1
Computable functions

We begin this chapter with a discussion of the fundamental idea of an algorithm or effective procedure. In subsequent sections we describe the way in which this idea can be made precise using a kind of idealised computer; this lays the foundation for a mathematical theory of computability and computable functions.

1. Algorithms, or effective procedures

When taught arithmetic in junior school we all learnt to add and to multiply two numbers. We were not merely taught that any two numbers have a sum and a product – we were given methods or rules for finding sums and products. Such methods or rules are examples of *algorithms* or *effective procedures*. Their implementation requires no ingenuity or even intelligence beyond that needed to obey the teacher's instructions.

More generally, an *algorithm* or *effective procedure* is a mechanical rule, or automatic method, or programme for performing some mathematical operation. Some more examples of operations for which easy algorithms can be given are

(1.1) (a) given n, finding the nth prime number,
(b) differentiating a polynomial,
(c) finding the highest common factor of two numbers (the Euclidean algorithm),
(d) given two numbers x, y deciding whether x is a multiple of y.

Algorithms can be represented informally as shown in fig. 1a. The input is the raw data or object on which the operation is to be performed (e.g. a polynomial for (1.1) (b), a pair of numbers for (1.1) (c) and (d)) and the output is the result of the operation (e.g. for (1.1) (b), the derived polynomial, and for (1.1) (d), the answer yes or no). The output is produced mechanically by the black box – which could be thought of as a

Fig. 1a.

calculating machine, a computer, or a schoolboy correctly taught – or even a very clever dog trained appropriately. The algorithm is the procedure or method that is carried out by the black box to obtain the output from the input.

When an algorithm or effective procedure is used to calculate the values of a numerical function then the function in question is described by phrases such as *effectively calculable*, or *algorithmically computable*, or *effectively computable*, or just *computable*. For instance, the functions xy, $\mathrm{HCF}(x, y)$ = the highest common factor of x and y, and $f(n)$ = the nth prime number, are computable in this informal sense, as already indicated. Consider, on the other hand, the following function:

$$g(n) = \begin{cases} 1 & \text{if there is a run of exactly } n \text{ consecutive 7s} \\ & \text{in the decimal expansion of } \pi, \\ 0 & \text{otherwise.} \end{cases}$$

Most mathematicians would accept that g is a perfectly legitimate function. But is g computable? There *is* a mechanical procedure for generating successive digits in the decimal expansion of π,[1] so the following 'procedure' for computing g suggests itself.

'Given n, start generating the decimal expansion of π, one digit at a time, and watch for 7s. If at some stage a run of exactly n consecutive 7s has appeared, then stop the process and put $g(n) = 1$. If no such sequence of 7s appears put $g(n) = 0$.'

The problem with this 'procedure' is that, if for a particular n there is no sequence of exactly n consecutive 7s, then there is no stage in the process where we can stop and conclude that this is the case. For all we know at any particular stage, such a sequence of 7s could appear in the part of the expansion of π that has not yet been examined. Thus the 'procedure' will go on for ever for inputs n such that $g(n) = 0$; so it is not an *effective* procedure. (It is conceivable that there *is* an effective procedure for computing g based, perhaps, on some theoretical properties of π. At the present time, however, no such procedure is known.)

[1] This will be established in chapter 3 (example 7.1(3)).

This example pinpoints two features implicit in the idea of an effective procedure – namely, that such a procedure is carried out in a sequence of stages or steps (each completed in a finite time), and that any output should emerge after a finite number of steps.

So far we have described informally the idea of an algorithm, or effective procedure, and the associated notion of computable function. These ideas must be made precise before they can become the basis for a mathematical theory of computability – and *non*-computability.

We shall make our definitions in terms of a simple 'idealised computer' that operates programs. Clearly, the procedures that can be carried out by a real computer are examples of effective procedures. Any particular real computer, however, is limited both in the size of the numbers that it can receive as input, and in the amount of working space available; it is in these respects that our 'computer' will be idealised in accordance with the informal idea of an algorithm. The programs for our machine will be finite, and we will require that a completed computation takes only a finite number of steps. Inputs and outputs will be restricted to natural numbers; this is not a significant restriction, since operations involving other kinds of object can be coded as operations on natural numbers. (We discuss this more fully in § 5.)

2. The unlimited register machine

Our mathematical idealisation of a computer is called an *unlimited register machine* (URM); it is a slight variation of a machine first conceived by Shepherdson & Sturgis [1963]. In this section we describe the URM and how it works; we begin to explore what it can do in § 3.

The URM has an infinite number of *registers* labelled $R_1, R_2, R_3, \ldots$, each of which at any moment of time contains a natural number; we denote the number contained in R_n by r_n. This can be represented as follows

R_1	R_2	R_3	R_4	R_5	R_6	R_7	...
r_1	r_2	r_3	r_4	r_5	r_6	r_7	...

The contents of the registers may be altered by the URM in response to certain *instructions* that it can recognise. These instructions correspond to very simple operations used in performing calculations with numbers. A finite list of instructions constitutes a *program*. The instructions are of four kinds, as follows.

Zero instructions For each $n = 1, 2, 3, \ldots$ there is a *zero instruction* $Z(n)$. The response of the URM to the instruction $Z(n)$ is to change the contents of R_n to 0, leaving all other registers unaltered.

Example Suppose that the URM is in the following configuration

R_1	R_2	R_3	R_4	R_5	R_6	
9	6	5	23	7	0	...

and obeys the zero instruction Z(3). Then the resulting configuration is

(*)

9	6	0	23	7	0	...

The response of the URM to a zero instruction $Z(n)$ is denoted by $0 \to R_n$, or $r_n := 0$ (this is read r_n *becomes* 0).

Successor instructions For each $n = 1, 2, 3, \ldots$ there is a *successor instruction* $S(n)$. The response of the URM to the instruction $S(n)$ is to increase the number contained in R_n by 1, leaving all other registers unaltered.

Example Suppose that the URM is in the configuration (*) above and obeys the successor instruction S(5). Then the new configuration is

	R_1	R_2	R_3	R_4	R_5	R_6	
(**)	9	6	0	23	8	0	...

The effect of a successor instruction $S(n)$ is denoted by $r_n + 1 \to R_n$, or $r_n := r_n + 1$ (r_n *becomes* $r_n + 1$).

Transfer instructions For each $m = 1, 2, 3, \ldots$ and $n = 1, 2, 3, \ldots$ there is a *transfer instruction* $T(m, n)$. The response of the URM to the instruction $T(m, n)$ is to replace the contents of R_n by the number r_m contained in R_m (i.e. transfer r_m into R_n); all other registers (including R_m) are unaltered.

