blank tape.

3.2 When this machine halts, the tape is blank if and only if there were equal numbers of 1s and 2s in the original string.


2:R
1:3
2:L
B:R
1:R
2:3
3:L
B:R
3:B
B:R
3:R
3:L
3:R
1:L
3:3
1:1
2:2
Change leftmost 1 to 3 and return to the left
Change leftmost 2 to 3 and return to the left
Erase all 3s from the extreme left

There are other ways of doing this, e.g. ways in which the machine uses no symbols other than 1 and 2. The simplest way of doing it which is compatible with the letter, but not the spirit of the problem statement, is to adopt the following convention regarding the way in which the contents of the tape when the machine halts indicate whether or not there were equal numbers of 1s and 2s initially: there were equal numbers of 1s and 2s initially if and only if there are equal numbers of 1s and 2s on the tape when the machine halts. The machine can then halt immediately, so that the final tape contents are identical with the initial contents. The graph of such a machine would look like this: ①

3.3 There are 11 possible scanned symbols, among which B is the blank and 0 is not the blank but the cipher zero; 1, ..., 9 are the usual ciphers. In the body of the table, each pair represents the *act to be per-*

Machine table for adding 1 in decimal notation

Scanned symbol	Present state 1	2
B	1 2	
0	1 2	L 1
1	2 2	
2	3 2	
3	4 2	
4	5 2	
5	6 2	
6	7 2	
7	8 2	
8	9 2	
9	0 2	

formed and the *next state*, in that order. The many blanks in column 2 represent situations in which the machine has no instructions, and therefore halts.

3.4 There are lots of equally efficient ways of multiplying in monadic notation. Many of them involve use of extra symbols beyond the B and 1 that appear on the tape initially and finally, and that is perfectly all right: they are symbols which the machine uses to keep track of where it is in the process, and which it erases before halting. But the method shown uses only B and 1 throughout the process. We have done that in order to illustrate this point: *any function from positive integers to positive integers which is computable at all is computable in monadic notation by a machine which uses only the symbols B and 1.*

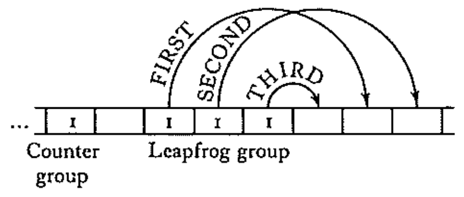
Here is how this machine works. The first block, of p 1s, is used as a *counter*, to keep track of how many times the machine has added q 1s to the group at the right. To start, the machine erases the leftmost of the p 1s and sees if there are any 1s left in the counter group. If not, $pq = q$, and all the machine has to do is position itself over the leftmost 1 on the tape, and halt. But if there are any 1s left in the counter the machine goes into its *leapfrog routine*: in effect, it moves the block of q 1s (the 'leapfrog group') q places to the right along the tape, e.g. with $p = 2$ and $q = 3$ the tape looks like this initially

$11B111$

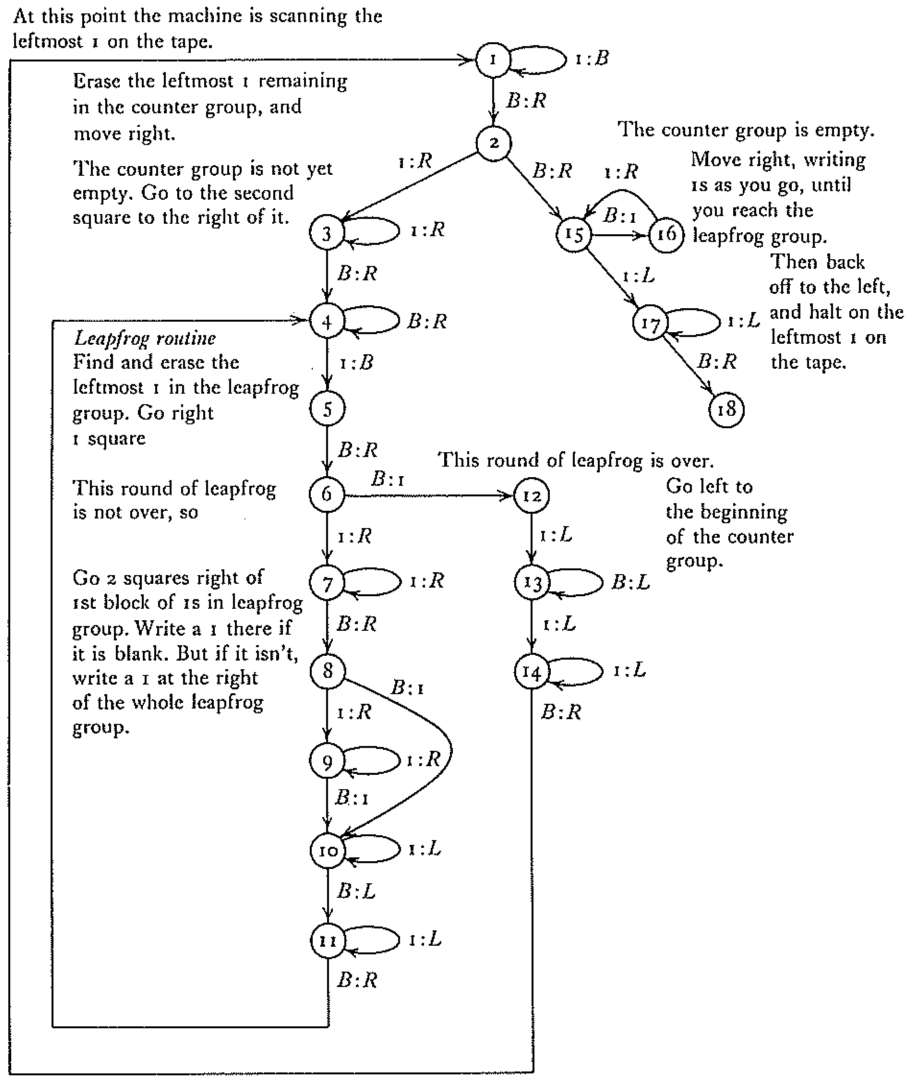
and looks like this after going through the leapfrog routine:

$B1BBBB111$

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 Bs to 1s until it comes to a 1, at which point it continues on to the leftmost 1 and halts. This is how the leapfrog routine works:



In general, the leapfrog group consists of a block of 0 or 1 or ... or q 1s, followed by a blank, followed by the remainder of the q 1s. The blank is there to tell the machine when the leapfrog game is over: without it the


At this point the machine is scanning the leftmost 1 on the tape.
1
1:B
B:R
Erase the leftmost 1 remaining in the counter group, and move right.
2
The counter group is not yet empty. Go to the second square to the right of it.
1:R
3
1:R
B:R
The counter group is empty. Move right, writing 1s as you go, until you reach the leapfrog group.
B:R
15
1:R
B:1
16
Then back off to the left, and halt on the leftmost 1 on the tape.
1:L
17
1:L
B:R
18
4
B:R
Leapfrog routine
Find and erase the leftmost 1 in the leapfrog group. Go right 1 square
1:B
5
B:R
This round of leapfrog is not over, so
6
B:1
This round of leapfrog is over. Go left to the beginning of the counter group.
12
1:L
13
B:L
1:L
14
1:L
B:R
1:R
Go 2 squares right of 1st block of 1s in leapfrog group. Write a 1 there if it is blank. But if it isn't, write a 1 at the right of the whole leapfrog group.
7
1:R
B:R
8
B:1
1:R
9
1:R
B:1
10
1:L
B:L
11
1:L
B:R

Flow graph for multiplying in monadic notation

group of q 1s would keep moving right along the tape forever. (In playing leapfrog, the portion of the q 1s to the left of the blank in the leapfrog group functions as a counter: it controls the process of adding 1s to the portion of the leapfrog group to the right of the blank. That is why there are two big loops in the flow graph: one for each counter-controlled subroutine.)

3.5 The object is to erase the leftmost 1, fill the gap between the two blocks of 1s, and halt scanning the leftmost 1 that remains on the tape. Here is one way of doing it, in quadruple notation: $q_1S_1S_0q_1$; $q_1S_0Rq_2$; $q_2S_1Rq_2$; $q_2S_0S_1q_3$; $q_3S_1Lq_3$; $q_3S_0Rq_4$.

3.6 According to Exercise 2.3 the set of all functions from positive integers to positive integers is not enumerable. On the other hand, the set of all Turing machines *is* enumerable, for any Turing machine is describable by a finite string of letters of the infinite alphabet

$$;, R, L, S_0, q_1, S_1, q_2, S_2, q_3, \ldots$$

and according to Exercise 2.4(*d*) (with $a_1 = ;$, $a_2 = R$, $a_3 = L$, $a_4 = S_0$, $a_5 = q_1, \ldots$) the set of all such strings is enumerable. (We need a convention for identifying the starting state, e.g. we might require that the starting state be assigned the lowest q-number.) Then whatever convention one may adopt, for deciding what (if anything) the function is which is computed by each Turing machine in our list, our list of Turing machines yields a (possibly gappy) list of *all* the functions from positive integers to positive integers which are computed by Turing machines according to our convention. One possible convention is the one specified in Exercise 3.4, according to which we may take it that Turing machines whose quadruples contain any of the symbols $S_2, S_3, \ldots$ compute *no* functions from positive integers to positive integers. Note that if $f_1, f_2, f_3, \ldots$ is a gapless list of all functions computed by Turing machines according to our convention, the antidiagonal function u of Exercise 2.3 is an example of a function from positive integers to positive integers which is computed in our format by no Turing machine. Another example is the partial function t which we define:

$$t(n) = \begin{cases} 1 & \text{if } f_n(n) \text{ is undefined,} \\ \text{undefined} & \text{if } f_n(n) \text{ is defined.} \end{cases}$$

(For further details, see Chapter 5.)

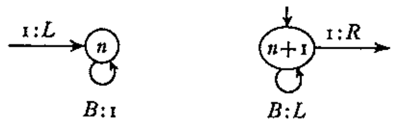
3.10(*a*) If the self-halting problem were solvable, then by Church's thesis there would be some Turing machine S which computes s in some format, e.g. in the format of Exercise 3.9. The function t of Exercise 3.6 would then be computable in the following format. To determine $t(n)$, start machines S and M_n in their initial states, scanning the leftmost of unbroken strings of n 1s on their otherwise blank tapes.

Case 1: *M_n never halts.* Then S will eventually halt, scanning a 1 on its otherwise blank tape. At that time we shall know that M_n will never halt, and thus we shall know that the corresponding function f_n is undefined for the argument n, so that $t(n) = 1$.

Case 2: *M_n eventually halts.* When it does, we can determine by examining the tape whether or not $f_n(n)$ is defined, and will thus know whether $t(n)$ is undefined or equal to 1. Then if the self-halting problem is solvable, t is computable in an unusual format, and hence, by Church's thesis, t is computed by some Turing machine, in contradiction to what we have proved in Exercise 3.6.

3.10(*b*) If the halting problem were solvable, i.e., if h were computable, then the self-halting problem would be solvable, i.e., s would be computable, for we have $s(n) = h(n, n)$ for every positive integer n.

state may well have other functions: the machine may return to it at various later stages of its operation, if it occurs in closed loops, as does the starting state of the multiplier in Exercise 3.4.) Thus, for all we know, the halting state of the 'Write n 1s' machine and the starting state of BB might clash, e.g. as follows:



Superimposing these states, the machine would have conflicting instructions about what to do in state n ($= n+1$) when scanning a blank.

5 Uncomputability via diagonalization

In this chapter we exhibit various uncomputable functions as in Exercises 3.8, 3.9 and 3.10. In calling these functions simply 'uncomputable' instead of 'uncomputable by Turing machines' we are presupposing Church's thesis, evidence for which will be given in Chapters 6–8. Throughout this chapter, 'function' will mean *function from positive integers to positive integers*.

The bare existence of uncomputable functions is quickly proved, in a rough and ready way, for we know (Exercise 2.4(d)) that the set of all Turing machines (represented by finite strings of quadruples) can be enumerated, whereas (Exercise 2.3) the set of all functions cannot. If the set of Turing computable functions were identical with the set of all functions, that set would be both enumerable and not. Therefore, the two sets are distinct: the former is a proper subset of the latter.

The foregoing proof is rough and ready in that we have not yet precisely defined the notion of Turing computability of functions from positive integers to positive integers. Turing machines operate on symbols, not numbers. Before we can speak of Turing machines as computing numerical functions, we must specify the notation in which the numerical arguments and values are to be represented on the machine's tape, e.g. as in the adder and multiplier of Exercises 3.4 and 3.5. The choice of monadic notation makes for simplicity, but it is not essential: The decimal notation, or various others, would serve as well. What *is* essential is that such particulars as notation *be* specified, in one way or another. Our specifications are as follows.

(a) The arguments of the function will be represented in monadic notation by blocks of 1s, separated by single blanks, on an otherwise blank tape. Then 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. Then if the machine's initial state is 1, the initial configuration in computing $3+2$ will be this: $\underset{1}{1}11B11$.

(c) If the function which is to be computed assigns a value to the arguments which are represented initially on the tape, the machine will eventually *halt in a standard final configuration*, i.e., scanning the leftmost of a block of 1s on an otherwise blank tape. The number of 1s in that block will be the value which the function assigns to the given arguments. Thus, the final configuration in the computation of $3+2$ by the adder in Exercise 3.4 will be $\underset{4}{1}1111$.

(*d*) If, on the other hand, the function assigns no value to the given arguments, the machine will not eventually halt in a standard final configuration: It will either go on running forever, or will eventually halt in some such nonstandard final configuration as $\underset{1}{B}111$ or $\underset{1}{1}B1$ or $\underset{1}{1}11$, etc.

The initial configurations described in (*a*) and (*b*) above will be called 'standard' initial configurations, just as the final configurations described in (*c*) are called 'standard' final configurations.

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 *n* arguments, for each positive integer *n*. Thus, consider the machine which is specified by the single quadruple '$q_1 1 1 q_2$' with initial state q_1. Started in a standard initial configuration, it halts immediately, leaving the tape unaltered. If there was only a single block of 1s on the tape initially, its final configuration will be standard, and thus this machine computes the identity function, *i*, of one argument: $i(n) = n$ for each positive integer *n*. Then the machine computes a certain total function of one argument, but if there are two or more blocks of 1s on the tape initially, the final configuration will not be standard. Accordingly, the machine computes the extreme partial function of two arguments which is undefined for all pairs of arguments. And in general, for *n* arguments, this machine computes the extreme partial function which assigns values to no *n*-tuples of arguments.

On the other hand, the machine of Figure 5-1 computes, for each *n*, the total function which assigns the same value, *viz.*, 1, to each *n*-tuple of argument. Started in state 1 in a standard initial configuration, this machine erases the first block of 1s (cycling between states 1 and 2) 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 final configuration. But if it sees a 1 there, it reenters the cycle between states 1 and 2, erasing the second block of 1s and inquiring again, in state 3, whether the whole tape is blank, or whether there are one or more blocks still to be dealt with.

In this chapter, we shall be concerned with functions of a single argument. We show that the set of all such functions cannot be enumer-

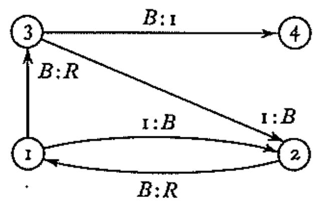


Figure 5-1. $f(x_1, \ldots, x_n) = 1$.

ated, while the set of functions of one argument which are computed by Turing machines can be enumerated. And by attending to the details of that enumeration, we define a particular function, *t*, of one argument, which is not computed by any Turing machine. According to our specifications, a machine *M* computes the function *f* of one argument if and only if for each $n = 1, 2, 3, \ldots$ for which $f(n)$ is defined, and for no other *n*s, *M* eventually halts in a standard configuration with $f(n)$ 1s on its tape when it is started in a standard configuration with *n* 1s on its tape. The function *f* is said to be *Turing computable* if and only if it is computed by some machine *M*, according to these specifications.

We enumerate the Turing computable functions of one argument by enumerating the Turing machines. To this end, we specify each machine by writing its quadruples in a single string, with no spaces separating adjacent quadruples, beginning with a quadruple whose first symbol is the machine's initial state. Example: the machine of figure 5-1, which has initial state q_1, is represented by the string,

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

Clearly, each Turing machine is specified in this way by some finite string of letters of the enumerably infinite alphabet

$$R, L, S_0, q_1, S_1, q_2, S_2, q_3, S_3, \ldots \qquad (1)$$

Not every word on this alphabet specifies a Turing machine. A word specifies a Turing machine if and only if it meets these conditions:

(*a*) Its length is exactly divisible by 4.
(*b*) Only the symbols q_1, q_2, q_3, etc. occur in positions $1, 4, 5, 8, 9, 12, \ldots, 4n, 4n+1, \ldots$
(*c*) Only the symbols S_0, S_1, S_2, etc. occur in positions $2, 6, 10, \ldots, 4n+2, \ldots$ (2)
(*d*) Only the symbols $R, L, S_0, S_1, \ldots$ occur in positions $3, 7, 11, \ldots, 4n+3, \ldots$
(*e*) No configuration of form $q_i S_j$ occurs more than once in the word.

Condition (*e*) is a bit arbitrary. It serves to rule out contradictory instructions, but that purpose would have been served as well by this variant, which allows repetitions of the same quadruple: If a configuration of form $q_i\,S_j$ occurs more than once in the word, the next pair of symbols must be the same at each occurrence.

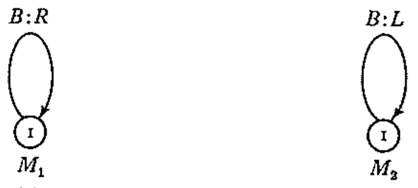
Now by Exercise 2.4(*d*) we know that the set of all words on the enumerably infinite alphabet (1) is enumerable: to speak vividly, it is possible to arrange them in a single (possibly, gappy) list, $w_1, w_2, w_3, \ldots$. If from this list we delete the words which fail to specify Turing machines, we are left with a gappy list in which each Turing machine is named at least once, and nothing else is named. Now (still speaking vividly) let us close the gaps to obtain a gapless list $M_1, M_2, M_3, \ldots$ of all Turing machines. Since every Turing machine determines a definite function from positive integers to positive integers, this list of machines determines a list

$$f_1, f_2, f_3, \ldots \tag{3}$$

of all Turing computable functions of one argument, where for each n, f_n is the total or partial function of one argument which machine M_n computes.

Then as soon as we specify a particular enumeration $w_1, w_2, w_3, \ldots$ of the words on the alphabet (1), we shall have specified a particular enumeration (3) of the Turing computable functions of a single argument. Let us enumerate the *w*s by the method which was described at the end of Chapter 2. This gives us a gappy list in which the first entry is w_{12}, *viz.* R. In general, the position of a word in this list is given, in decimal notation, by the string obtained from that word by replacing each occurrence of R by 12, of L by 122, of S_0 by 1222, and so on: The *n*th letter in alphabet (1) is replaced by a 1 followed by *n* 2s.

Having determined the sequence of *w*s, it is now trivial (but laborious) to determine the sequence of *M*s. The first of the *w*s which determines a Turing machine (i.e., which meets conditions (2)) is $q_1S_0Rq_1$. This is w_n for $n = 1\,222\,212\,221\,212\,222$: A quadrillion and then some. Then M_1 is as shown in Figure 5-2(*a*). In any standard initial configuration it is scanning a 1, and therefore halts immediately, in its initial position. Then



(*a*) (*b*)

Figure 5-2

f_1 is i, the identity function: we have $f_1(n) = n$ for each n. Furthermore, f_2 is the same function as f_1, for the second of the *w*s which represent Turing machines is '$q_1S_0Lq_1$', which is w_n for $n = 12\,222\,122\,212\,212\,222$. Then M_2 is as in Figure 5-2(*b*): It too, halts immediately, in a standard final configuration, when it is started in a standard configuration with a single block of 1s on the tape.

We have now enumerated the Turing computable functions of one argument by enumerating the machines which compute them. The fact that such an enumeration is possible shows that there must exist uncomputable functions of a single argument. The point of actually specifying one such enumeration is to be able to exhibit a particular such function, *viz.*, the function u which is defined as follows. (Cf. Exercises 2.3 and 3.6.)

$$u(n) = \begin{cases} 1 & \text{if } f_n(n) \text{ is undefined,} \\ f_n(n) + 1 & \text{otherwise.} \end{cases} \tag{4}$$

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

$$u(m) = f_m(m) = \begin{cases} 1 & \text{if } f_m(m) \text{ is undefined,} \\ f_m(m) + 1 & \text{otherwise.} \end{cases} \tag{5}$$

Then whether f_m is or is not defined for the argument m, we have a contradiction: Either $f_m(m)$ is undefined, in which case (5) tells us that it has the value 1 (and is thus both defined and undefined), or $f_m(m)$ is defined and equal to its own successor (from which we deduce that $0 = 1$). Since we have derived a contradiction from the assumption that u appears somewhere in the list $f_1, f_2, \ldots, f_m, \ldots$, we may conclude that the supposition is false: We conclude that the function u is not Turing computable.

Although no Turing machine computes the function u, we can readily determine its first few values. Thus, since $f_1 = f_2 = i =$ the identity function, we have $f_1(1) = 1$ and $f_2(2) = 2$, so that by (4) we have $u(1) = 2$ and $u(2) = 3$. Note that there is no contradiction here, or in the definition (4) of u. Equation (5) was contradictory because it incorporated the assumption that u is Turing computable: Computed by machine M_m for some positive integer m. The contradiction in (5) refutes that assumption, but does not impeach definition (4).

Now it may seem that we can actually compute $u(n)$ for any positive integer n – if we don't run out of time. Certainly, we have discovered the values that u assigns to the arguments 1 and 2, and it may seem that it must be perfectly routine to do this for as many subsequent arguments as time permits. Here, the real question is, whether we ever really need luck or ingenuity to discover the values which u assigns to its arguments. It may seem that we do not, for it is straightforward to discover which quadruples determine M_n for $n = 1, 2, 3, \ldots$. (This is straightforward, but, eventually, humanly impossible 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 notion of computability, we ignore the fact that human life is limited.) The essential question concerns the business of determining whether machine M_n, started scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape, does or does not eventually halt scanning the leftmost of an unbroken string of 1s on an otherwise blank tape. Is *that* perfectly routine? If so, the function u is computable after all – although not by any Turing machine.

But *is* it perfectly routine, to discover whether M_n eventually halts in standard position as above? 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 shall eventually get that information by following its operations. But what if M_n is destined never to halt, given the initial configuration? 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 certainly so, e.g. it is obvious that the quadruple $q_1S_1S_1q_1$ represents a machine which never halts if it starts scanning a square with a 1 in it, and it is obvious that the machine described by the string $q_1S_1S_0q_1q_1S_0Lq_1$ will never halt. But when we consider more complicated cases it is by no means evident that there is a uniform, mechanical procedure for discovering whether or not a given machine, started in a given configuration, will ever halt. Thus, consider the multiplier of Exercise 3.4. It is described by various strings of 124 symbols on the alphabet (1). Imagine encountering one such string in the list w_1, w_2, ... It is routine to verify that the string describes a Turing machine, and one can easily enough derive from that string a flow graph like the one shown in Chapter 3 but without the annotation, and without the accompanying text which explains the point and the workings of the leapfrog routine and the rest. Is there a uniform method which, in this and in much more complicated cases, allows one to determine whether the machine eventually halts in a standard final configuration, once the initial tape configuration has been specified? If so, the function u is computable, although not by any Turing machine. But if not, we have failed in our attempt to see u as a function which is computable in some way, but not in Turing's way.

The crucial problem is to specify a uniform routine which, given any machine's quadruples and a description of any initial tape configuration, determines after some finite number of steps whether the machine would eventually halt in a standard configuration, having been started in the given initial configuration. If there is such a routine, then Turing's notion of computability is too narrow: there are then computable functions which are not Turing computable, and in particular, u is such a function. But if Turing's notion of computability is broad enough to encompass all functions which are computable in any intuitive sense, this problem is unsolvable.

A closely related problem, likewise unsolvable if Turing's notion of computability is universal, is the halting problem of Exercise 3.10: specify a uniform effective procedure for discovering whether or not Turing machines eventually halt (at all – whether in standard final configuration or in some other configuration) when started in their initial states scanning the leftmost 1 in an unbroken string of them on an otherwise blank tape. It is noteworthy that whether or not Church's thesis is to be accepted, the following function h, which is computable (in the intuitive sense) if and only if the halting problem is solvable, can be proved not to be Turing computable: $h(m, n) = 2$ or 1 accordingly as machine number m does or does not eventually halt after being started in its initial state scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape. Therefore *if Church's thesis is correct, the halting problem is unsolvable.*

The proof that h is not Turing computable which we now give is a more direct alternative to the proof given in Exercise 3.10: we do not now deduce the uncomputability of h from that of u, say, but establish it directly, by a diagonal argument. Toward that end we have specified h completely, and as a total function, whereas in Exercise 3.10 we specified only the conditions under which h assumes the value 1. The argument is by *reductio ad absurdum*: we show that if h is computed by some Turing machine H, then there must exist another Turing machine M_m such that for all positive integers n, M_m halts if and only if machine M_n does not, after being started in its initial state, scanning the leftmost of an unbroken string of n 1s. But of course there can be no such machine M_m, for if there were, it would have to halt if and only if it never halts, after being started in its initial state scanning the leftmost of an unbroken string of m 1s on an otherwise blank tape.

The truth of the claim that if h is computed by some Turing machine H then there exists a Turing machine M_m of the sort described can be seen by considering Figure 5-3. The 'H' box in Figure 5-3 represents the

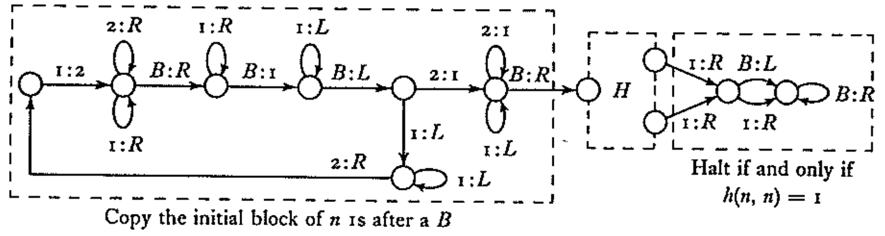

1:2
2:R
1:R
B:R
1:R
B:1
1:L
B:L
2:1
1:L
2:R
1:L
2:1
B:R
1:L
H
1:R
B:L
B:R
1:R
1:R
Copy the initial block of n 1s after a B
Halt if and only if $h(n, n) = 1$

Figure 5-3. Graph of M_m.

graph of H. The node on the left represents its initial state. Since h is total there will be one or more nodes in the graph of H from which emerge no arrows with 1s before their colons. All the states in which H might halt are represented by such nodes. (So, perhaps, are some states in which H does not halt because in computing values of h, H never scans a 1 while in those states.) We have supposed, for definiteness, that there are just two such states, which are indicated by the nodes at the right of the box. The other states and arrows of H are hidden inside the box.

Figure 5-3 thus represents the graph of a machine M_m which contains the machine H as a part and which, when started in its initial state (= leftmost node) scanning the leftmost of a block of n 1s on an otherwise blank tape, first recopies that block to the right, leaving a blank between the two blocks, and then sets H to work on the pair of blocks. When H is through, the whole machine halts if and only if H has left exactly one 1 on the tape. (H will always end by scanning the left of either 1 or 11 on an otherwise blank tape.)

Thus M_m halts if and only if M_n does not (if and only if $h(n,n) = 1$) when started in its initial state scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape. Setting $m = n$, we see that M_m cannot exist, and thus H cannot exist: the function h is not Turing computable, whether or not Church's thesis is true. In other words, the halting problem is solvable by no Turing machine. If Church's thesis is true, it is absolutely unsolvable.

Exercises

5.1 What are the functions of one and of three arguments which are computed by the adder of Exercise 3.5?

5.2 Identify M_3 and M_4, and determine the values $u(3)$ and $u(4)$.

5.3 Define the function t as follows:

$$t(n) = \begin{cases} 1 & \text{if } f_n(n) \text{ is undefined,} \\ \text{undefined} & \text{if } f_n(n) \text{ is defined.} \end{cases}$$

Prove (without using Church's thesis) that t is not Turing computable.

5.4 Define $f(m, n) = f_m(n)$ for each $m, n = 1, 2, \ldots$. Is f computable?

Solutions

5.1 The adder of Exercise 3.5 computes the identity function of one argument, and computes the extreme partial function of three arguments which assigns values to no triples of positive integers.

5.2 M_3 is $q_1 S_0 R q_2$ and M_4 is $q_1 S_0 S_0 q_1$. Both compute the identity function of one argument, so $u(3) = 4$ and $u(4) = 5$.

5.3 If t were Turing computable – say, by machine M_k in our enumeration – we should have

$$t(k) = f_k(k) = \begin{cases} 1 & \text{if } f_k(k) \text{ is undefined,} \\ \text{undefined} & \text{if } f_k(k) \text{ is defined.} \end{cases}$$

Then the hypothesis that t is Turing computable leads to a contradiction (without use of Church's thesis) and is therefore false.

5.4 Yes. (Use Church's thesis.)

6
Abacus computable functions are Turing computable

Put very broadly, Church's thesis says that any mechanical routine for symbol manipulation can be carried out in effect by some Turing machine or other. Thus, multiplication of positive integers, which we tend to think of as a mechanical routine in two dimensions involving the ciphers 0, 1, ..., 9, can be performed one-dimensionally by a Turing machine which reads and prints the ten ciphers, the blank, and perhaps a few extra symbols which it uses in the course of the computation for book-keeping purposes but which do not appear on the tape initially or when the machine halts. But in Exercise 3.4 we saw how multiplication can also be carried out by a Turing machine which has these special characteristics:

(1) The two numbers to be multiplied are represented by blocks of 1s, separated by a single blank, on an otherwise blank tape, and the value is represented by a single block of 1s on an otherwise blank tape, when the machine halts.

(2) The machine starts and halts only in standard position, i.e. scanning the leftmost 1 on the tape.

(3) Throughout the computation, the machine never goes more than two squares to the left of the leftmost 1 that was on the tape initially. Then the tape need only be infinite in one direction: to the right.

(4) The machine reads and writes no symbols other than B and 1, i.e. the machine's quadruples are words on the alphabet

$$R, L, B, 1, q_1, q_2, q_3, \ldots$$

Let us now reserve the term 'Turing computable' for functions which are computable by Turing machines in accordance with these four restrictions and with the further restriction (5) that the leftmost 1 in the block of 1s representing the value of the function appear in the very same square of the tape in which, initially, the leftmost 1 appeared. The evidence for Church's thesis which we shall adduce in this and the following two chapters is a survey of the Turing computable functions: a survey which shows them to comprise a remarkably broad class, which

it is plausible to suppose inclusive of all functions from positive integers to positive integers that are effectively computable at all.

The restriction to monadic notation (instead of, say, decimal notation) and to the standard position (scanning the leftmost 1) for starting and halting is inessential; but *some* assumptions had to be made about initial and final positions of the machine, and these assumptions seem especially simple. It is even possible to describe a process whereby any decimal computation can be transformed into one that satisfies conditions (1)–(5) above: a process whereby the flow graph of a decimal machine which computes a certain function can be converted step by step into the flow graph of a machine which computes the same function in a format which meets conditions (1)–(5).