Example Suppose that the URM is in the configuration (**) above and obeys the transfer instruction T(5, 1). Then the resulting configuration is

R_1	R_2	R_3	R_4	R_5	R_6	
8	6	0	23	8	0	...

The response of the URM to a transfer instruction $T(m, n)$ is denoted by $r_m \to R_n$, or $r_n := r_m$ (r_n *becomes* r_m).

Jump instructions In the operation of an informal algorithm there may be a stage when alternative courses of action are prescribed, depending on the progress of the operation up to that stage. In other situations it may be necessary to repeat a given routine several times. The URM is able to reflect such procedures as these using *jump instructions*; these will allow jumps backwards or forwards in the list of instructions. We shall, for example, be able to use a jump instruction to produce the following response:

> 'If $r_2 = r_6$, go to the 10th instruction in the program; otherwise, go on to the next instruction in the program.'

The instruction eliciting this response will be written J(2, 6, 10).

Generally, for each $m = 1, 2, 3, \ldots$, $n = 1, 2, 3, \ldots$ and $q = 1, 2, 3, \ldots$ there is a *jump instruction* $J(m, n, q)$. The response of the URM to the instruction $J(m, n, q)$ is as follows. Suppose that this instruction is encountered in a program P. The contents of R_m and R_n are compared, but all registers are left unaltered. Then

if $r_m = r_n$, the URM proceeds to the qth instruction of P;

if $r_m \neq r_n$, the URM proceeds to the next instruction in P.

If the jump is impossible because P has less than q instructions, then the URM stops operation.

Zero, successor and transfer instructions are called *arithmetic* instructions.

We summarise the response of the URM to the four kinds of instruction in table 1.

Computations To perform a computation the URM must be provided with a program P and an *initial configuration* – i.e. a sequence $a_1, a_2, a_3, \ldots$ of natural numbers in the registers $R_1, R_2, R_3, \ldots$. Suppose that P consists of s instructions $I_1, I_2, \ldots, I_s$. The URM begins the computation by obeying I_1, then I_2, I_3, and so on unless a jump

Table 1

Type of instruction	Instruction	Response of the URM
Zero	$Z(n)$	Replace r_n by 0. ($0 \to R_n$, or $r_n := 0$)
Successor	$S(n)$	Add 1 to r_n. ($r_n + 1 \to R_n$, or $r_n := r_n + 1$)
Transfer	$T(m, n)$	Replace r_n by r_m. ($r_m \to R_n$, or $r_n := r_m$)
Jump	$J(m, n, q)$	If $r_m = r_n$, jump to the qth instruction; otherwise go on to the next instruction in the program.

instruction, say $J(m, n, q)$, is encountered. In this case the URM proceeds to the instruction prescribed by $J(m, n, q)$ and the current contents of the registers R_m and R_n. We illustrate this with an example.

2.1. *Example*
Consider the following program:

I_1 J(1, 2, 6)
I_2 S(2)
I_3 S(3)
I_4 J(1, 2, 6)
I_5 J(1, 1, 2)
I_6 T(3, 1)

Let us consider the computation by the URM under this program with initial configuration

R_1	R_2	R_3	R_4	R_5	
9	7	0	0	0	...

(We are not concerned at the moment about what function this program actually computes; we wish to illustrate the way in which the URM operates programs in a purely mechanical fashion *without* needing to understand the algorithm that is being carried out.)

We can represent the progress of the computation by writing down the successive configurations that occur, together with the next instruction to be obeyed at the completion of each stage.

	R_1	R_2	R_3	R_4	R_5		Next instruction
Initial configuration	9	7	0	0	0	...	I_1
	9	7	0	0	0		I_2 (since $r_1 \neq r_2$)
	9	8	0	0	0		I_3
	9	8	1	0	0		I_4
	9	8	1	0	0		I_5 (since $r_1 \neq r_2$)
	9	8	1	0	0		I_2 (since $r_1 = r_1$)

and so on. (We shall continue this computation later.)

We can describe the operation of the URM under a program $P = I_1, I_2, \ldots, I_s$ in general as follows. The URM starts by obeying instruction I_1. At any future stage in the computation, suppose that the URM is obeying instruction I_k. Then having done so it proceeds to the *next instruction in the computation*, defined as follows:

if I_k is not a jump instruction, the *next instruction* is I_{k+1};

if $I_k = J(m, n, q)$ the *next instruction* is $\begin{cases} I_q & \text{if } r_m = r_n, \\ I_{k+1} & \text{otherwise,} \end{cases}$

where r_m, r_n are the current contents of R_m and R_n.

The URM proceeds thus as long as possible; the computation *stops* when, and only when, there is no next instruction; i.e. if the URM has just obeyed instruction I_k and the 'next instruction in the computation' according to the above definition is I_v where $v > s$. This can happen in the following ways:

(i) if $k = s$ (the last instruction in P has been obeyed) and I_s is an arithmetic instruction,
(ii) if $I_k = J(m, n, q)$, $r_m = r_n$ and $q > s$,
(iii) if $I_k = J(m, n, q)$, $r_m \neq r_n$ and $k = s$.

We say then that the computation stops after instruction I_k; the *final configuration* is the sequence $r_1, r_2, r_3, \ldots$, the contents of the registers at this stage.

Let us now continue the computation begun in example 2.1.

Example 2.1 (continued)

	R_1	R_2	R_3	R_4	R_5		Next instruction
	9	8	1	0	0	...	I_2
	9	9	1	0	0		I_3
	9	9	2	0	0		I_4
	9	9	2	0	0		I_6 (since $r_1 = r_2$)
Final configuration	2	9	2	0	0		I_7: STOP.

This computation stops as indicated because there is no seventh instruction in the program.