In broadest outline, the process is this. Code the ciphers 0, 1, ..., 9 as strings of 1s: 1, 11, ..., 111111111. Then e.g. decimal 213 would look like this: 111$B$11$B$1111. It is not hard to design a monadic-to-coded decimal converter, i.e. a machine which, started in standard position scanning a string of n 1s, eventually halts in standard position scanning the coded decimal form of the number n. (Thus, for $n = 10$, the initial tape contents would be 1111111111 and the final contents would be 11B1.) Similarly, one can design a coded-decimal-to-monadic converter. And one can modify the flow graph of the given decimal machine so as to obtain the flow graph of a machine which behaves in essentially the same way, except that it works with the *coded* forms of the ten ciphers, and meets condition (3). (That might be arranged by systematically modifying the graph so that every movement of the machine to the left on the tape is preceded by a routine which moves all 1s on the tape a single square to the right. And *that* is a matter of replacing each arrow labelled $B:L$ or $1:L$ by a graph which executes the required general movement to the right and then returns to the main routine at the appropriate position.) Now the required standardized routine is obtained by stringing together, in order, a monadic-to-coded-decimal converter, the transformed graph of the given decimal machine, and a coded-decimal-to-monadic converter.

It would be laborious to specify this process in full detail; nor, after all that work, would we have succeeded in showing that monadic notation can serve as well as *any* other, for purposes of Turing computation. No end of notations might be invented, and there is no hope of proving that everything computable in any of them is computable in monadic notation. It is for this reason that we adopt the monadic notation at the outset: *define* Turing computability as computability in a format which satisfies conditions (1)–(5); and interpret

Church's thesis: all computable functions are Turing computable

in the light of that definition.

As an important part of the evidence for Church's thesis we show how the operations of the ordinary sort of high-speed digital computer can be simulated by a Turing machine which meets requirements (1)–(5). By a digital computer of the 'ordinary' sort we mean a device which has an unlimited amount of *random-access storage*. In contrast to a Turing machine, which stores information symbol by symbol on squares of a one-dimensional tape along which it can move a single step at a time, a machine of the seemingly more powerful 'ordinary' sort has access to an unlimited number of *registers* R_0, R_1, R_2, ..., in each of which can be written numbers of arbitrary size. Thus, the usual sort of machine is a bit like a generalized Turing machine which can read and write an infinite variety of symbols S_0, S_1, S_2, ..., any one of which can appear in any square of its tape. (The symbols S_n might represent the number n.) But the usual sort of machine has still another feature – random access – which allows it to go directly to register R_n without inching its way, square by square, along the tape. That is, each register (R_n, say) has its own *address* (the number n) which allows the machine to carry out such instructions as

compute the sum of the numbers in registers R_m and R_n, and store it in register R_p,

which we abbreviate:

$$[m]+[n] \rightarrow p$$

(In general, $[n]$ is the number in register R_n, and the number at the right of an arrow identifies the register in which the result of the operation at the left of the arrow is to be stored.)

It should be noted that our 'usual' sort of computing machine is really quite unusual in one respect: although real digital computing machines often have random-access storage, there is always a finite upper limit on the size of the numbers which can be stored, e.g. a real machine might have the ability to store any of the numbers 0, 1, ..., 10 000 000 in each of its registers, but no number greater than ten million. Thus, it is entirely possible that a function which is computable by one of our idealized 'usual' machines is not computable by any real machine, simply because, for certain arguments, the computation would require more

capacious registers than any real machine possesses. (Indeed, addition is a case in point: there is no finite bound on the sizes of the numbers one might think of adding, and hence no finite bound on the sizes of the registers needed for the arguments and the sum.) But this is in line with our objective of abstracting from technological limitations so as to arrive at a notion of computability which is *not too narrow*. We seek to show that certain functions are uncomputable in an absolute sense: uncomputable even by our idealized machines, and, therefore, uncomputable by any past, present, or future real machine.

One last point: instead of working with functions whose arguments and values are positive integers 1, 2, 3, ... we shall now work with functions whose arguments and values are natural numbers (non-negative integers) 0, 1, 2, ... We do this simply in order to conform with current fashion and thus ease comparison of what we say here with what is said in most books on the subject. The change is simply a matter of interpreting a solid block of one or more 1s on a Turing machine tape as representing the natural number $n-1$ instead of the positive natural number n. Thus, in a computation of $0+2$ in our standard format the tape would have $1B111$ on it initially and would have 111 on it finally.

In order to avoid discussion of electronic or mechanical details we may imagine the 'usual' sort of machine in crude, stone age terms. Each register may be thought of as a roomy, numbered box capable of holding any number of stones: none or one or two or ..., so that $[n]$ will be the number of stones in box number n. The 'machine' is operated by a man who is capable of carrying out two sorts of operations:

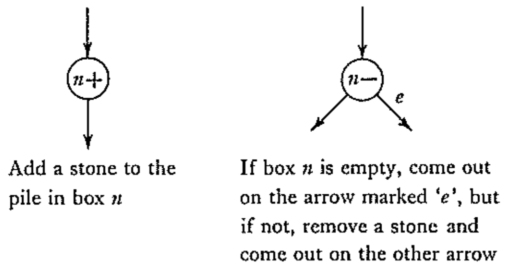
add a stone to the pile in box number n

and

remove a stone from the pile in box number n if there are any there.

A program can be specified by a flow chart which indicates the sequence in which these two sorts of operations are to be performed. Typically, the program will have one or more *branch points* at which the next operation will be one thing or another depending on whether the box from which a stone is to be removed (if possible) is already empty. Thus, branch points are associated with elementary operations of the second sort, above. Following Joachim Lambek, we shall refer to such an 'ordinary' machine as an (infinite) *abacus*.

The elementary operations will be symbolized as in Figure 6-1.

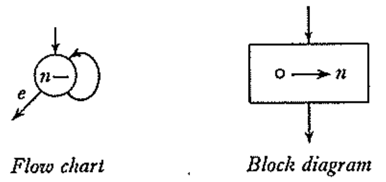


n+
Add a stone to the pile in box n
n−
e
If box n is empty, come out on the arrow marked 'e', but if not, remove a stone and come out on the other arrow

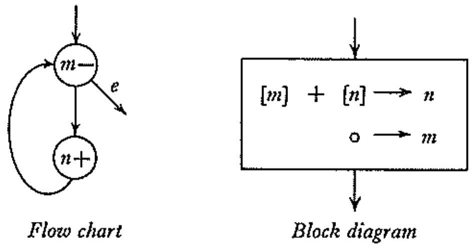
Figure 6-1. Elementary operations in abaci.

Flow charts can then be built up as in the following examples.

Example 6.1. Empty box n



Example 6.2. Empty box m into box n (where $m \neq n$)



(If $m = n$, the program halts – exits on the 'e' arrow – either at once or never, accordingly as the box is empty or not, initially.) When (as intended) $m \neq n$, the *effect* of the program is the same as that of carrying stones from box m to box n until box m is empty, but there is no way of instructing our cave man to do precisely that. What he *can* do is ($m-$) take stones out of box m, one at a time, and throw them away on the ground, and ($n+$) pick stones up off the ground, one at a time, and put them in box n. There is no assurance that the stones he puts in box n are the very ones he removed from box m, but we need no such assurance in order to be assured of the desired *effect* as described in the block diagram, *viz.*,

$[m]+[n] \rightarrow n$: the number of stones in box n after *this* move equals the sum of the numbers in m and in n before the move

and then

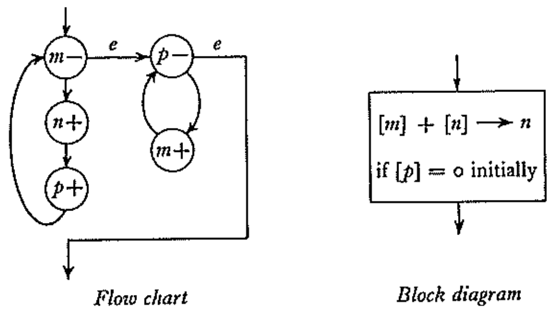
$o \rightarrow m$: the number of stones in box m after this move is o.

In general, the number on the right of an arrow identifies a box (a storage register, a pile) and the number on the left of that arrow is the number of stones that will be in that box (storage register, pile) after the move in question has been carried out.

In future, we shall assume (unless the contrary possibility is explicitly allowed) that when boxes are referred to by letters 'm', 'n', 'p', etc., distinct letters represent distinct boxes.

Example 6.3. Add box m into box n without loss from m

If the program is to have this effect, box p must be empty to begin with. In that case, p will be empty at the end, as well.



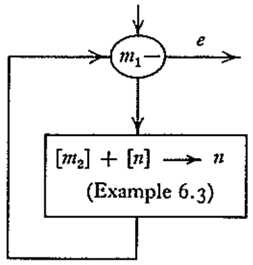
A more general description of the operation of this program, in which no special assumption is made about $[p]$, is this:

$$\begin{aligned}[m]+[n] &\rightarrow n,\\ [m]+[p] &\rightarrow m,\\ o &\rightarrow p.\end{aligned}$$

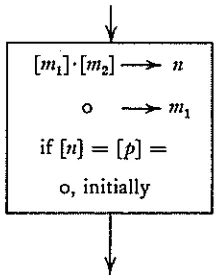
Here, as always, the vertical order represents a *sequence* of moves, from top to bottom, Thus, box p is emptied *after* the other two moves are made. (The order of the first two moves is arbitrary: the effect would be the same, if their order were reversed.)

Example 6.4. Multiplication

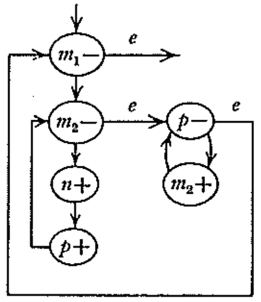
The numbers to be multiplied are in distinct boxes m_1 and m_2; two other boxes, n and p, are empty to begin with. The product appears in box n. Instead of constructing a flow chart *de novo*, we use the block diagram of Example 6.3 as shorthand for the flow chart of that example. It is then straightforward to draw the full flow chart, as at the right, where the 'm'



Abbreviated flow chart



Block diagram



Full flow chart

of Example 6.3 is changed to 'm_2'. The procedure is to repeatedly dump $[m_2]$ stones into box n, using box m_1 as a counter: we remove a stone from box m_1 before each dumping operation, so that when box m_1 is empty we have

$$\underbrace{[m_2]+[m_2]+\ldots+[m_2]}_{[m_1]}$$

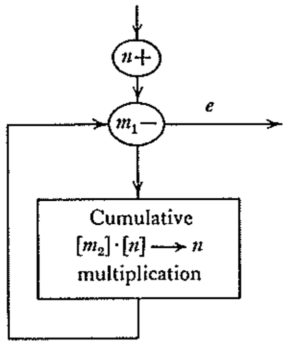
stones in box n.

Example 6.5. Exponentiation

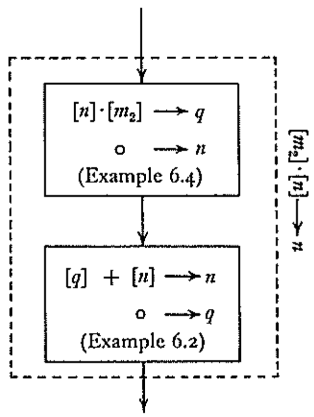
Just as multiplication is repeated addition, so exponentiation is repeated multiplication: to raise x to the power y, multiply together y xs,

$$x^y = \underbrace{x\cdot x\cdot \ \ldots \ \cdot x}_{y}.$$

The program is perfectly straightforward, once we arrange to modify the multiplication program of Example 6.4 so as to have $[m_2]\cdot[n] \to n$: see the abbreviated flow graph for exponentiation. The required modification of the routine of Example 6.4 is obtained (following Peter Tovey) by appending the routine of Example 6.2: see the diagram for cumulative



Abbreviated flow graph for exponentiation


$[n]\cdot[m_2] \longrightarrow q$
$0 \longrightarrow n$
(Example 6.4)
$[q] + [n] \longrightarrow n$
$0 \longrightarrow q$
(Example 6.2)
$[m_2]\cdot[n] \longrightarrow n$

Cumulative multiplication

multiplication. Provided boxes n, p and q are empty initially, the program for exponentiation has the effect,

$$[m_2]^{[m_1]} \to n,$$
$$0 \to m_1$$

in strict analogy to the program for multiplication in Example 6.4: see the block diagram there. Structurally, the abbreviated flow graphs for multiplication and exponentiation differ only in that for exponentiation, we need to put a single stone in box n at the beginning. If $[m_1] = 0$ we have $n = 1$ when the program terminates (as it will at once, without going through the multiplication routine). This corresponds to the convention that

$$x^0 = 1.$$

for any natural number x. But if $[m_1]$ is positive, $[n]$ will finally be a product of $[m_1]$ factors $[m_2]$, corresponding to repeated application of the rule

$$x^{y+1} = x\cdot x^y$$

which is implemented by means of cumulative multiplication, using box m_1 as a counter.

It should now be clear that the initial restriction to two elementary sorts of acts, $n+$ and $n-$, does not prevent us from computing fairly complex functions, e.g., all functions in the series that begins *sum*, *product*, *power*, ..., and where the $n+1$st member of the series is obtained by iterating the nth member. Apparently, the next member of the series, *super-exponentiation*, would be defined

$$\sup(x, y) = \underbrace{x^{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}}_{y}$$

where we have

$$\sup(x, 0) = 1, \quad \sup(x, y+1) = x^{\sup(x,\, y)}$$

so that the grouping in the general scheme is (e.g. for $y = 4$)

$$\sup(x, 4) = x^{x^{x^{x}}} = x^{\left(x^{\left(x^{x}\right)}\right)}$$

(The grouping does matter: $3^{(3^3)}$ is 3^{27}, which is the product of 27 threes, but $(3^3)^3$ is 27^3, which is the product of three twenty-sevens, or of only 9 threes.) Apparently the flow graph for a superexponentiator can be obtained from that of an exponentiator just as the flow graph of an exponentiator was obtained from that of a multiplier in Example 6.5.

Having proved that some very complex functions are abacus computable, we now present some evidence for Church's thesis: we prove that every abacus computable function is Turing computable. We do this by giving a method which, applied to the flow graph of any abacus program, yields the flow graph of a Turing machine which computes the same function that the abacus does. The format of the Turing machine computation will be the standard one described at the beginning of this chapter. But before describing our method for transforming abacus flow graphs into equivalent Turing machine flow graphs we must standardize certain features of abacus computations: we must know where to look, initially, for the arguments of the function which the abacus computes, and where to look, finally, for the value. The following conventions will do as well as any:

Format for abacus computation of $f(x_1, \ldots, x_r)$. Initially, the arguments are the numbers of stones in the first r boxes, and all other boxes are empty: $x_1 = [1], \ldots, x_r = [r], 0 = [r+1] = [r+2] = \ldots$. Finally, the value of the function is the number of stones in some previously specified box n (which may but need not be one of the first r). Thus, $f(x_1, \ldots, x_r) = [n]$ when the computation halts, i.e., when we come to an arrow in the flow graph which terminates in no node. If the computation never halts, $f_1(x_1, \ldots, x_r)$ is undefined.

Note that our computation routines for addition, multiplication, and exponentiation in Examples 6.3–6.5 are essentially in this form, with $r = 2$ in each case. They were formulated in a general way, so as to leave open the question of just which boxes are to contain the arguments and value, e.g. in the adder of Example 6.3 we only specified that the arguments are to be stored in distinct boxes numbered m and n, that the value will be found in box n, and that a third box, numbered p, and initially empty, will be used in the course of the computation. But now we must specify m, n and p, subject to the restriction that m and n shall be 1 and 2 (in either order) and p shall be some number greater than 2. Then we might settle on $n = 1$, $m = 2$, $p = 3$ to get a particular program for addition in the standard format (Figure 6-2).

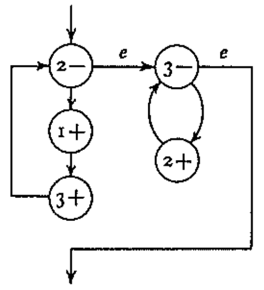


Figure 6-2. Addition (Example 6.3) in standard format.