2.2. *Exercise*

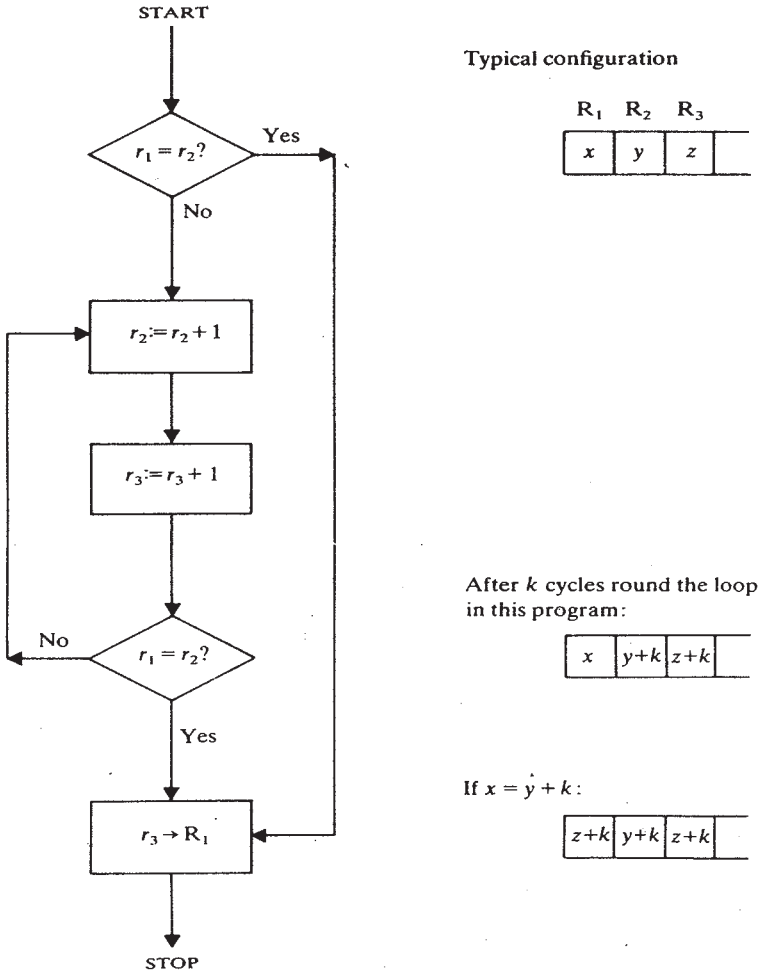
Carry out the computation under the program of example 2.1 with initial configuration 8, 4, 2, 0, 0, . . .

The essence of a program and the progress of computations under it is often conveniently described informally using a *flow diagram*. For example, a flow diagram representing the program of example 2.1 is given in fig. 1*b*. (We have indicated alongside the flow diagram the typical configuration of the registers at various stages in a computation.) Note the convention that tests or questions (corresponding to jump instructions) are placed in diamond shaped boxes.

The translation of this flow diagram into the program of exercise 2.1 is almost self-explanatory. Notice that the backwards jump on answer 'No' to the second question '$r_1 = r_2$?' is achieved by the fifth instruction J(1, 1, 2) which is an *unconditional* jump: we always have $r_1 = r_1$, so this instruction causes a jump to I_2 whenever it is encountered.

When writing a program to perform a given procedure it is often helpful to write an informal flow diagram as an intermediate step: the translation of a flow diagram into a program is then usually routine.

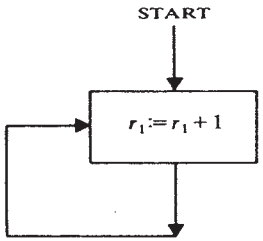
Fig. 1*b*. Flow diagram for the program of example 2.1.



There are, of course, computations that never stop: for example, no computation under the simple program S(1), J(1, 1, 1) ever stops. Computation under this program is represented by the flow diagram in fig. 1*c*. The jump instruction invariably causes the URM to return, or loop back, to the instruction S(1).

There are more sophisticated ways in which a computation may run for ever, but always this is caused essentially by the above kind of repetition or looping back in the execution of the program.

Fig. 1c.



2.3 *Exercise*

Show that the computation under the program of example 2.1 with initial configuration 2, 3, 0, 0, 0, . . . never stops.

The question of deciding whether a particular computation eventually stops or not is one to which we will return later.

Some notation will help us now in our discussion. Let $a_1, a_2, a_3, \ldots$ be an infinite sequence from $\mathbb{N}$ and let P be a program; we will write

(i) $P(a_1, a_2, a_3, \ldots)$ for the computation under P with initial configuration $a_1, a_2, a_3, \ldots$;

(ii) $P(a_1, a_2, a_3, \ldots)\downarrow$ to mean that the computation $P(a_1, a_2, a_3, \ldots)$ eventually stops;

(iii) $P(a_1, a_2, a_3, \ldots)\uparrow$ to mean that the computation $P(a_1, a_2, a_3, \ldots)$ never stops.

In most initial configurations that we shall consider, all but finitely many of the a_i will be 0. Thus the following notation is useful. Let $a_1, a_2, \ldots, a_n$ be a finite sequence of natural numbers; we write

(iv) $P(a_1, a_2, \ldots, a_n)$ for the computation $P(a_1, a_2, \ldots, a_n, 0, 0, 0, \ldots)$,

Hence

(v) $P(a_1, a_2, \ldots, a_n)\downarrow$ means that $P(a_1, a_2, \ldots, a_n, 0, 0, 0, \ldots)\downarrow$;

(vi) $P(a_1, a_2, \ldots, a_n)\uparrow$ means that $P(a_1, a_2, \ldots, a_n, 0, 0, 0, \ldots)\uparrow$.

Often a computation that stops is said to *converge*, and one that never stops is said to *diverge*.

3. URM-computable functions

Suppose that f is a function from $\mathbb{N}^n$ to $\mathbb{N}$ ($n \geq 1$); what does it mean to say that f is computable by the URM? It is natural to think in terms of computing a value $f(a_1, \ldots, a_n)$ by means of a program P on initial configuration $a_1, a_2, \ldots, a_n, 0, 0, \ldots$. That is, we consider computations of the form $P(a_1, a_2, \ldots, a_n)$. If any such computation

stops, we need to have a single number that we can regard as the output or result of the computation; we make the convention that this is the number r_1 finally contained in R_1. The final contents of the other registers can be regarded as rough work or jottings, that can be ignored once we have the desired result in R_1.

Since a computation $P(a_1, \ldots, a_n)$ may not stop, we can allow our definition of computability to apply to functions f from $\mathbb{N}^n$ to $\mathbb{N}$ whose domain may not be all of $\mathbb{N}^n$; i.e. partial functions. We shall require that the relevant computations stop (and give the correct result!) *precisely* for inputs from the domain of f. Thus we make the following definitions.

3.1 *Definitions*

Let f be a partial function from $\mathbb{N}^n$ to $\mathbb{N}$.

(*a*) Suppose that P is a program, and let $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$.

(i) The computation $P(a_1, a_2, \ldots, a_n)$ *converges to* b if $P(a_1, a_2, \ldots, a_n)\downarrow$ and in the final configuration b is in R_1. We write this $P(a_1, \ldots, a_n)\downarrow b$;

(ii) P *URM-computes* f if, for every $a_1, \ldots, a_n, b$ $P(a_1, \ldots, a_n)\downarrow b$ if and only if $(a_1, \ldots, a_n) \in \mathrm{Dom}(f)$ and $f(a_1, \ldots, a_n) = b$. (In particular, this means that $P(a_1, \ldots, a_n)\downarrow$ if and only if $(a_1, \ldots, a_n) \in \mathrm{Dom}(f)$.)

(*b*) The function f is *URM-computable* if there is a program that URM-computes f.

The class of URM-computable functions is denoted by $\mathscr{C}$, and n-ary URM-computable functions by $\mathscr{C}_n$. From now on we will use the term *computable* to mean URM-computable, except in chapter 3 where other notions of computability are discussed.

We now consider some easy examples of computable functions.

3.2 *Examples*

(*a*) $x + y$.

We obtain $x + y$ by adding 1 to x (using the successor instruction) y times. A program to compute $x + y$ must begin on initial configuration $x, y, 0, 0, 0, \ldots$; our program will keep adding 1 to r_1, using R_3 as a counter to keep a record of how many times r_1 is thus increased. A typical configuration during the computation is

R_1	R_2	R_3	R_4	R_5	
$x+k$	y	k	0	0	. . .

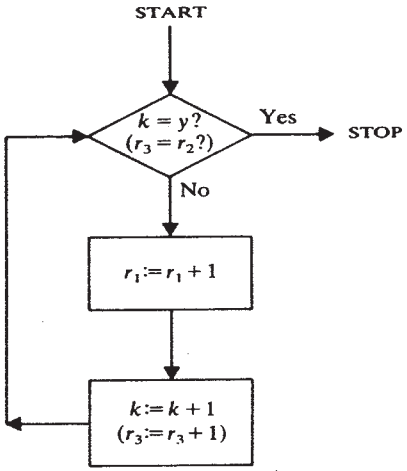
The program will be designed to stop when $k = y$, leaving $x + y$ in R_1 as required.

The procedure we wish to embody in our program is represented by the flow diagram in fig. 1*d*. A program that achieves this is the following:

I_1 J(3, 2, 5) ←--
I_2 S(1)
I_3 S(3)
I_4 J(1, 1, 1) ---

(The dotted arrow, which is *not* part of the program, is to indicate to the reader that the final instruction has the effect of always jumping back to the first instruction.) Note that the STOP has been achieved by a jump instruction to 'I_5' which does not exist. Thus, $x + y$ is computable.

Fig. 1*d*. Flow diagram for addition (example 3.2(*a*)).



(*b*) $x \dot{-} 1 = \begin{cases} x - 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0. \end{cases}$

(Since we are restricting ourselves to functions from $\mathbb{N}$ to $\mathbb{N}$, this is the best approximation to the function $x - 1$.)

We will write a program embodying the following procedure. Given initial configuration $x, 0, 0, 0, \ldots$, first check whether $x = 0$; if so, stop; otherwise, run two counters, containing k and $k + 1$, starting with $k = 0$. A typical configuration during a computation will be

R_1	R_2	R_3	R_4	
x	k	$k+1$	0	...

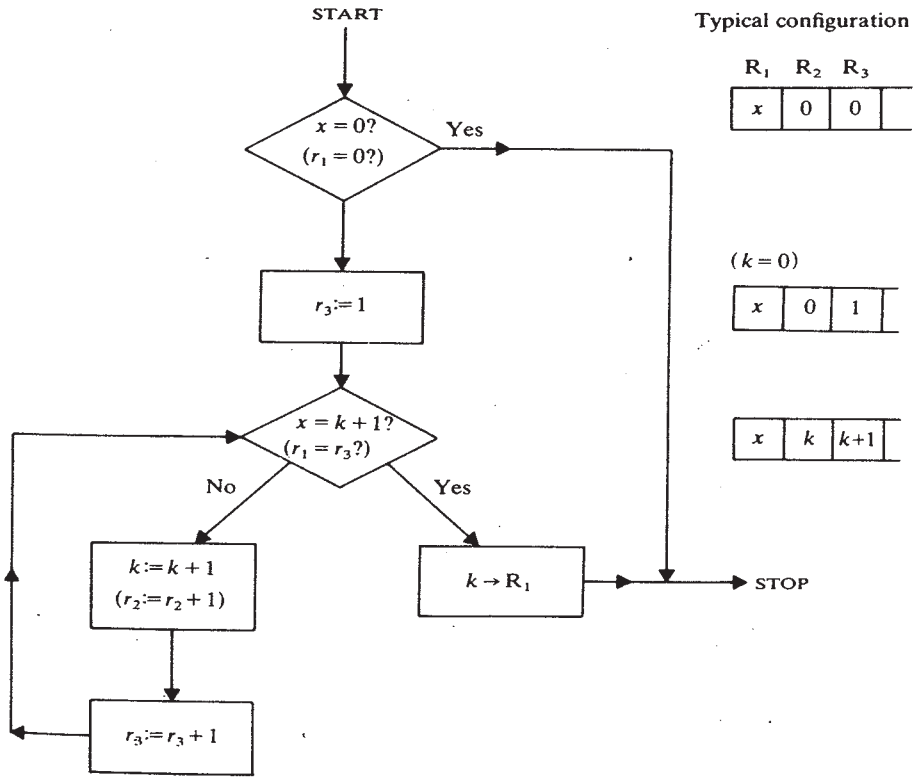
Check whether $x = k + 1$; if so, the required result is k; otherwise increase both counters by 1, and check again.

A flow diagram representing this procedure is given in fig. 1*e*. A program that carries out this procedure is the following:

I_1 J(1, 4, 9)
I_2 S(3)
I_3 J(1, 3, 7) ←--
I_4 S(2)
I_5 S(3)
I_6 J(1, 1, 3) ---
I_7 T(2, 1)

Thus the function $x \dot{-} 1$ is computable.

Fig. 1*e*. Flow diagram for $x \dot{-} 1$ (example 3.2(*b*)).



(c) $f(x) = \begin{cases} \frac{1}{2}x & \text{if } x \text{ is even,} \\ \text{undefined} & \text{if } x \text{ is odd.} \end{cases}$

In this example, $\mathrm{Dom}(f) = \mathbb{E}$ (the even natural numbers) so we must ensure that our program does not stop on odd inputs.