The standard format associates a definite function from natural numbers to natural numbers with each abacus, once we specify the number r of arguments and the number n of the box in which the values will appear. Similarly the standard format for Turing machine computations associates a definite function from positive integers to positive integers with each Turing machine, once we specify the number r of arguments. Observe that once we have specified the graph of an abacus A in standard form, then for each register n that we might specify as holding the result of the computation there are infinitely many functions A_n^r which we have specified as computed by the abacus: one function for each possible number of arguments. Thus if A is determined by the simplest graph for addition, in Example 6.2, with $n = 1$ and $m = 2$, we have

$$A_1^2(x, y) = x + y$$

for all natural numbers x and y, but we also have the identity function $A_1^1(x) = x$ of one argument, and for three or more arguments we have

$A_1^r(x_1, x_2, \dots, x_r) = x_1 + x_2$. Indeed for $r = 0$ arguments we may think of A_1 as computing a 'function' of a sort, *viz.*, the number $A_1^0 = 0$ of 1s in box 1 when the computation halts, having been started with all boxes ('except for the first r') empty. Of course, the case is entirely parallel for Turing machines, each of which computes a function of r arguments in standard format for each $r = 0, 1, 2, \dots$.

Having settled on standard formats for the two kinds of computation, we can turn to the problem of designing a method for converting the flow graph of an abacus A_n with n designated as the box in which the values will appear into the graph of a Turing machine which computes the same functions: for each r, the Turing machine will compute the same function A_n^r of r arguments that the abacus computes. Our method will specify a Turing machine flow graph which is to replace each node of type $n+$ with its exiting arrow, in the abacus flow graph; a Turing machine flow graph which is to replace each node of type $n-$ with its two exiting arrows; and a Turing machine flow graph which, at the end, makes the machine erase all but the nth block of 1s on the tape and halt, scanning the leftmost of the remaining 1s.

It is important to be clear about the relationship between boxes of the abacus and corresponding parts of the Turing machine's tape.

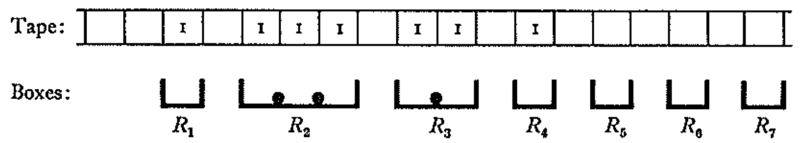
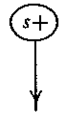


Figure 6-3. Correspondence between boxes and portions of the tape (● = one stone).

Example: in computing A_n^4 (0, 2, 1, 0), the initial tape and box configurations would be as shown in the accompanying figure. Boxes containing one or two or ... stones are represented by blocks of two or three or ... 1s on the tape (see Figure 6-3). Single blanks separate portions of the tape corresponding to successive boxes. Empty boxes are always represented by single squares, which may be blank (see $R_5, R_6, R_7, \dots$) or contain a 1 (see R_1 and R_4). The 1 is mandatory if there are any 1s further to the right on the tape, and is mandatory initially for empty argument boxes (R_1 through R_r) and the blank is mandatory initially for $R_{r+1}, R_{r+2}, \dots$. Then at any stage of the computation we can be sure that when in moving to the right/left we encounter two successive blanks, there are

no further 1s to be found anywhere to the right/left on the tape. The exact portion of the tape which represents a box will wax and wane with the contents of that box as the program progresses, and will shift to the right or left on the tape as stones are added to or removed from lower-numbered boxes.

The first step in our method for converting abacus flow graphs into equivalent Turing machine flow graphs can now be specified: replace each $s+$ node



by a copy of the $s+$ *flow graph* shown in Figure 6-4. The first $2s$ nodes of the $s+$ graph take the Turing machine from its standard position, scanning the leftmost 1 on the tape, to the blank immediately to the right of the sth block of 1s. (In seeking the sth block, the machine substitutes the 1-representation for the B-representation of any empty boxes.) On leaving node *sb*, the machine writes a 1, moves 1 square right, and does one thing or another (node x) depending on whether it is then scanning a blank or a 1. If it is scanning a blank there can be no more 1s to the right, and it therefore returns to standard position. But if it is scanning a 1 at that point it has more work to do before returning to standard position, for there are more blocks of 1s to be dealt with, to the right on the tape. These must be shifted one square right, by erasing the first 1 in each block and filling the blank to the block's right with a 1 – continuing this

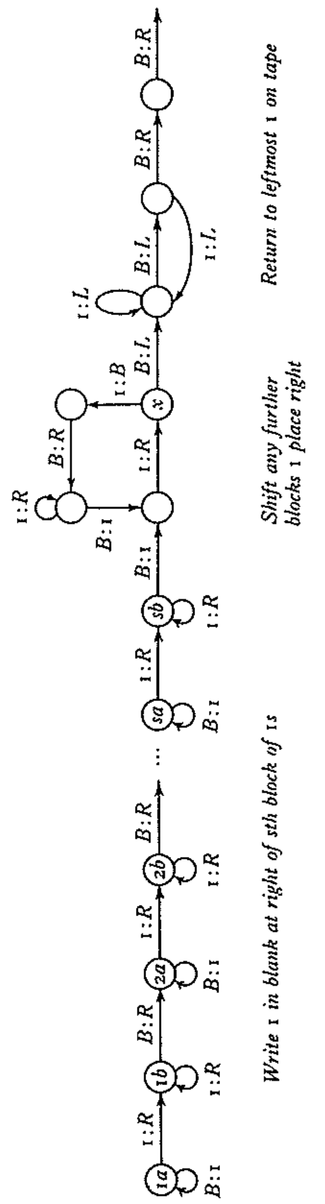

Return to leftmost 1 on tape
Shift any further blocks 1 place right
Write 1 in blank at right of sth block of 1s
1a
1b
2a
2b
sa
sb
x
B:1
1:R
B:R
B:L
1:L
1:B

Figure 6-4. The $s+$ flow graph.

routine until it finds a blank to the right of the last blank it has replaced by a 1. At that point there can be no further 1s to the right, and the machine returns to standard position.

Note that node 1*a* is needed in case the number r of arguments is 0: in case the 'function' A_n^0 which the abacus computes is a number. Note, too, that the first $s-1$ pairs of nodes (with their efferent arrows) are identical, while the last pair is different only in that the arrow from node *sb* to the right is labelled '$B:1$' instead of '$B:R$'.

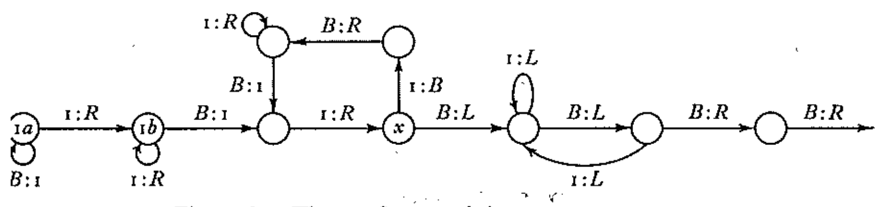
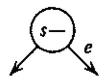


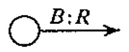
Figure 6-5. The $s+$ flow graph in case $s = 1$.

The second step in our method of converting abacus flow graphs into equivalent Turing machine flow graphs can now be specified: replace

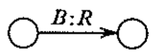


each $s-$ node by a copy of the $s-$ *flow graph* which you are invited to design in the pattern shown by Figure 6-6.

When the first and second steps of the method have been carried out, the abacus flow graph will have been converted into something which is not quite the flow graph of a Turing machine which computes the same functions that the abacus does. The graph will (probably) fall short in two respects. *First*, if the abacus ever halts, there must be one or more 'loose' arrows in the abacus graph: arrows which terminate in no node. Then as a glance at the $s+$ flow graph and the block diagram of the $s-$ flow graph will show, our Turing machine graph will have one or more loose arrows of form



which terminate in no node. But in a proper Turing machine flow graph, all arrows must terminate in nodes. We could repair this defect by simply drawing isolated nodes at the ends of loose arrows



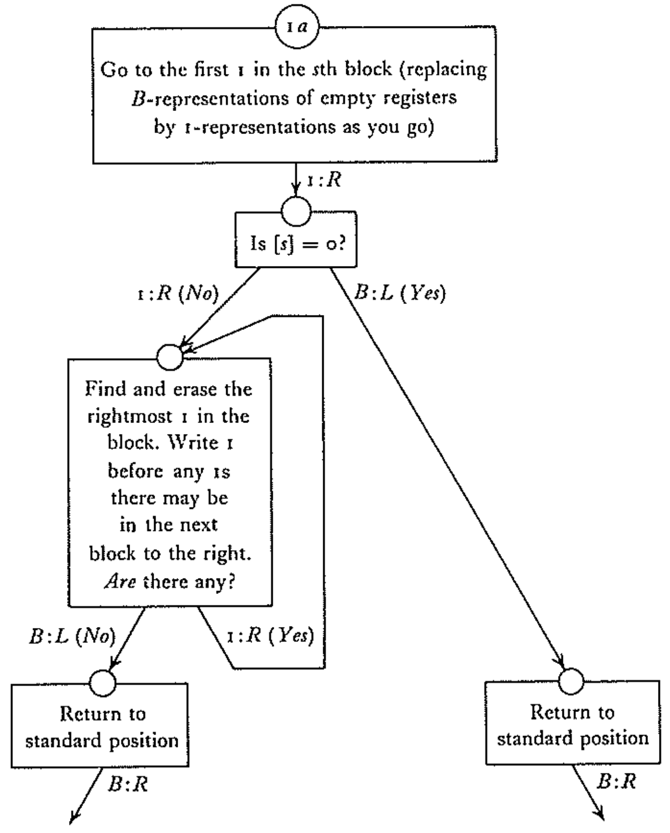

1a
Go to the first 1 in the sth block (replacing B-representations of empty registers by 1-representations as you go)
1:R
Is [s] = 0?
1:R (No)
B:L (Yes)
Find and erase the rightmost 1 in the block. Write 1 before any 1s there may be in the next block to the right. Are there any?
B:L (No)
1:R (Yes)
Return to standard position
B:R
Return to standard position
B:R

Figure 6-6. Block diagram for s-flow graph.

but a *second* shortcoming would remain: in computing $A_n^r(x_1, \ldots, x_r)$ the Turing machine would halt scanning the leftmost 1 on the tape, *but the value of the function would be represented by the nth block of* 1s *on the tape*. Even if $n = 1$, we cannot depend on there being no 1s on the tape after the first block, so our method requires one more step.

The third step: after completing the first two steps, redraw all loose arrows so that they terminate in the input node of a *mop-up graph* which you are invited to design. This graph makes the machine (which will be scanning the leftmost 1 on the tape at the beginning of this routine) erase all but the first block of 1s, if $n = 1$, and halt scanning the leftmost of the remaining 1s. But if $n \neq 1$, it erases everything on the tape except for both the leftmost 1 on the tape and the nth block, repositions all 1s but the rightmost in the nth block immediately to the right of the leftmost 1, erases the rightmost 1, and then halts, scanning the leftmost 1. In both cases the effect is to place the leftmost 1 in the block representing the value just where the leftmost 1 was initially.

When you have carried out the details of the second and third of these moves, you will have established that all abacus computable functions are Turing computable. This is rather persuasive evidence for Church's thesis, for the abacus computable functions are the ones that modern digital computing machines would be able to compute if restrictions on the number and size of random-access storage registers could be removed. Of course it remains conceivable that some day, essentially different sorts of computing machines will be invented: machines which, suitably idealized, would be able to compute functions which are not abacus computable and not Turing computable. (Without knowing how such machines would operate we cannot know what idealizations would be suitable. But the idealizations would presumably be analogous to these concerning the storage capacities of Turing machines and abaci: a Turing machine always has more tape than it has used so far, and an abacus has room in its registers for indefinitely many, indefinitely large numbers.) Church's thesis has not been proved, mathematically. Rather, it has been supported by evidence, much as an empirical scientific theory might be. More such evidence will be presented in the following two chapters.

Exercises

6.1 Draw the $s-$ flow graph (second step of the method for converting abacus flow graphs to equivalent Turing machine flow graphs).

6.2 Draw the mop-up graph (third step of the method).

6.3 *The abacus halting problem.* Let $h(x,y) = 0$ or 1 accordingly as abacus number x in some list of all abaci does or does not eventually halt after being started with y stones in R_1 and all other registers empty. Prove (without using the analogue of Church's thesis for abaci) that h is not abacus computable. *Hint*: ape the argument at the end of Chapter 5.

6.4 *The busy abacus problem.* If an abacus never halts after being started with all registers empty, its productivity is said to be 0, and otherwise its productivity is said to be the number of stones in R_1 when it halts. Let $p(n)$ = the productivity of the most productive n node abaci. Prove that p is not abacus computable.

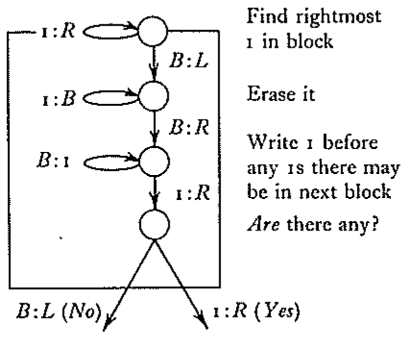
Solutions

6.1 The top block of the block diagram for the $s-$ flow graph contains a graph identical with the material from node $1a$ to node sa (inclusive) of the $s+$ flow graph. The arrow labelled '$1:R$' from the bottom of this block corresponds to the one that goes right from node sa in the $s+$ flow graph.

The '*Is* $[s] = 0$?' box contains nothing but the shafts of the two emergent arrows: they originate in the node shown at the top of that block.

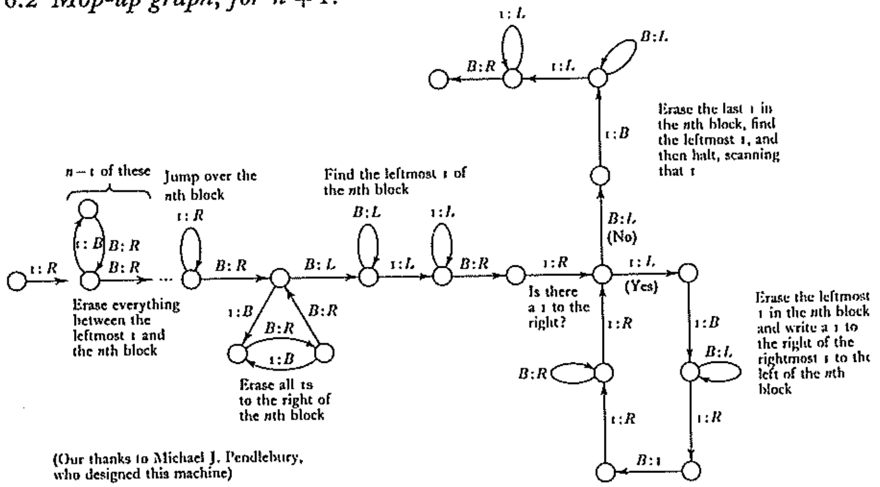
The 'Return to standard position' blocks contain replicas of the material to the right of node x in the $s+$ graph: the '$B:L$' arrows entering those boxes correspond to the '$B:L$' arrow from node x.

The only novelty is in the remaining block: 'Find and erase the...' That block contains the graph shown below.



1:R
B:L
1:B
B:R
B:1
1:R
Find rightmost 1 in block
Erase it
Write 1 before any 1s there may be in next block
Are there any?
B:L (No)
1:R (Yes)

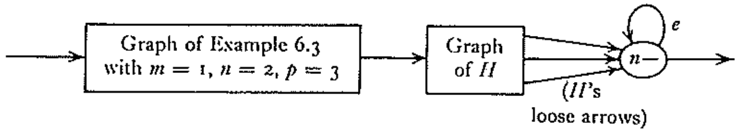
6.2 *Mop-up graph, for* $n \neq 1$.