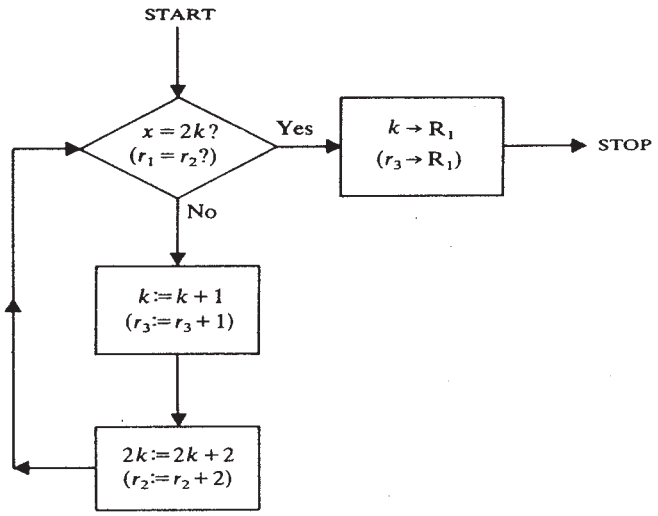
A procedure for computing $f(x)$ is as follows. Run two counters, containing k and $2k$ for $k = 0, 1, 2, 3, \ldots$; for successive values of k, check whether $x = 2k$; if so, the answer is k; otherwise increase k by one, and repeat. If x is odd, this procedure will clearly continue for ever.

The typical configuration will be

R_1	R_2	R_3	R_4	
x	$2k$	k	0	...

with $k = 0$ initially. A flow diagram for the above process is given in fig. 1*f*.

Fig. 1*f*. Flow diagram for example 3.2(*c*)).



A program that executes it is

I_1 J(1, 2, 6)
I_2 S(3)
I_3 S(2)
I_4 S(2)
I_5 J(1, 1, 1)
I_6 T(3, 1)

Hence f is computable.

Note. The programs in these examples are in no sense the only programs that will compute the functions in question.

Given any program P (i.e. any finite list of instructions), and $n \geq 1$, by thinking of the effect of P on initial configurations of the form $a_1, a_2, \ldots, a_n, 0, 0, \ldots$ we see that there is a unique n-ary function that P computes, denoted by $f_P^{(n)}$. From the definition it is clear that

$$f_P^{(n)}(a_1, \ldots, a_n) = \begin{cases} \text{the unique } b \text{ such that } P(a_1, \ldots, a_n)\downarrow b, \\ \quad \text{if } P(a_1, \ldots, a_n)\downarrow; \\ \text{undefined, if } P(a_1, \ldots, a_n)\uparrow. \end{cases}$$

In a later chapter we shall consider the problem of determining $f_P^{(n)}$ for any given program P.

It is clear that a particular computable function can be computed by many different programs; for instance, any program can be altered by adding instructions that have no effect. Less trivially, there may be different informal methods for calculating a particular function, and when formalised as programs these would give different programs for the same function. In terms of the notation we have introduced, we can have different programs P_1 and P_2, with $f_{P_1}^{(n)} = f_{P_2}^{(n)}$ for some (or all) n. Later we shall consider the problem of deciding whether or not two programs compute the same functions.

3.3 *Exercises*

1. Show that the following functions are computable by devising programs that will compute them.

(*a*) $f(x) = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{if } x \neq 0; \end{cases}$

(*b*) $f(x) = 5$;

(*c*) $f(x, y) = \begin{cases} 0 & \text{if } x = y, \\ 1 & \text{if } x \neq y; \end{cases}$

(*d*) $f(x, y) = \begin{cases} 0 & \text{if } x \leq y, \\ 1 & \text{if } x > y; \end{cases}$

(*e*) $f(x) = \begin{cases} \frac{1}{3}x \text{ if } x \text{ is a multiple of 3,} \\ \text{undefined otherwise;} \end{cases}$

(*f*) $f(x) = [2x/3]$. ($[z]$ denotes the greatest integer $\leq z$).

2. Let P be the program in example 2.1. What is $f_P^{(2)}$?
3. Suppose that P is a program without any jump instructions. Show that there is a number m such that either

$$f_P^{(1)}(x) = m, \qquad \text{for all } x,$$

or

$$f_P^{(1)}(x) = x + m, \qquad \text{for all } x.$$

4. Show that for each transfer instruction $T(m, n)$ there is a program without any transfer instructions that has exactly the same effect as $T(m, n)$ on any configuration of the URM. (Thus transfer instructions are really redundant in the formulation of our URM; it is nevertheless natural and convenient to have transfer as a basic facility of the URM.)

4. Decidable predicates and problems

In mathematics a common task is to *decide* whether numbers possess a given property. For instance, the task described in (1.1) (*d*) is to decide, given numbers x, y, whether they have the property that x is a multiple of y. An algorithm for this operation would be an effective procedure that on inputs x, y gives output Yes or No. If we adopt the convention that 1 means Yes, and 0 means No, then the operation amounts to calculation of the function

$$f(x, y) = \begin{cases} 1 & \text{if } x \text{ is a multiple of } y, \\ 0 & \text{if } x \text{ is not a multiple of } y. \end{cases}$$

Thus we can say that the property or predicate 'x is a multiple of y' is *algorithmically* or *effectively decidable*, or just *decidable* if this function f is computable.

Generally, suppose that $M(x_1, x_2, \ldots, x_n)$ is an n-ary predicate of natural numbers. The *characteristic function* $c_M(\boldsymbol{x})$ (setting $\boldsymbol{x} = (x_1, \ldots, x_n)$) is given by

$$c_M(\boldsymbol{x}) = \begin{cases} 1 & \text{if } M(\boldsymbol{x}) \text{ holds,} \\ 0 & \text{if } M(\boldsymbol{x}) \text{ doesn't hold.} \end{cases}$$

4.1 *Definition*

The predicate $M(\boldsymbol{x})$ is *decidable* if the function c_M is computable; $M(\boldsymbol{x})$ is *undecidable* if $M(\boldsymbol{x})$ is not decidable.

4.2 *Examples*

The following predicates are decidable:

(*a*) '$x \neq y$': the function f of exercise 3.3 (1*c*) is the characteristic function of this predicate.

(*b*) '$x = 0$': the characteristic function is given by

$$g(x) = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{if } x \neq 0. \end{cases}$$

The following simple program computes g:

J(1, 2, 3)

J(1, 1, 4)

S(2)

T(2, 1)

(*c*) 'x is a multiple of y': it is possible to write a program for the characteristic function, but this would be somewhat lengthy and complicated. A simpler demonstration that this predicate is decidable will emerge from the next chapter, where techniques for generating more complex computable functions are developed.

Note that when discussing decidability (or undecidability) we are always concerned with the computability (or non-computability) of *total* functions.

In the context of decidability, properties or predicates are sometimes described as *problems*. Thus we might say that the problem '$x \neq y$' is decidable. In chapter 6 we will study undecidable problems.

4.3 *Exercise*

Show that the following predicates are decidable.

(*a*) '$x < y$',

(*b*) '$x \neq 3$',

(*c*) 'x is even'.

5. Computability on other domains

Since the URM handles only natural numbers, our definition of computability and decidability applies only to functions and predicates

of natural numbers. These notions are easily extended to other kinds of object (e.g. integers, polynomials, matrices, etc.) by means of coding, as follows.

A *coding* of a domain D of objects is an explicit and effective injection $\alpha: D \to \mathbb{N}$. We say that an object $d \in D$ is *coded* by the natural number $\alpha(d)$. Suppose now that f is a function from D to D; then f is naturally coded by the function f^* from $\mathbb{N}$ to $\mathbb{N}$ that maps the code of an object $d \in \mathrm{Dom}(f)$ to the code of $f(d)$. Explicitly we have

$$f^* = \alpha \circ f \circ \alpha^{-1}.$$

Now we may extend the definition of URM-computability to D by saying that f is *computable* if f^* is a computable function of natural numbers.

5.1 *Example*

Consider the domain $\mathbb{Z}$. An explicit coding is given by the function α where

$$\alpha(n) = \begin{cases} 2n & \text{if } n \geq 0, \\ -2n-1 & \text{if } n < 0. \end{cases}$$

Then α^{-1} is given by

$$\alpha^{-1}(m) = \begin{cases} \frac{1}{2}m & \text{if } m \text{ is even}, \\ -\frac{1}{2}(m+1) & \text{if } m \text{ is odd}. \end{cases}$$

Consider now the function $x-1$ on $\mathbb{Z}$; if we call this function f, then $f^*: \mathbb{N} \to \mathbb{N}$ is given by

$$f^*(x) = \begin{cases} 1 & \text{if } x = 0 \text{ (i.e. } x = \alpha(0)), \\ x-2 & \text{if } x > 0 \text{ and } x \text{ is even (i.e. } x = \alpha(n),\ n > 0), \\ x+2 & \text{if } x \text{ is odd. (i.e. } x = \alpha(n),\ n < 0). \end{cases}$$

It is a routine exercise to write a program that computes f^*; hence $x-1$ is a computable function on $\mathbb{Z}$.

The definitions of computable n-ary function on a domain D and decidable predicate on D are obtained by the obvious extension of the above idea.

5.2 *Exercises*

1. Show that the function $2x$ on $\mathbb{Z}$ is computable.
2. Show that the predicate '$x \geq 0$' is a decidable predicate on $\mathbb{Z}$.

2
Generating computable functions

In this chapter we shall see that various methods of combining computable functions give rise to other computable functions. This will enable us to show quite rapidly that many commonly occurring functions are computable, without writing a program each time – a task that would be rather laborious and tedious.

1. The basic functions

First we note that some particularly simple functions are computable; from these *basic functions* (defined in lemma 1.1 below) we shall then build more complicated computable functions using the techniques developed in subsequent sections.

1.1. *Lemma*

The following basic functions are computable:

(*a*) *the zero function* $\mathbf{0}(\mathbf{0}(x) = 0$ for all $x)$;

(*b*) *the successor function* $x+1$;

(*c*) *for each* $n \geq 1$ and $1 \leq i \leq n$, *the projection function* U_i^n *given by* $U_i^n(x_1, x_2, \ldots, x_n) = x_i$.

Proof. These functions correspond to the arithmetic instructions for the URM. Specifically, programs are as follows:

(*a*) **0**: program Z(1);

(*b*) $x+1$: program S(1);

(*c*) U_i^n: program T(i, 1). □

2. Joining programs together

In each of §§ 3–5 below we need to write programs that incorporate other programs as *subprograms* or *subroutines*. In this section we deal with some technical matters so as to make the program writing of later sections as straightforward as possible.

A simple example of program building is when we have programs P and Q, and we wish to write a program for the composite procedure: first do P, and then do Q. Our instinct is to simply write down the instructions in P followed by the instructions in Q. But there are two technical points to consider.

Suppose that $P = I_1, I_2, \ldots, I_s$. A computation under P is completed when the 'next instruction for the computation' is I_v for some $v > s$; we then require the computation under our composite program to proceed to the *first* instruction of Q. This will happen automatically if $v = s + 1$, but not otherwise. Thus for building composite programs we must confine our attention to programs that invariably stop because the next instruction is I_{s+1}. Such programs are said to be in *standard form*. Clearly it is only jump instructions that can cause a program to stop in non-standard fashion. Thus we have the following definition.

2.1. *Definition*

A program $P = I_1, I_2, \ldots, I_s$ is in *standard form* if, for every jump instruction $J(m, n, q)$ in P we have $q \leq s + 1$.

Examples. In examples 1-3.2 the programs for (a) and (c) are in standard form, whereas the program in (b) is not.

Insisting on standard form if necessary is no restriction, as we now see.

2.2. *Lemma*

For any program P there is a program P^* in standard form such that any computation under P^* is identical to the corresponding computation under P, except possibly in the manner of stopping. In particular, for any $a_1, \ldots, a_n, b$,

$$P(a_1, \ldots, a_n)\downarrow b \text{ if and only if } P^*(a_1, \ldots, a_n)\downarrow b,$$

and hence $f_P^{(n)} = f_{P^*}^{(n)}$ for every $n > 0$.

Proof. Suppose that $P = I_1, I_2, \ldots, I_s$. To obtain P^* from P simply change the jump instructions so that all jump stops occur because the jump is to I_{s+1}. Explicitly, put $P^* = I_1^*, I_2^*, \ldots, I_s^*$ where

if I_k is not a jump instruction, then $I_k^* = I_k$;

$$\text{if } I_k = J(m, n, q), \text{ then } I_k^* = \begin{cases} I_k & \text{if } q \leq s+1, \\ J(m, n, s+1) & \text{if } q > s+1. \end{cases}$$

Then clearly P^* is as required. □

Let us assume now that the programs P and Q are in standard form. The second problem when joining P and Q concerns the jump instructions in Q. A jump $J(m, n, q)$ occurring in Q commands a jump to the qth instruction of Q (if $r_m = r_n$). But the qth instruction of Q will become the $s + q$th instruction in the composite program; thus each jump $J(m, n, q)$ in Q must be modified to become $J(m, n, s + q)$ in the composite program if the sense is to be preserved.