1:R
n − 1 of these
1:B
B:R
B:R
Erase everything between the leftmost 1 and the nth block
Jump over the nth block
1:R
B:R
1:B
B:R
B:R
1:B
Erase all 1s to the right of the nth block
B:L
Find the leftmost 1 of the nth block
B:L
1:L
1:L
B:R
1:R
Is there a 1 to the right?
B:L (No)
1:B
B:L
1:L
B:R
1:L
Erase the last 1 in the nth block, find the leftmost 1, and then halt, scanning that 1
1:L (Yes)
1:B
B:L
1:R
B:1
1:R
B:R
1:R
Erase the leftmost 1 in the nth block and write a 1 to the right of the rightmost 1 to the left of the nth block

(Our thanks to Michael J. Pendlebury, who designed this machine)

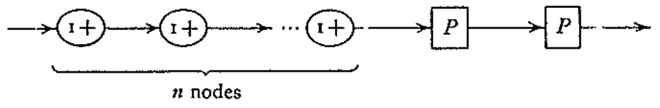
6.3 Suppose that $h = H_n^2$ for some abacus H, i.e., suppose that H computes h and stores the value of the function in R_n. Then for any x, the abacus shown in the accompanying graph eventually halts after being started with x stones in R_1 if and only if abacus number x never halts after being started with x stones in R_1 – where in each case, all other registers are empty initially. (Note that with the last node deleted, the abacus in question computes the function $h(x, x)$.) If there is such an



Graph of abacus number m

abacus as H then there is such an abacus as the one we have graphed, and it must appear in our list of all abaci: it is abacus number m, say. But if we set $x = m$ we see that abacus m stops if and only if it does not, after being started with m stones in R_1 and all other registers empty. Then there is no such abacus: and no abacus computes h: the abacus halting problem is unsolvable.

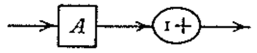
6.4 Suppose that some abacus P computes p, storing the values in register 1. (If some other abacus computes p but stores the values in register n, we can convert into an abacus of the sort we want by adding three more modes: one to empty R_1 and then two as in Example 6.2 to empty R_n into R_1.) If P has k nodes then the machine with the accompanying graph has $n + 2k$ nodes and has productivity $p(p(n))$. Then if P exists



we have

$$p(n+2k) \geqslant p(p(n))$$

for all n. It is easily proved that $p(n)$ is an increasing function of n, for if A is one of the most productive abaci with n nodes, we can obtain a more productive $n+1$ node abacus by appending a single $1+$ node.



Then we have

$$n+2k \geqslant p(n) \qquad (*)$$

for all n, if P exists. With n nodes, we can put n stones in R_1, and with an additional 5 nodes we can copy those n stones in R_2 by the routine of Example 6.3. Now by appending another replica of the routine of Example 6.3, we obtain an $n+10$ node abacus which has productivity $2n$. Thus, for all n we have

$$p(n+10) \geqslant 2n$$

whether or not P exists. Putting '$n+10$' for 'n' in (*) and combining with the last inequality we have

$$n+10+2k \geqslant 2n$$

for all n if P exists. Then if P exists we have

$$10+2k \geqslant n$$

for *all* n. Putting '$11+2k$' for 'n' we then have

$$0 \geqslant 1$$

if P exists. Then there is no such abacus as P: the busy abacus problem is unsolvable.

Note. The treatment of abaci in this and the following chapters is drawn from Joachim Lambek 'How to program an infinite abacus', *Canadian Mathematical Bulletin* 4 (1961), 295–302, with a correction noted in 5 (1962), 297. Lambek's abaci are simplifications of machines studied by Z. A. Melzak 'An informal arithmetical approach to computability and computation', *Canadian Mathematical Bulletin* 4 (1961), 279–293.

7 Recursive functions are abacus computable

The functions in the sequence sum, product, power, ... noted in Chapter 6 belong to the class of *primitive recursive functions*. We now give a precise definition of that class and of the more extensive class R of *recursive* functions. We shall then prove that all functions in R belong to the class A of abacus computable functions, so that we have

$$R \underset{7}{\subseteq} A \underset{6}{\subseteq} T,$$

where the subscripts identify the chapters in which the inclusions are proved. This will provide further support for Church's thesis, for as we shall see, the class of recursive functions is very broad indeed – so broad as to make it plausible that all functions computable in any intuitive sense are recursive.

To fix ideas, consider the third function in the foregoing sequence – the function exp which, applied to an argument pair (x,y), yields the yth power of x,

$$\exp(x,y) = x^y.$$

Here we have

$$x^0 = 1,$$
$$x^1 = x,$$
$$x^2 = x \cdot x,$$
$$\vdots$$
$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y \text{ times}},$$
$$x^{y+1} = \underbrace{x \cdot x \cdot \ldots \cdot x \cdot x}_{y+1 \text{ times}} = x \cdot x^y.$$

The first and last of these equations,

$$x^0 = 1, \quad x^{y+1} = x \cdot x^y,$$

suffice for computation of all values of the function.

Example. Computation of 5^3

$$5^3 = 5^{2+1} = 5 \cdot 5^2 \text{ (setting } x = 5 \text{ and } y = 2 \text{ in the last equation)},$$
$$5^2 = 5^{1+1} = 5 \cdot 5^1 \text{ (setting } x = 5 \text{ and } y = 1 \text{ in the last equation)},$$
$$5^1 = 5^{0+1} = 5 \cdot 5^0 \text{ (setting } x = 5 \text{ and } y = 0 \text{ in the last equation)},$$
$$5^0 = 1 \qquad \text{(setting } x = 5 \text{ in the first equation)}.$$

Combining, we have

$$5^3 = 5 \cdot 5^2 = 5 \cdot 5 \cdot 5^1 = 5 \cdot 5 \cdot 5 \cdot 5^0 = 5 \cdot 5 \cdot 5 \cdot 1.$$

We have thus reduced the problem of computing a power to that of computing a product. *That* can be reduced in a similar way to the problem of computing sums, and that in turn can be reduced to a matter of repeatedly adding 1. The overall result of these computations will be the information that

$$5^3 = \underbrace{1 + 1 + \ldots + 1}_{125 \text{ `1's}}.$$

Here we have 5^3 expressed in a form of monadic notation, where the numbers zero and one are denoted by the signs '0' and '1', and where the sign for the number $n+1$ thereafter is obtained by appending the sign '+1' to the sign for the number n.

To make it clear that we are not smuggling the operation of summation into our very notation for numbers, we shall use the symbol 's' for the successor function, and denote the successive natural numbers as follows:

$$0, \quad s(0), \quad s(s(0)), \ldots$$

Thus, our official monadic notation explicitly incorporates the sign 's' for the one-place function which, applied to any natural number argument, assumes the next larger natural number as value. It is then clear that e.g. the pair of equations

$$x + 0 = x, \quad x + s(y) = s(x+y),$$

conveys information which was not already implicit in the notation for numbers.

Example

Computation of 2+3, i.e., $s(s(o))+s(s(s(o)))$.

$$\underbrace{s(s(o))}_{x}+\underbrace{s(s(s(o)))}_{y} = s(\underbrace{s(s(o))}_{x}+\underbrace{s(s(o))}_{y}) \quad \text{(second equation)},$$

$$\underbrace{s(s(o))}_{x}+\underbrace{s(s(o))}_{y} = s(\underbrace{s(s(o))}_{x}+\underbrace{s(o)}_{y}) \quad \text{(second equation)},$$

$$\underbrace{s(s(o))}_{x}+\underbrace{s(o)}_{y} = s(\underbrace{s(s(o))}_{x}+\underset{y}{o}) \quad \text{(second equation)},$$

$$\underbrace{s(s(o))}_{x}+o = \underbrace{s(s(o))}_{x} \quad \text{(first equation)}.$$

Combining, we have

$$\begin{aligned} s(s(o))+s(s(s(o))) &= s(s(s(o))+s(s(o))) \\ &= s(s(s(s(o))+s(o))) \\ &= s(s(s(s(s(o))+o))) \\ &= s(s(s(s(s(o))))). \end{aligned}$$

Having seen how the thing goes in uncompromisingly mechanical terms, we can summarize by writing

$$so, \quad sso, \quad ssso, \ldots$$

as shorthand for

$$s(o), \quad s(s(o)), \quad s(s(s(o))), \ldots$$

and so aiding the human eye by displaying only the main pairs of parentheses. With these abbreviations, the equations can be written

$$x+o = x, \quad x+sy = s(x+y),$$

and the computation proceeds as follows:

$$\begin{aligned} sso+ssso &= s(sso+sso), \\ sso+sso &= s(sso+so), \\ sso+so &= s(sso+o), \\ sso+o &= sso. \end{aligned}$$

Combining we have

$$\begin{aligned} sso+ssso &= s(sso+sso) \\ &= s(s(sso+so)) \\ &= s(s(s(sso+o))) \\ &= s(s(s(sso))) = sssss o, \end{aligned}$$

where the last step is a matter of mentally rewriting '*sso*' as '$s(s(o))$' and and then abbreviating '$s(s(s(s(s(o)))))$' as '*ssssso*'.

Another common notation for the successor function is an accent, written *after* the argument. Thus, suppressing parentheses, the successive natural numbers would be denoted

$$o, \quad o', \quad o'', \ldots$$

in this notation, and the equations for summation would be

$$x+o = x, \quad x+y' = (x+y)'.$$

The point of using the fussy notation in which we write '$s(s(s(o)))$' instead of '3' etc. is that we are applying to a mode of computation with which we are very familiar the same sort of minute analysis which we applied to Turing and abacus computations. Clarity about such minutiae requires us to be wary of familiar shortcuts until such time as we have established their validity within the present framework. To achieve such clarity, we must be quite explicit about the structure of that framework. There are various equivalent ways of doing that. Therefore, any one of those ways will have certain features which seem unnatural because arbitrary. Choice of the '*s*' notation is a trivial first step toward explicitness. Full explication requires us to make several further choices, which we now do in what seems the simplest way and is certainly the commonest way.

We shall define the class of primitive recursive functions by specifying an initial stock of functions as belonging to the class, and then specifying two sorts of operations which, applied to members of the class, yield functions which are to be understood as belonging to the class as well. To complete the definition we stipulate that nothing belongs to the class unless it is either in the initial stock or is obtainable from functions in that stock by a finite sequence of applications of operations of the two sorts.

The initial stock of functions consists of the *zero function*, the *successor function*, and the various *identity functions*.

The zero function, z, is a function of one argument. To each natural number as argument, it assigns the same value, *viz.*, the natural number zero:

$$z(x) = 0.$$

Thus, $z(0) = z(1) = z(2) = \ldots = 0$

or, in our fussy notation,

$$z(0) = z(s(0)) = z(s(s(0))) = \ldots = 0.$$

We have already encountered the successor function, s, of one argument, which assigns to each natural number as argument its successor as value. Thus, $s(0) = 1$, $s(1) = 2$, $s(2) = 3, \ldots$. Translated into our fussy notation, these equations become vacuous ('$s(0) = s(0)$' etc.) since the properties of s are packed into that notation.

The identity function of one argument, id, assigns to each natural number as argument that same number as value:

$$\mathrm{id}\,(x) = x.$$

There are two identity functions of two arguments: id^2_1 and id^2_2. For any pair of natural numbers as arguments, these pick out the first and second, respectively, as values:

$$\mathrm{id}^2_1(x,y) = x, \quad \mathrm{id}^2_2(x,y) = y.$$

In general, for each positive integer n, there are n identity functions of n arguments, which pick out as values the first, the second, ..., the nth of the arguments:

$$\mathrm{id}^n_i(x_1, \ldots, x_i, \ldots, x_n) = x_i.$$

An obvious alternative notation for id, the identity function of one argument, is 'id^1_1'. Identity functions are often called *projection functions.* (In terms of analytic geometry, $\mathrm{id}^2_1(x,y)$ and $\mathrm{id}^2_2(x,y)$ are the projections x and y of the point (x,y) on the X-axis and the Y-axis respectively.)

From this initial stock of *basic functions* – z, s, and the identity functions in the infinite list id, id^2_1, id^2_2, $\mathrm{id}^3_1, \ldots$ – we may form new primitive recursive functions by operations of two sorts: *composition* and *primitive recursion.* The *primitive recursive functions* are simply the basic functions together with those functions obtainable from them by finite numbers of applications of operations of the two sorts, in any order.

The first sort of operation, composition (= *substitution*), is familiar and straightforward. If f is a function of m arguments and each of $g_1, \ldots, g_m$ is a function of n arguments, then the function h where

$$\boxed{h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))} \quad \text{(Cn)}$$

is the function obtained from $f, g_1, \ldots, g_m$ by *composition.* One might indicate this in shorthand:

$$h = \mathrm{Cn}[f, g_1, \ldots, g_m].$$

Example

The function h of three arguments which always takes the successor of its third argument as value is obtained from the pair s, id^3_3 by composition: $h = \mathrm{Cn}\,[s, \mathrm{id}^3_3]$. In terms of the format (Cn) we have $n = 3$ and $m = 1$, i.e., the special case of the format with which we are dealing is

$$h(x_1, x_2, x_3) = f(g_1(x_1, x_2, x_3)),$$

where $f = s$ and $g_1 = \mathrm{id}^3_3$.

Another example. Cn [*s*, *s*]

This is a function of one argument, x, of which the value is always $x+2$, i.e., $s(s(x))$. In terms of the boxed format (Cn) we have $m = n = 1$, so that we are dealing with the special case

$$h(x_1) = f(g_1(x_1)),$$

where $f = g_1 = s$.

Of course, not all of the functions used in composition need be basic.

Example. $h(x) = x+3$

Here, h is obtainable by composition of the functions s, $\mathrm{Cn}\,[s,s]$ where the second is not basic, but *is* obtainable by composition of basic functions. Thus, h is primitive recursive: $h = \mathrm{Cn}\,[s, \mathrm{Cn}[s,s]]$.

Observe that such notations as '$\mathrm{Cn}\,[s, \mathrm{id}^3_3]$' are genuine function symbols: they belong to the same grammatical category as 'h'. Then just as we can write '$h(x_1, x_2, x_3) = s(x_3)$', so can we write

$$\mathrm{Cn}\,[s, \mathrm{id}^3_3]\,(x_1, x_2, x_3) = s(x_3).$$

The foregoing equation is clearly true: the function $\mathrm{Cn}\,[s, \mathrm{id}^3_3]$ does apply to triples of natural numbers, and its value is always the successor of its third argument.

The second sort of operation for generating new primitive recursive functions is called 'primitive recursion'. The format is this:

$$\boxed{h(x, 0) = f(x), \quad h(x, s(y)) = g(x, y, h(x, y))} \quad \text{(Pr)}$$

Where the boxed equations hold, h is said to be definable by *primitive recursion* from the functions f, g. In shorthand:

$$h = \mathrm{Pr}[f, g].$$

The definitions of sum, product, and power above are approximately in this format. By fussing over them, we can put them exactly into the format (Pr).