Now without any further worry we can define the *join* or *concatenation* of two programs in standard form:

2.3. *Definition*

Let P and Q be programs of lengths s, t respectively, in standard form. The *join* or *concatenation* of P and Q, written PQ or $\begin{matrix} P \\ Q \end{matrix}$, is the program $I_1, I_2, \ldots, I_s, I_{s+1} \ldots, I_{s+t}$ where $P = I_1, \ldots, I_s$, and the instructions $I_{s+1}, \ldots, I_{s+t}$ are the instructions of Q with each jump $J(m, n, q)$ replaced by $J(m, n, s + q)$.

With this definition it is clear that the effect of PQ is as desired: any computation under PQ is identical to the corresponding computation under P followed by the computation under Q whose initial configuration is the final configuration from the computation under P.

There are two further considerations before we can proceed to the major tasks of this chapter. Suppose that we wish to compose a program Q having a given program P as a subroutine. To write Q it is often important to be able to find some registers that are unaffected by computations under P. This can be done as follows.

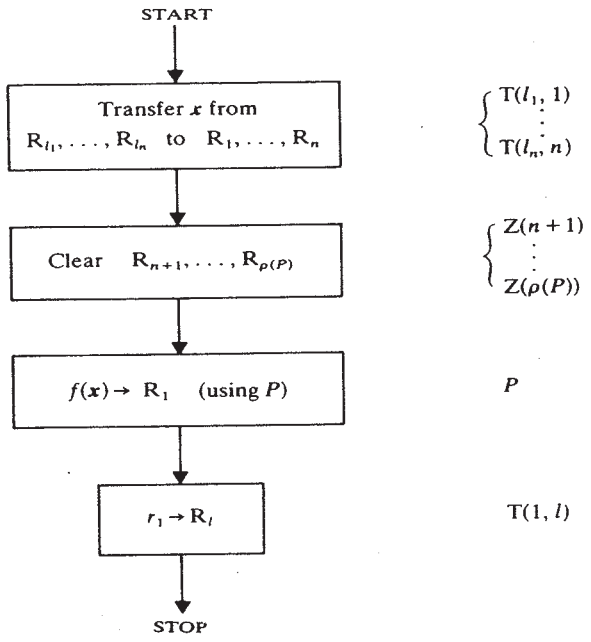
Since P is finite, there is a smallest number u such that none of the registers R_v for $v > u$ is mentioned in P; i.e. if $Z(n)$, or $S(n)$, or $T(m, n)$, or $J(m, n, q)$ is an instruction in P, then $m, n \leq u$. Clearly, during any computation under P, the contents of R_v for $v > u$ remain unaltered, and have no effect on the values of $r_1, \ldots, r_u$. Thus when writing our new program Q the registers R_v for $v > u$ can be used, for example, to store information without affecting any computation under the subroutine P. We denote the number u by $\rho(P)$.

Finally, we introduce some notation that will greatly simplify the main proofs of this chapter. Suppose that P is a program in standard form designed to compute a function $f(x_1, \ldots, x_n)$. Often when using P as a

subroutine in a larger program the inputs $x_1, \ldots, x_n$ for which $f(x_1, \ldots, x_n)$ is desired may be held in registers $R_{l_1}, \ldots, R_{l_n}$ rather than $R_1, \ldots, R_n$ as the program P requires; further, the output $f(x_1, \ldots, x_n)$ may be required for future purposes to be in some register R_l rather than the conventional R_1; and finally the working registers $R_1, \ldots, R_{\rho(P)}$ for P may contain all kinds of unwanted information. We can modify P to take account of all of these points as follows.

We write $P[l_1, \ldots, l_n \to l]$ for the program in fig. 2*a* that translates the flow diagram alongside. The program $P[l_1, \ldots, l_n \to l]$ has the effect of computing $f(r_{l_1}, \ldots, r_{l_n})$ and placing the result in R_l. Moreover, the only registers affected by this program are (at most) $R_1, R_2, \ldots, R_{\rho(P)}$ and R_l. (We have assumed in defining $P[l_1, \ldots, l_n \to l]$ that $R_{l_1}, \ldots, R_{l_n}$ are distinct from $R_1, \ldots, R_n$; this will be the case in all our uses of this notation. The reader should be able to modify the definition for situations where this is not the case.)

Fig. 2*a*. Flow diagram for $P[l_1, \ldots, l_n \to l]$.



3. Substitution

A common way of manufacturing new functions from old is to substitute functions into other functions, otherwise known as composition of functions. In the following theorem we show that when this process is applied to computable functions, the resulting functions are also computable. In short, we say that $\mathscr{C}$ is *closed* under the operation of substitution.

3.1. *Theorem*

Suppose that $f(y_1, \ldots, y_k)$ and $g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x})$ are computable functions, where $\boldsymbol{x} = (x_1, \ldots, x_n)$. Then the function $h(\boldsymbol{x})$ given by

$$h(\boldsymbol{x}) \simeq f(g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x}))$$

is computable.

(Note. $h(\boldsymbol{x})$ is defined if and only if $g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x})$ are all defined and $(g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x})) \in \mathrm{Dom}(f)$; thus, if f and $g_1, \ldots, g_k$ are all total functions, then h is total.)

Proof. Suppose that $F, G_1, \ldots, G_k$ are programs in standard form which compute $f, g_1, \ldots, g_k$ respectively. We will write a program H that embodies the following natural procedure for computing h. 'Given $\boldsymbol{x}$, use the programs $G_1, \ldots, G_k$ to compute in succession $g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x})$, making a note of these values as they are obtained. Then use the program F to compute $f(g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x}))$.'

We must take a little care to avoid losing information needed at later stages in the procedure, namely $\boldsymbol{x}$ and those values $g_i(\boldsymbol{x})$ already obtained. Putting $m = \max(n, k, \rho(F), \rho(G_1), \ldots, \rho(G_k))$, we shall begin by storing $\boldsymbol{x}$ in $R_{m+1}, \ldots, R_{m+n}$; the registers $R_{m+n+1}, \ldots, R_{m+n+k}$ will be used to store the values $g_i(\boldsymbol{x})$ as they are computed for $i = 1, 2, \ldots, k$. These registers are completely ignored by computations under $F, G_1, \ldots, G_k$. A typical configuration during computation under H will be

Storage registers

$R_1 \ldots R_m$	$R_{m+1} \ldots R_{m+n}$	R_{m+n+1}	R_{m+n+2}	$\ldots$	R_{m+n+i}			
$\ldots$	$\boldsymbol{x}$	$g_1(\boldsymbol{x})$	$g_2(\boldsymbol{x})$	$\ldots$	$g_i(\boldsymbol{x})$	0	0	$\ldots$