Example. $x+0 = x, x+s(y) = s(x+y)$

To begin, let us replace the operation symbol '+' by the function symbol 'sum':

$$\mathrm{sum}(x, 0) = x, \quad \mathrm{sum}(x, s(y)) = s(\mathrm{sum}(x, y)).$$

To put these into the boxed format (Pr) we must find functions f and g for which we have

$$f(x) = x, \quad g(x, y, -) = s(-)$$

for all natural numbers x, y, and $-$. Such functions lie ready to hand: $f = \mathrm{id}$, and $g = \mathrm{Cn}[s, \mathrm{id}_3^3]$. Further, since both of these are primitive recursive (being basic or obtained from basic functions by composition), *sum* is seen to be primitive recursive as well. In the boxed format we have

$$\mathrm{sum}(x, 0) = \mathrm{id}(x), \quad \mathrm{sum}(x, s(y)) = \mathrm{Cn}[s, \mathrm{id}_3^3](x, y, \mathrm{sum}(x, y))$$

and in shorthand we have

$$\mathrm{sum} = \mathrm{Pr}[\mathrm{id}, \mathrm{Cn}[s, \mathrm{id}_3^3]].$$

Another example. prod = Pr [z, Cn [sum, id_1^3, id_3^3]]

To verify this claim we relate it to the boxed formats (Cn) and (Pr). In terms of (Pr) the claim is that the equations

$$\mathrm{prod}(x, 0) = z(x), \quad \mathrm{prod}(x, s(y)) = g(x, y, \mathrm{prod}(x, y))$$

hold for all natural numbers x and y, where (setting $h = g$, $f = \mathrm{sum}$, $g_1 = \mathrm{id}_1^3$, and $g_2 = \mathrm{id}_3^3$ in the boxed (Cn) format) we have

$$\begin{aligned} g(x_1, x_2, x_3) &= \mathrm{Cn}[\mathrm{sum}, \mathrm{id}_1^3, \mathrm{id}_3^3](x_1, x_2, x_3) \\ &= \mathrm{sum}(\mathrm{id}_1^3(x_1, x_2, x_3), \mathrm{id}_3^3(x_1, x_2, x_3)) \\ &= x_1 + x_3 \end{aligned}$$

for all natural numbers x_1, x_2, x_3. Overall, then, the claim is that the equations

$$\mathrm{prod}(x, 0) = 0, \quad \mathrm{prod}(x, s(y)) = x + \mathrm{prod}(x, y)$$

hold for all natural numbers x and y, which is true:

$$x \cdot 0 = 0, \quad x \cdot (y+1) = x + x \cdot y.$$

Exercise 7.1

Verify the truth of the description

$$\sup = \mathrm{Pr}[\mathrm{Cn}[s, z], \mathrm{Cn}[\exp, \mathrm{id}_1^3, \mathrm{id}_3^3]]$$

of the superexponentiation function which was defined in Chapter 6.

Exercise 7.2

Write out a true description exp = ... of the exponentiation function which shows that exponentiation (and therefore sup as well) is primitive recursive.

Our rigid format for primitive recursive serves for functions of two variables such as sum and prod, but we shall sometimes wish to use such a scheme to define functions of a single variable, and functions of more than two variables. Where there are three or more variables $x_1, \ldots, x_n, y$ instead of the two variables x, y which appear in (Pr), the modification is achieved by viewing each of the five occurrences of x in the boxed format as shorthand for $x_1, \ldots, x_n$. Thus, with $n = 2$, the format is

$$\begin{aligned} h(x_1, x_2, 0) &= f(x_1, x_2) \\ h(x_1, x_2, s(y)) &= g(x_1, x_2, y, h(x_1, x_2, y)). \end{aligned}$$

If there is only one variable, y, we take the format to be

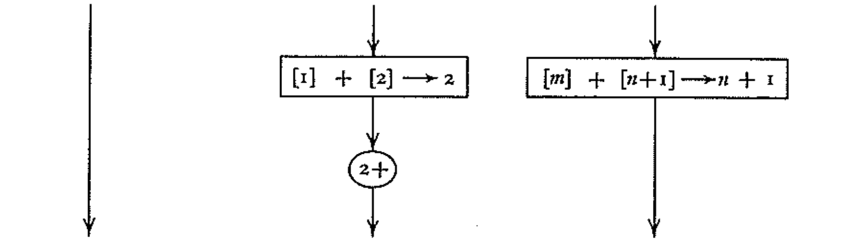
$$\begin{aligned} &h(0) = 0 \quad \text{or} \quad s(0) \quad \text{or} \quad s(s(0)) \text{ or } \ldots \\ &h(s(y)) = g(y, h(y)). \end{aligned}$$

Here the *x*s have disappeared altogether, and the 'function' *f* of 0 arguments is a constant: 0 or 1 or 2 or ...

Exercise 7.3

The *factorial* of *y* is the product of all the positive integers up to and including *y* (but is taken to be 1 when *y* is 0). Define *fac* by primitive recursion.

We now verify that all primitive recursive functions are abacus computable. To compute a function of *n* arguments on an abacus, we specify *n* registers or boxes in which the arguments are to be stored initially (represented by piles of rocks) and we specify a box in which the value of the function is to appear (as a pile of rocks) at the end of the computation. To facilitate comparison with computations by Turing machines in standard form, let us assume that the arguments appear in boxes 1, ..., *n*, and that the value is to appear in box $n+1$; and let us suppose that initially, all boxes except for the first *n* are empty. Now the programs of Figure 7-1 compute the basic functions.



The function *z* is computed by a vacuous program: box 2 is empty anyway!

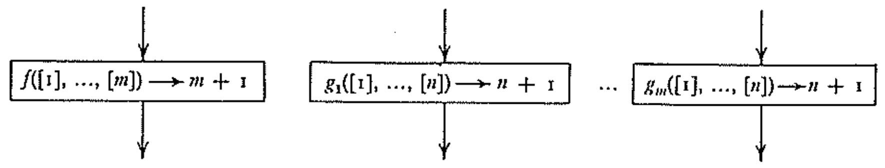
s is computed by modifying the program of Example 6.3. Initially and finally, $[1] = x$; initially, $[2] = 0$; finally, $[2] = s(x)$.

id^n_m is computed by the program of Example 6.3. Initially and finally, $[1] = x_1, ..., [n] = x_n$; initially, $[n+1] = 0$; finally, $[n+1] = x_m$.

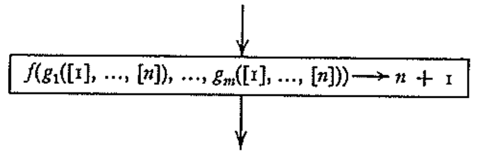
Figure 7-1. Abacus computation of the basic functions.

It only remains to show how, once we are given programs which compute functions $f, g_1, ..., g_m$, we can arrange them into a program which computes their composition, $h = \text{Cn}[f, g_1, ..., g_m]$, and how, given programs which compute functions *f* and *g*, we can arrange them into a program which computes the function *h* obtained from them by primitive recursion, $h = \text{Pr}[f, g]$.

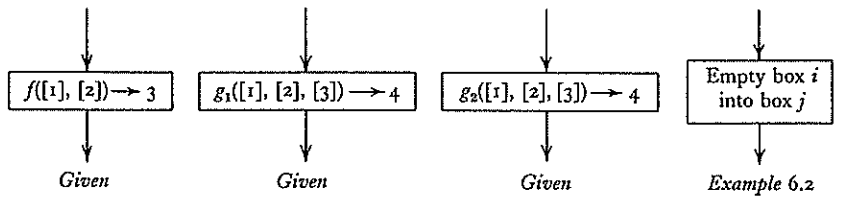
Composition. Suppose that we are given $m+1$ programs



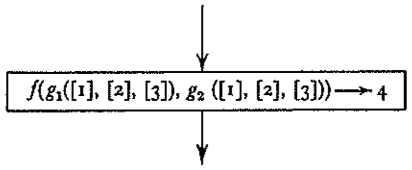
from which we seek to construct a single program



The thing is perfectly straightforward: it is a matter of shuttling the results of the subcomputations around so as to be in the right boxes at the right times. Here is how it is done for the case $m = 2$, $n = 3$; it will be obvious from that, how to treat any other case. First, we identify $m+n = 5$ registers, none of which are used in any of the $m+1$ given programs. Let us call these registers p_1, p_2, q_1, q_2, and q_3. They will be used for temporary storage. Recall that in the single program which we want to construct, the $n = 3$ arguments are stored initially in boxes 1, 2, and 3; all other boxes are empty, initially; and at the end, we want the *n* arguments back in boxes 1, 2, 3, and want the value, $f(g_1([1], [2], [3]), g_2([1], [2], [3]))$, in box number $n+1 = 4$. To arrange that, all we need are the three given programs and the program of Example 6.2:



We shall simply compute $g_1([1], [2], [3])$ and store the result in box p_1 (which figures in none of the given programs, remember): then compute g_2 of the arguments in boxes 1, 2, and 3, and store the result in box p_2; then we shall store the arguments in boxes 1, 2, and 3 in boxes q_1, q_2, and q_3, and empty boxes 1 through 4; then get the results of the first two computations out of boxes p_1 and p_2 (which we empty), and put them in boxes 1 and 2, so that we can compute $f([1],[2]) = f(g_1(\text{original contents of } 1, 2, 3), g_2(\text{original contents of } 1, 2, 3))$ and find the result in box 3, as provided in the given program for f. Finally, we tidy up; move the result of the overall computation from box 3 to box $n+1 = 4$, emptying box 3 in the process; empty boxes 1 and 2; and refill the first three boxes with the $n = 3$ arguments of the overall computation, which were stored in boxes q_1, q_2, and q_3. Now everything is as it should be, and we have a flow chart of which the block diagram is this:



The structure of the flow chart is shown in Figure 7-2.

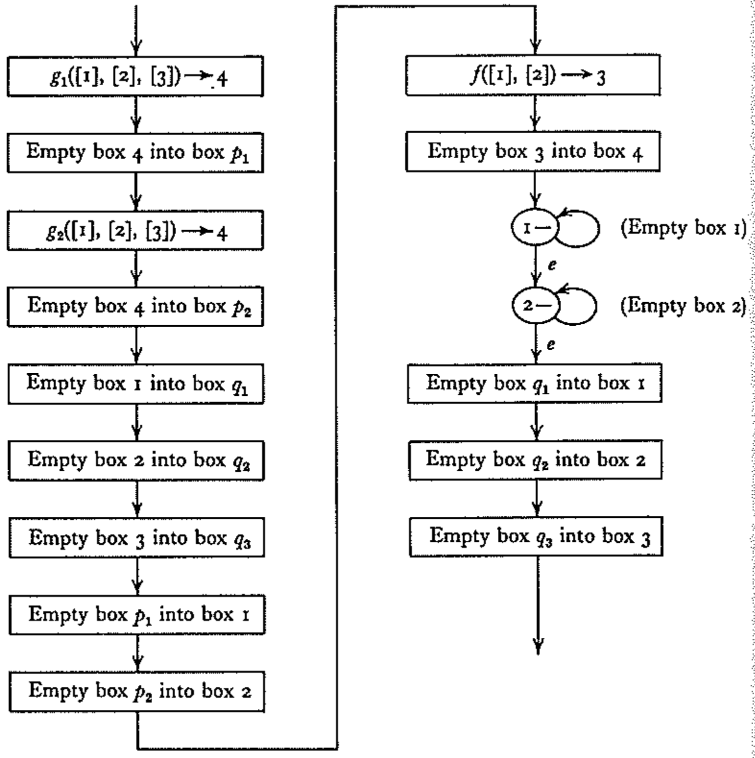
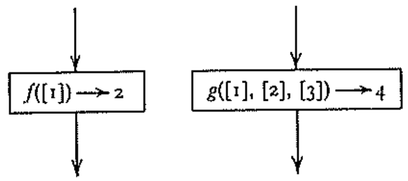
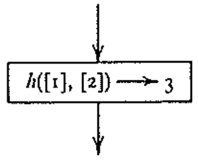


Figure 7-2. Composition of programs in case $m = 2$, $n = 3$.

It only remains to prove that if we have programs to compute the functions f and g, we can always design a program to compute the function h which is obtained from them by primitive recursion. We consider the case where h is a function of two arguments, as in the boxed format. The cases in which h has only one argument, or has more than two arguments, are treated similarly. Then the given flow graphs have these block diagrams:



Using these, we want to design a flow graph with this block diagram:



The thing is easily done, as in Figure 7-3. Initially $[1] = x$, $[2] = y$, and $[3] = [4] = \ldots = 0$. We use register number p (which is not used in the f or g programs) as a counter: we put y into it at the beginning, and after each stage of the computation we see whether $[p] = 0$. If so, the computation is essentially finished; if not, we subtract 1 from $[p]$ and go through another stage. In the first three steps (Figure 7-3) we calculate $f(x)$ and then see whether y was 0. If so, the first of the boxed equations is operative: $h(x, y) = h(x, 0) = f(x)$, and the computation is finished, with the result in box 3, as required, If not, we successively compute $h(x, 1)$, $h(x, 2)\ldots$ (see the cycle in Figure 7-3) until the counter (box p) is empty. At that point the computation is finished, with $h(x, y)$ in box 3, as required. Given programs to compute functions f and g of 1 and 3 arguments respectively, this program computes the function $h = \mathrm{Pr}[f, g]$ of 2 arguments.

We have now proved that all primitive recursive functions are abacus computable. But there are still more functions that abaci can compute: not only the primitive recursive functions, but all (total and partial) *recursive* functions. These consist of the basic functions ($z, s, \mathrm{id}, \mathrm{id}_1^2, \ldots$) together with all functions obtainable from them by finite numbers of applica-

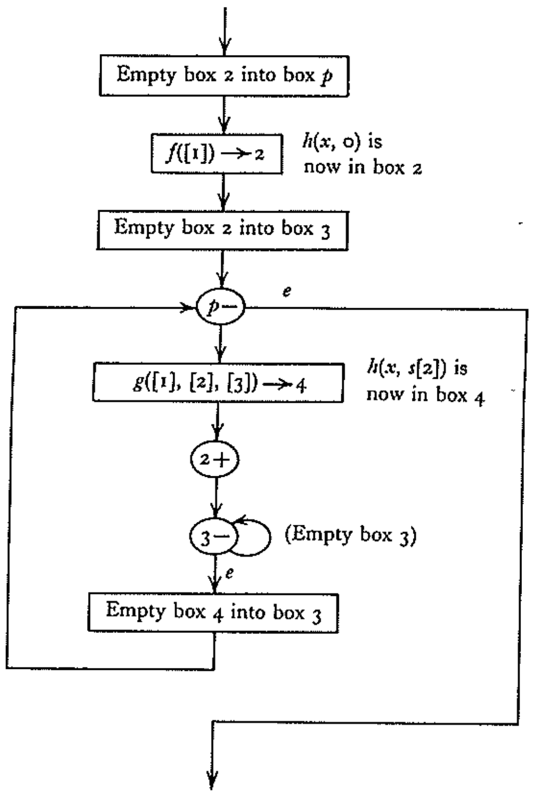


Figure 7-3. Primitive recursion.

tions of *three* sorts of operations: composition, primitive recursion, and *minimization*. Applied to a *total* function f of $n+1$ arguments, the operation of minimization yields a function h of n arguments as follows:

$$h(x_1, \dots, x_n) = \begin{cases} \text{the smallest } y \text{ for which } f(x_1, \dots, x_n, y) = 0, \text{ if any,} \\ \text{undefined if } f(x_1, \dots, x_n, y) = 0 \text{ for no } y. \end{cases}$$

We shall write '$h = \mathrm{Mn}[f]$' to indicate that h is obtainable from f by minimization.

Example

Mn [sum] is the partial function of one argument which takes the value 0 when its argument is 0 and is otherwise undefined, for $x+y = 0$ if and only if $x = y = 0$. (Remember that we are dealing with *natural* numbers: the xs and y cannot assume negative values, here.)

Another example

Mn [prod] = z, for $x \cdot y = 0$ whenever $y = 0$, regardless of the value of x. Then for every x, Mn [prod] $(x) = 0$.

In general – where the function f need not be total – we define Mn [f] as follows.

$$\mathrm{Mn}[f](x_1, \dots, x_n) = \begin{cases} y, \text{ in case } f(x_1, \dots, x_n, y) = 0 \text{ while } f(x_1, \dots, x_n, t) \text{ is} \\ \quad \text{defined and positive for all } t \text{ less than } y, \\ \text{undefined, in case there is no such } y. \end{cases}$$

Where the function f is total, this definition agrees with that given earlier; but if the earlier definition were applied to partial functions it would sometimes give different results from the present one.

Example

$f(0)$ is undefined and $f(1) = 0$. Then Mn[f] is undefined (according to the definition which we have adopted), but would be defined (and equal to 1) if the earlier definition had been adopted without the restriction to total functions.

Observe that all primitive recursive functions are total: the basic functions are defined for all natural number arguments, and the operations of composition and primitive recursion always yield total functions when applied to total functions. But as we have seen in the case of Mn [sum], the operation of minimization can yield partial functions even when applied to total functions.

Exception. Mn [f], where f is *regular*. A function f of $n+1$ arguments is said to be regular when, for every n-tuple of natural numbers $x_1, \dots, x_n$, there is a natural number y for which we have $f(x_1, \dots, x_n, y) = 0$.

To establish that all recursive functions are abacus computable, it only remains to show how, given an abacus program which computes a function f, we can construct an abacus program which computes the function Mn [f], as in Figure 7-4, where we set $n = 1$ for definiteness, so that the function f has two arguments, and Mn [f] has one. The program of Figure 7-4 computes Mn [f] (x), where x is the number of stones in box 1 initially. Initially box 2 is empty, so that if $f(x, 0) = 0$, the program will halt with the correct answer, Mn[f]$(x) = 0$, in box 2. (Box 3 will be empty.) Otherwise, box 3 will be emptied and a single rock placed in box 2, preparatory to computation of $f(x, 1)$. If this value is 0, the program halts, with the correct value, Mn [f]$(x) = 1$, in box 2. Otherwise,

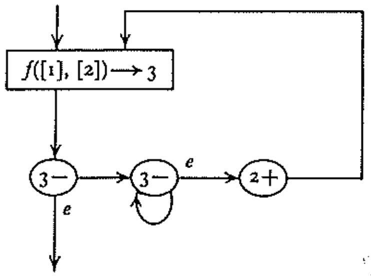


Figure 7-4. Minimization.

another rock is placed in box 2, and the procedure continues until such time (if any) as we have enough rocks (y) in box 2 to make $f(x,y) = 0$.

We have now proved that all recursive functions are abacus computable and hence computable by Turing machines in standard form. To draw the moral for Church's thesis it is necessary to grasp how wide the class of recursive functions is. To get an inkling of that, let us get some sense of how wide the class of primitive recursive functions is.

Examples of primitive recursive functions

7.1 $\text{sum} = \text{Pr}[\text{id}, \text{Cn}[s, \text{id}^3_3]]$ $(\text{sum}(x,y) = x+y)$.

7.2 $\text{prod} = \text{Pr}[z, \text{Cn}[\text{sum}, \text{id}^3_1, \text{id}^3_3]]$ $(\text{prod}(x,y) = x\cdot y)$.

7.3 $\text{exp} = \text{Pr}[\text{Cn}[s,z], \text{Cn}[\text{prod}, \text{id}^3_1, \text{id}^3_3]]$ $(\text{exp}(x,y) = x^y)$.

7.4 $\text{fac} = \text{Pr}[s(0), \text{Cn}[\text{prod}, \text{Cn}[s, \text{id}^2_1], \text{id}^2_2]]$ $(\text{fac}(y) = y!)$.

7.5 $\text{pred} = \text{Pr}[0, \text{id}^2_1]$ (*predecessor*, except that $\text{pred}(0) = 0$).

7.6 $\text{dif} = \text{Pr}[\text{id}, \text{Cn}[\text{pred}, \text{id}^3_3]]$ $(\text{dif}(x,y) = x \dot{-} y = x-y$ if $x \geqslant y$; otherwise, $x \dot{-} y = 0)$.

More informally, the foregoing definitions may be written:

7.1 $x+0 = x, \quad x+y' = (x+y)'$.

7.2 $x\cdot 0 = 0, \quad x\cdot y' = x + x\cdot y$.

7.3 $x^0 = 1, \quad x^{y'} = x\cdot x^y$.

7.4 $0! = 1, \quad y'! = y'\cdot y!$.

7.5 $\text{pred}(0) = 0, \quad \text{pred}(y') = y$.

7.6 $x \dot{-} 0 = x, \quad x \dot{-} y' = \text{pred}(x \dot{-} y)$.

We continue in this informal style:

7.7 $|x-y| = (x \dot{-} y) + (y \dot{-} x)$.

7.8 $\text{sg}(y) = 1 \dot{-} (1 \dot{-} y)$ (*signum*$(0) = 0$, otherwise, $\text{sg}(y) = 1$).

7.9 $\overline{\text{sg}}(y) = 1 \dot{-} y$ $(\overline{\text{sg}}(0) = 1$; otherwise, $\overline{\text{sg}}(y) = 0)$.

7.10 *Definition by cases.* Suppose that the function f is defined in the form

$$f(x,y) = \begin{cases} g_1(x,y) & \text{if} \quad C_1, \\ \vdots \\ g_n(x,y) & \text{if} \quad C_n, \end{cases}$$

where $C_1, \ldots, C_n$ are mutually exclusive, collectively exhaustive conditions on x,y, and the functions $g_1, \ldots, g_n$ are primitive recursive. The *characteristic function* of a condition C_i on x,y is a function c_i which takes the value 1 for argument pairs (x,y) which satisfy the condition, and takes the value 0 for all other argument pairs. Now if the characteristic functions $c_1, \ldots, c_n$ of the conditions $C_1, \ldots, C_n$ in the definition are primitive recursive, so is the function f, for it can be defined by composition out of the gs and cs *via*

$$\boxed{f(x,y) = g_1(x,y)c_1(x,y) + \ldots + g_n(x,y)c_n(x,y)}$$

This works because for each pair of natural number arguments x,y, all but one of the cs must assume the value 0, while the nonzero characteristic function, say, c_i, will assume the value 1 and will correspond to the condition C_i which the numbers x,y actually satisfy.

7.11 *An example of definition by cases*: $\max(x,y)$ = the larger of the numbers x,y. This can be defined by cases as follows:

$$\max(x,y) = \begin{cases} x & \text{if} \quad x \geqslant y, \\ y & \text{if} \quad x < y \end{cases}$$

in the format boxed in 7.10, $g_1 = \text{id}^2_1$ and $g_2 = \text{id}^2_2$. To find the characteristic function c_1 of the condition $x \geqslant y$, observe that $y \dot{-} x = 0$ if and only if $x \geqslant y$. Then $\overline{\text{sg}}(y \dot{-} x) = 1$ if $x \geqslant y$ and $\overline{\text{sg}}(y \dot{-} x) = 0$ otherwise, *viz.*, if $x < y$. Then $\overline{\text{sg}}(y \dot{-} x)$ is the characteristic function of the condition $x \geqslant y$; and apparently $\text{sg}(y \dot{-} x)$ is the characteristic function of the condition $x < y$. Then the definition is

$$\max(x,y) = x\cdot\overline{\text{sg}}(y \dot{-} x) + y\cdot\text{sg}(y \dot{-} x)$$

whereby the function max is defined by composition of primitive recursive functions.

7.12 $\min(x,y) =$ the smaller of $x,y = y\cdot\overline{\text{sg}}(y \dot{-} x) + x\cdot\text{sg}(y \dot{-} x)$.

7.13 *General summation*:

$$g(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n, 0) + \ldots + f(x_1, \ldots, x_n, y) = \sum_{i=0}^{y} f(x_1, \ldots, x_n, i).$$

This function is primitive recursive if f is, with these recursion equations:

$$g(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n, 0),$$
$$g(x_1, \ldots, x_n, y') = f(x_1, \ldots, x_n, y') + g(x_1, \ldots, x_n, y).$$

Example: $n = 0$, $f =$ the square. Then $g(y) = 0^2 + \ldots + y^2$, and the recursion equations are: $g(0) = 0$, $g(y') = y'^2 + g(y)$.

7.14 *General product*:

$$g(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n, 0) \cdot \ldots \cdot f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} f(x_1, \ldots, x_n, i).$$

Here the recursion equations are as in 7.13, except that '$\cdot$' appears in place of '$+$' in the second. *Example*: the product of the first y positive integers. If we set $g(0) = 1$, the second recursion equation will be $g(y') = y' \cdot g(y)$. Thus, $f(0) = 1$ but $f(y') = y'$.

7.15 *Logical composition of conditions*. If C is a condition of which c is the characteristic function, then $\mathrm{Cn}[\overline{\mathrm{sg}}, c]$ is the characteristic function of the condition $-C$ (the *denial* of C) which holds when C fails and fails when C holds. Thus, if C is a condition on x, y, the characteristic function of $-C$ will be the function $\bar{c}$ for which we have $\bar{c}(x, y) = \overline{\mathrm{sg}}(c(x, y))$. Similarly, suppose that $C_1, \ldots, C_n$ are conditions on x, y of which the corresponding characteristic functions are $c_1, \ldots, c_n$. Then the characteristic function of the condition $C_1 \& \ldots \& C_n$ (the *conjunction* of the Cs) will simply be the product of the cs: $c_1(x, y) \cdot \ldots \cdot c_n(x, y)$ will be 1 if all of the conditions $C_1, \ldots, C_n$ are met, and will be 0 if even one of the conditions fails. Since all truth-functional modes of composition of conditions can be built out of denial and conjunction, the characteristic function of any such compound condition will be primitive recursive as long as the characteristic functions of the constituent conditions are primitive recursive. *Example*: *disjunction*. Since $C_1 \vee C_2$ is the denial of the conjunction of the denials of the Cs, $-(-C_1 \& -C_2)$, we have

$$d(x, y) = \overline{\mathrm{sg}}(\overline{\mathrm{sg}}(c_1(x, y)) \cdot \overline{\mathrm{sg}}(c_2(x, y)))$$

if d is the characteristic function of the disjunction $C_1 \vee C_2$.

7.16 *Bounded quantification*. Given a condition $C(x, y)$ on x, y, we can form other conditions by *bounded universal quantification*, $\forall i (i \leqslant y \rightarrow C(x, i))$ ('$C(x, i)$ holds for every i from 0 to y') and *bounded existential quantification*, $\exists i (i \leqslant y \& C(x, i))$ ('$C(x, i)$ holds for some i from 0 to y'). If the characteristic function $c(x, y)$ of condition $C(x, y)$ is primitive recursive, so are the characteristic functions u and e of the conditions obtained by bounded quantification, for we have

$$u(x, y) = \prod_{i=0}^{y} c(x, i), \quad e(x, y) = \mathrm{sg}\left(\sum_{i=0}^{y} c(x, i)\right).$$

(See Examples 7.13 and 7.14.)

7.17 *Minimization with primitive recursive bound*. Suppose that $f(x_1, \ldots, x_n, y)$ is primitive recursive and *regular*. Then $\mathrm{Mn}[f]$ is a total recursive function which need not be primitive recursive. (It *will* be primitive recursive if there is another, equivalent definition in which the Mn operator is not used.) Suppose, however, that we have a primitive recursive function g which gives a bound on the size of the smallest y for which $f(x_1, \ldots, x_n, y) = 0$; suppose, that is, that for each n-tuple $x_1, \ldots, x_n$ there is a y no greater than $g(x_1, \ldots, x_n)$ for which $f(x_1, \ldots, x_n, y) = 0$. In such a case, the function $\mathrm{Mn}[f]$ is necessarily primitive recursive, for we have

$$\mathrm{Mn}[f](x_1, \ldots, x_n) = \sum_{i=0}^{g(x_1, \ldots, x_n)} \mathrm{sg}\left(\prod_{j=0}^{i} f(x_1, \ldots, x_n, j)\right),$$

where the expression on the right is a composition out of functions known to be primitive recursive. To see what is going on, imagine that $x_1, \ldots, x_n$ are given fixed values for which the smallest y that makes $f(x_1, \ldots, x_n, y)$ vanish is, say, y_0. By our assumption, $y_0 \leqslant g(x_1, \ldots, x_n)$. Then for $i = 0, 1, \ldots, y_0 - 1$, each of the products $\prod_{j=0}^{i} f(x_1, \ldots, x_n, j)$ will be positive – for each of $f(x_1, \ldots, x_n, 0), \ldots, f(x_1, \ldots, x_n, y_0 - 1)$ is nonzero and thus positive. There are exactly y_0 such products, so the sum of their *signum* values will be y_0. Starting with $j = y_0$, however, all further products contain a factor of 0 and are thus 0: they do not contribute to the sum, and so the sum of the *signum* values of the products is y_0. By 7.8, 7.13, and 7.14, the function $\mathrm{Mn}[f]$ is thus primitive recursive if f and g are.

7.18 *Bounded minimization*. $\mathrm{Mn}_w[f] =$ the smallest y between 0 and w, inclusive, for which $f(x_1, \ldots, x_n, y) = 0$, and 0 if there is no such y. This function is primitive recursive if f is (whether or not f is regular), for we can define it by cases:

$$\mathrm{Mn}_w[f](x_1, \ldots, x_n) = \begin{cases} 0 & \text{if } \forall y (y \leqslant w \rightarrow f(x_1, \ldots, x_n, y) \neq 0), \\ \sum_{i=0}^{w} \mathrm{sg}\left(\prod_{j=0}^{i} f(x_1, \ldots, x_n, j)\right) & \text{otherwise.} \end{cases}$$

7.19 *Bounded maximization.* $\mathrm{Mx}_w[f]$ = the largest y between 0 and w, inclusive, for which $f(x_1, \ldots, x_n, y) = 0$, and 0 if there is no such y. Notice that if there is such a y, it can be characterized as the *smallest* y no greater than w for which we have $f(x_1, \ldots, x_n, y) = 0$, $f(x_1, \ldots, x_n, y+1) \neq 0$, $f(x_1, \ldots, x_n, y+2) \neq 0, \ldots, f(x_1, \ldots, x_n, w) \neq 0$. Thus we can define Mx_w in terms of Mn_w as follows:

$$\mathrm{Mx}_w[f] = \mathrm{Mn}_w[\mathrm{Cn}[\overline{\mathrm{sg}}, g]]$$

where g is the characteristic function of the condition

$$f(x_1, \ldots, x_n, y) = 0 \,\&\, \forall u(u \leqslant w \to (u > y \to f(x_1, \ldots, x_n, u) \neq 0)).$$

7.20 *Graphs.* If f is an n-place function, then the $(n+1)$-place relation R such that

$$R(x_1, \ldots, x_n, x_{n+1}) \quad \text{iff} \quad f(x_1, \ldots, x_n) = x_{n+1}$$

for all $x_1, \ldots, x_{n+1}$ is called the *graph* of f. If f is primitive recursive, then so is the characteristic function c of its graph R, for since

$$R(x_1, \ldots, x_n, x_{n+1}) \quad \text{iff} \quad |f(x_1, \ldots, x_n) - x_{n+1}| = 0,$$

we have

$$c(x_1, \ldots, x_n, x_{n+1}) = \overline{\mathrm{sg}}(|f(x_1, \ldots, x_n) - x_{n+1}|).$$

(See 7.7 and 7.9.)

7.21 *Eliminating constants.* We have regarded such schemes as

$$h(0) = m$$
$$h(s(y)) = g(y, h(y)), \text{ where } m \text{ is a } \textit{natural number},$$

as admissible definitions of one-place primitive recursive functions h from two-place primitive recursive functions g. But we have gained no generality by doing so. If we let $g^* = \mathrm{Cn}[g, \mathrm{id}^3_2, \mathrm{id}^3_3]$, f_m = the one-place function whose value is always $= m$, and $h^* = \mathrm{Pr}[f_m, g^*]$, then $h = \mathrm{Cn}[h^*, z, \mathrm{id}^1_1]$, and h is thus definable from *the zero function* and the successor and identity functions by repeated applications of composition and primitive recursion provided that g is so definable. For f_m is certainly so definable, $h^*(x, 0) = f_m(x) = m$, and $h^*(x, s(y)) = g^*(x, y, h^*(x, y)) = g(\mathrm{id}^3_2(x, y, h^*(x, y)), \mathrm{id}^3_3(x, y, h^*(x, y))) = g(y, h^*(x, y))$. Thus $h(0) = m = h^*(0, 0)$; and if $h(y) = h^*(0, y)$, then $h(s(y)) = g(y, h(y)) = g(y, h^*(0, y)) = h^*(0, s(y))$ by substituting '0' for 'x'. By induction then, for all y, $h(y) = h^*(0, y) = h^*(z(y), \mathrm{id}^1_1(y))$, and thus $h = \mathrm{Cn}[h^*, z, \mathrm{id}^1_1]$.

8 Turing computable functions are recursive

We have seen that all recursive functions are abacus computable ($R \subseteq A$) and that all abacus computable functions are Turing computable ($A \subseteq T$). We shall now prove that all Turing computable functions are recursive ($T \subseteq R$). This will close the circle of inclusions

$$R \underset{7}{\subseteq} A \underset{6}{\subseteq} T \underset{8}{\subseteq} R$$

and complete the demonstration that the three notions of computability are equivalent:

$$R = A = T.$$

Let us suppose then that f is a function that is computed by a Turing machine M that meets conditions (1)–(5) of Chapter 6. We must show that f is recursive. For definiteness, we suppose f to be a function of two arguments; a construction similar to the present one can be given for any other number of arguments.

Let x_1, x_2 be arbitrary natural numbers. At the beginning of its computation of $f(x_1, x_2)$, M's tape will be completely blank except for two blocks of 1s, which are separated by a single blank square. The left block contains $x_1 + 1$ 1s; the right block, $x_2 + 1$ 1s. At the outset M is scanning the leftmost 1 in the left block. When it halts, it is scanning the leftmost 1 in a block of $f(x_1, x_2) + 1$ 1s on an otherwise completely blank tape. And throughout the computation there are finitely many 1s to the left of the scanned square, finitely many 1s to the right, and at most one 1 in the scanned square.

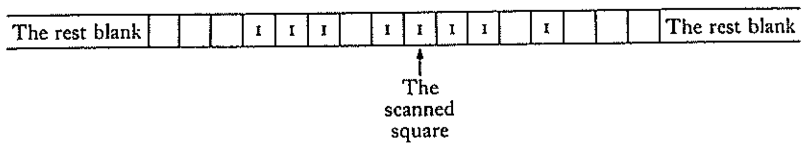
Thus at any time during the computation, if there is a 1 to the left of the scanned square, there is a leftmost 1 to its left, and similarly for the right. We can therefore use *binary notation* to represent the contents of the tape and the content of the scanned square by means of *a pair of natural numbers*, in the following manner:†

† The technique of our proof is due to Hao Wang. See his *Survey of Mathematical Logic*. Science Press (Peking) and North-Holland Publishing Company (Amsterdam), 1963, p. 130.

The infinite portion of the tape to the left of the scanned square can be thought of as containing a binary numeral (e.g. 1011, 0, 1) prefixed by an infinite sequence of superfluous 0s. ($0 = B$.) We call this numeral *the left numeral* and the number it denotes in binary notation *the left number*. The rest of the tape, consisting of the scanned square and the portion to its right, can be thought of as containing a binary numeral, WRITTEN BACKWARDS, to which an infinite sequence of superfluous 0s is also attached. We call this numeral, which appears backwards on the tape, *the right numeral* and the number it denotes *the right number*. Thus the scanned square contains the digit in the 'one's place' of the right numeral. We take the right numeral to be written backwards to insure that changes on the tape will always take place in the vicinity of the one's place of both numerals.

Example 8.1

The tape looks like this:



Then the left numeral is 11101, the right numeral is 10111, the left number is 29, and the right number is 23.

We use '0' and '1' as names both for numerals and for the numbers they denote and rely upon context to resolve ambiguity.

Example 8.2

The tape is completely blank. Then the left numeral = the right numeral = 0, and the left number = the right number = 0.

When the symbol in the scanned square changes from 0 to 1 (1 to 0) the right number increases (decreases) by one and the left number does not change. What happens to the numbers when M moves left or moves right one square?

Let l (r) be the old, i.e. pre-move, left (right) number and let l' (r') be the new left (right) number. We want to see how l' and r' depend upon l, r, and the direction of the move. There are four cases to consider, all quite similar, and we shall examine in detail only the one in which M moves left and l is odd.

Since l is odd, the old left numeral ends in a 1. If $r = 0$, then the new right numeral is 1, and $r' = 1 = 2r+1$. And, if $r > 0$, then the new right numeral comes from the old by appending a 1 to it at its one's place end; again $r' = 2r+1$. As for l', if $l = 1$, then the old left numeral is just 1, the new left numeral is 0, and $l' = 0 = (l-1)/2$. And if l is any odd number greater than 1, then the new left numeral comes from the old by deleting the 1 in its one's place (thus shortening the numeral). This new numeral denotes $(l-1)/2$, and again $l' = (l-1)/2$. To sum up: if M moves left and l is odd, then $l' = (l-1)/2$ and $r' = 2r+1$.

Example 8.1 continued

Suppose M moves left. Then the new left numeral is 1110, $l' = 14 = (29-1)/2$, the new right numeral is 101111, and $r' = 47 = 2\cdot 23+1$.

Similar arguments show that

if M moves left and l is even, then $l' = l/2$ and $r' = 2r$;
if M moves right and r is odd, then $l' = 2l+1$ and $r' = (r-1)/2$; and
if M moves right and r is even, then $l' = 2l$ and $r' = r/2$.

What are the left and right numbers when M begins its computation? The tape is then completely blank to the left of the scanned square, and so the left numeral is 0 and the left number is 0. The right numeral is

$$\underbrace{1\ldots 1}_{x_2+1 \text{ 1s}} \quad 0 \quad \underbrace{1\ldots 1}_{x_1+1 \text{ 1s}}$$

A sequence of m 1s denotes 2^m-1 in binary notation; a sequence of m 1s followed by n 0s denotes $(2^m-1)2^n$; and a sequence of m 1s followed by one 0 followed by p 1s denotes $(2^m-1)2^{p+1}+(2^p-1)$. Thus the right number at the beginning of M's computation of $f(x_1, x_2)$ is $(2^{(x_2+1)} \dot{-} 1)2^{(x_1+2)}+(2^{(x_1+1)} \dot{-} 1)$. Let $s(x_1, x_2) = (2^{(x_2+1)} \dot{-} 1)2^{(x_1+2)}+(2^{(x_1+1)} \dot{-} 1)$. Then s is a primitive recursive function.

And what are the left and right numbers when M halts? When it halts, M is scanning the leftmost 1 of a block of $f(x_1, x_2)+1$ 1s on an otherwise blank tape. The left number is again 0, and the right number is $2^{f(x_1,x_2)+1}-1$, which is the number denoted in binary by a string of $f(x_1, x_2)+1$ 1s.

Let $\text{lo}(x)$ = the greatest $w \leqslant x$ such that $2^w \leqslant x$. By 7.19, lo is a primitive recursive function. (In fact, if $d(x,y) = 0$ if $2^y \leqslant x$, and

1 otherwise, then $\mathrm{lo}(x) = \mathrm{M}x_z[d](x)$.) $\mathrm{lo}(2^{z+1}-1) = z$. Thus $\mathrm{lo}(2^{f(x_1,x_2)+1}-1) = f(x_1,x_2)$. It follows that if M halts its computation of $f(x_1,x_2)$ at t and r is the right number at t, then $\mathrm{lo}(r) = f(x_1,x_2)$.

We turn now to the task of representing the flow graph or machine table of M by two primitive recursive functions, a and q. We interpret the machine states q_1, etc. as natural numbers 1 etc., the symbols B, 1 as the natural numbers 0, 1, and the acts $B, 1, L, R$, as the natural numbers 0, 1, 2, 3. If in state q_i while scanning 0, M performs act $j(= 0,1,2,3)$ and goes into state q_k, then we set $a(i,0) = j$ and $q(i,0) = k$; if in state q_i while scanning 1, M performs act j and goes into state q_k, we likewise set $a(i,1) = j$ and $q(i,1) = k$. For all pairs of numbers x,y not covered by this stipulation we set $a(x,y) = y$ and $q(x,y) = 0$. To halt, then, is to enter notional state 0. Since there are only finitely many quadruples in M's machine table, it is clear that a and q are primitive recursive functions, definable by cases.

Example 8.3

Consider the graph in Figure 8-1, of a machine which computes the successor function.

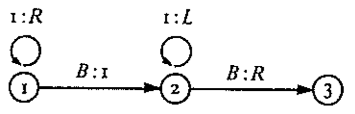


Figure 8-1. Computing the function s.

$$a(x,y) = \begin{cases} 1 & \text{if } x = 1 \text{ and } y = 0, \\ 3 & \text{if } x = 1 \text{ and } y = 1, \\ 3 & \text{if } x = 2 \text{ and } y = 0, \\ 2 & \text{if } x = 2 \text{ and } y = 1, \\ y & \text{otherwise} \end{cases}$$

$$q(x,y) = \begin{cases} 2 & \text{if } x = 1 \text{ and } y = 0, \\ 1 & \text{if } x = 1 \text{ and } y = 1, \\ 3 & \text{if } x = 2 \text{ and } y = 0, \\ 2 & \text{if } x = 2 \text{ and } y = 1, \\ 0 & \text{otherwise.} \end{cases}$$

In order to show that f is recursive, we need a device to code ordered triples of natural numbers as single natural numbers, and devices for recovering the first, second, and third components of a triple from the single number that codes the triple. The need for these devices arises from the fact that what the left number is at stage $t+1$ of the computation depends not only upon what the left number is at t but also upon what the right number is at t and upon the state of M at t. Similarly for the right number and the state of M at t. But once we have coded triples as single numbers, we shall be able to show that the triple l', c', r' consisting of the left number at $t+1$, the state at $t+1$, and the right number at $t+1$, depends (primitive recursively) on the triple l, c, r of the left number at t, the state at t, and the right number at t.

We thus define $\mathrm{tpl}(x,y,z) = 2^x3^y5^z$. tpl is primitive recursive; if $\mathrm{tpl}(x,y,z) = \mathrm{tpl}(a,b,c)$, then $x = a$, $y = b$, and $z = c$. We shall henceforth write '$\langle x,y,z\rangle$' instead of '$\mathrm{tpl}(x,y,z)$'.

Let $\mathrm{lft}(w)$ be the greatest $x \leqslant w$ such that 2^x divides w (without remainder), let $\mathrm{ctr}(w)$ be the greatest $x \leqslant w$ such that 3^x divides w, and let $\mathrm{rgt}(w)$ be the greatest $x \leqslant w$ such that 5^x divides w. By 7.19, lft, ctr, and rgt are primitive recursive. The functions lft, ctr, and rgt are inverses to tpl in the sense that if $w = \langle x,y,z\rangle$, then $\mathrm{lft}(w) = x$, $\mathrm{ctr}(w) = y$, and $\mathrm{rgt}(w) = z$.

We shall now define a primitive recursive function g of three arguments with the following property: *If t is a stage not later than the stage at which M halts when computing $f(x_1,x_2)$, then $g(x_1,x_2,t) = \langle$the left number at t, the state of M at t, the right number at $t\rangle$*. Since M is in state 1 when it begins its computation, we let

$$g(x_1,x_2,0) = \langle 0, 1, s(x_1,x_2)\rangle.$$

We let $e(x) = 0$ if x is even, and 1 if x is odd. The function e is of course primitive recursive. A number is even if and only if there is a 0 in the one's place of its binary representation. Thus if r is the right number at t, then $e(r) = 0$ if the symbol in the square scanned at t is $0(= B)$, and $e(r) = 1$ if this symbol is 1.

We can now complete our definition of g. In what follows we use 'l', 'c', and 'r' to abbreviate '$\mathrm{lft}(g(x_1,x_2,t))$', '$\mathrm{ctr}(g(x_1,x_2,t))$', and '$\mathrm{rgt}(g(x_1,x_2,t))$', respectively. We also abbreviate '$q(c,e(r))$', i.e. '$q(\mathrm{ctr}(g(x_1,x_2,t)), e(\mathrm{rgt}(g(x_1,x_2,t))))$' by '$q$'. Let

$$g(x_1, x_2, t+1) = \begin{cases} \langle l, q, r\rangle & \text{if } a(c, e(r)) = 0 \text{ and } e(r) = 0, \\ \langle l, q, r \dot{-} 1\rangle & \text{if } a(c, e(r)) = 0 \text{ and } e(r) = 1, \\ \text{(write a } B \text{ in the scanned square)} & \\ \langle l, q, r+1\rangle & \text{if } a(c, e(r)) = 1 \text{ and } e(r) = 0, \\ \langle l, q, r\rangle & \text{if } a(c, e(r)) = 1 \text{ and } e(r) = 1, \\ \text{(write a 1 in the scanned square)} & \\ \langle l/2, q, 2r\rangle & \text{if } a(c, e(r)) = 2 \text{ and } e(l) = 0, \\ \langle (l \dot{-} 1)/2, q, 2r+1\rangle & \text{if } a(c, e(r)) = 2 \text{ and } e(l) = 1, \\ \text{(move left one square)} & \\ \langle 2l, q, r/2\rangle & \text{if } a(c, e(r)) = 3 \text{ and } e(r) = 0, \\ \langle 2l+1, q, (r \dot{-} 1)/2\rangle & \text{if } a(c, e(r)) = 3 \text{ and } e(r) = 1, \\ \text{(move right one square)} & \\ 0 & \text{otherwise.} \end{cases}$$

(We take / to be defined primitive recursively so that x/y is the quotient on dividing x by y if there is a natural number that is the quotient, and to be (say) 0 if there is no such natural number.)

Since tpl, lft, ctr, rgt, s, q, e, a, $\dot{-}$, $+$, $\cdot$, and / are all primitive recursive, g is primitive recursive too.

Clause 2 of the definition of g can be interpreted: if M is in state c scanning a 1 and its machine table tells it to print a 0 on the scanned square, then the left number remains the same, the next state of M is the one prescribed by its table, and the right number decreases by 1. The other clauses have similar interpretations.

Our definitions guarantee that if t is the stage at which M halts, then M goes into state 0 for the first time at $t+1$. Thus $\text{ctr}(g(x_1, x_2, y)) \neq 0$ for all $y \leqslant t$. And if l, c, and r are the left number at t, the state of M at t, and the right number at t, respectively, then $a(c, e(r)) = e(r)(= 0, 1)$ and $q(c, e(r)) = 0$, and therefore $g(x_1, x_2, t+1) = \langle l, 0, r\rangle$ and $\text{ctr}(g(x_1, x_2, t+1)) = 0$. Thus M halts at t when computing $f(x_1, x_2)$ if and only if t is the least y such that $\text{ctr}(g(x_1, x_2, y+1)) = 0$. Let $h(x_1, x_2, y) = \text{ctr}(g(x_1, x_2, y+1))$. h is primitive recursive, and M halts at t when computing $f(x_1, x_2)$ iff $\text{Mn}[h](x_1, x_2) = t$. $\text{Mn}[h]$ is a recursive function. (It is not possible in general to conclude that it is a *primitive* recursive function.) And if M halts at t, and r is the right number at t, then $f(x_1, x_2) = \text{lo}(r) = \text{lo}(\text{rgt}(g(x_1, x_2, t)))$. Thus

$$f(x_1, x_2) = \text{lo}(\text{rgt}(g(x_1, x_2, \text{Mn}[h](x_1, x_2)))),$$

for all x_1, x_2. f is therefore a recursive function. The circle is closed.

We have also proved, by the way,

The Kleene normal form theorem for recursive functions

In obtaining a recursive function from basic functions via composition, primitive recursion, and minimization, the operation of minimization need not be used more than once.

Our usage of the term 'recursive' has been somewhat non-standard. What we have been calling the recursive functions are ordinarily called the *partial recursive* functions, while the functions that *are* standardly called 'recursive' are the partial recursive functions that are total, i.e. that assign values to *all* n-tuples of natural numbers (for the appropriate n). As we shall have no further dealings with partial recursive functions that are not total, we adopt standard usage henceforth: from now on, a recursive function is always *total*.

Exercise (John Burgess)

Let us say that the Turing machine M computes the n-place function f *in the wide sense* if and only if for all $x_1, \ldots, x_n, y$, $f(x_1, \ldots, x_n)$ is defined and equal to y iff M, when started in standard position for $x_1, \ldots, x_n$, halts in standard position for y. (Every machine, even one that halts in nonstandard positions on some arguments, computes an n-place function in the wide sense.) Show that a function is partial recursive if and only if some machine computes it in the wide sense. [Hint: M halts in standard position if and only if the left number is 0 and the right number is positive and one less than a power of two.]