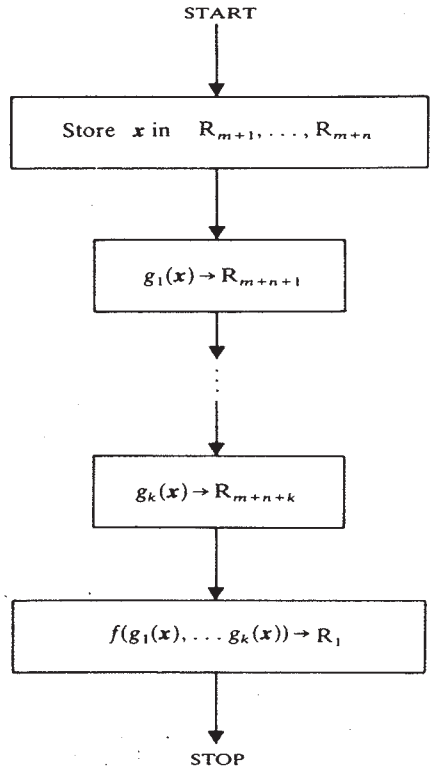
An informal flow diagram for computing h is given in fig. 2*b*. This is easily translated into the following program H that computes h:

$T(1, m+1)$
$\vdots$
$T(n, m+n)$

$G_1[m+1, m+2, \ldots, m+n \to m+n+1]$
$\vdots$
$G_k[m+1, m+2, \ldots, m+n \to m+n+k]$
$F[m+n+1, \ldots, m+n+k \to 1]$

(Recall the meaning of this notation from § 2.)

Fig. 2*b*. Substitution (theorem 3.1).



Clearly a computation $H(x)$ will stop if and only if each computation $G_i(x)$ stops $(1 \leq i \leq k)$ and the computation $F(g_1(x), \ldots, g_k(x))$ stops, which is exactly as required. □

New functions can be obtained from any given function by rearranging or identifying its variables, or by adding new dummy variables; for instance, from a function $f(y_1, y_2)$ we can obtain

$h_1(x_1, x_2) \simeq f(x_2, x_1)$ (rearrangement),
$h_2(x) \simeq f(x, x)$ (identification),
$h_3(x_1, x_2, x_3) \simeq f(x_2, x_3)$ (adding dummy variables).

The following application of theorem 3.1 shows that any of these operations (or a combination of them) transforms computable functions into computable functions.

3.2. *Theorem*

Suppose that $f(y_1, \ldots, y_k)$ *is a computable function and that* $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ *is a sequence of k of the variables* $x_1, \ldots, x_n$ *(possibly with repetitions). Then the function h given by*

$$h(x_1, \ldots, x_n) \simeq f(x_{i_1}, \ldots, x_{i_k})$$

is computable.

Proof. Writing $x = (x_1, \ldots, x_n)$ we have that

$$h(x) \simeq f(U^n_{i_1}(x), U^n_{i_2}(x), \ldots, U^n_{i_k}(x))$$

which is computable, by Lemma 1.1(*c*) and theorem 3.1. □

Using this result we can see that theorem 3.1 also holds when the functions $g_1, \ldots, g_k$ substituted into f are not necessarily functions of all of the variables $x_1, \ldots, x_n$, as in the following example.

3.3. *Example*

The function $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$ is computable; this can be deduced from the fact that $x + y$ is computable (example 1-3.2(*a*)), by substituting $x_1 + x_2$ for x, and x_3 for y in $x + y$.

Substitution combined with the principle described in the next section gives a powerful method of generating computable functions.

3.4. *Exercises*

1. Without writing any programs, show that for every $m \in \mathbb{N}$ the following functions are computable:
   (*a*) $\boldsymbol{m}$ (recall that $\boldsymbol{m}(x) = m$, for all x),
   (*b*) mx.
2. Suppose that $f(x, y)$ is computable, and $m \in \mathbb{N}$. Show that the function
   $h(x) \simeq f(x, m)$
   is computable.
3. Suppose that $g(x)$ is a total computable function. Show that the predicate $M(x, y)$ given by
   $M(x, y) \equiv$ '$g(x) = y$'
   is decidable.

4. Recursion

Recursion is a method of defining a function by specifying each of its values in terms of previously defined values, and possibly using other already defined functions.

To be precise, suppose that $f(\boldsymbol{x})$ and $g(\boldsymbol{x}, y, z)$ are functions (not necessarily total or computable). Consider the following 'definition' of a new function $h(\boldsymbol{x}, y)$:

(4.1) (i) $h(\boldsymbol{x}, 0) \simeq f(\boldsymbol{x})$,
(ii) $h(\boldsymbol{x}, y+1) \simeq g(\boldsymbol{x}, y, h(\boldsymbol{x}, y))$.

At first sight this may seem a little dubious as a definition, for in the second line it appears that h is being defined in terms of itself – a circular definition! However, with a little thought we can convince ourselves that this is a valid definition: to find the value of $h(\boldsymbol{x}, 3)$ for instance, first find $h(\boldsymbol{x}, 0)$ using (4.1)(i); then, knowing $h(\boldsymbol{x}, 0)$, use (4.1)(ii) to obtain $h(\boldsymbol{x}, 1)$; similarly, obtain $h(\boldsymbol{x}, 2)$, and finally $h(\boldsymbol{x}, 3)$ by further applications of (4.1)(ii). Thus, circularity is avoided by thinking of the values of $h(\boldsymbol{x}, y)$ as being defined one at a time, always in terms of a value already obtained.

A function h defined thus is said to be defined by *recursion* from the functions f and g; the equations 4.1 are known as *recursion equations*. Unless both f and g are total, then h as defined by (4.1) may not be total; the domain of h will satisfy the conditions

$$(\boldsymbol{x}, 0) \in \mathrm{Dom}(h) \quad \text{iff} \quad \boldsymbol{x} \in \mathrm{Dom}(f),$$
$$(\boldsymbol{x}, y+1) \in \mathrm{Dom}(h) \quad \text{iff} \quad (\boldsymbol{x}, y) \in \mathrm{Dom}(h) \text{ and } (\boldsymbol{x}, y, h(\boldsymbol{x}, y)) \in \mathrm{Dom}(g).$$

Let us summarise the above discussion in a theorem, whose proof we omit.

4.2. *Theorem*

Let $\boldsymbol{x} = (x_1, \ldots, x_n)$, and suppose that $f(\boldsymbol{x})$ and $g(\boldsymbol{x}, y, z)$ are functions; then there is a unique function $h(\boldsymbol{x}, y)$ satisfying the recursion equations

$$h(\boldsymbol{x}, 0) \simeq f(\boldsymbol{x}),$$
$$h(\boldsymbol{x}, y+1) \simeq g(\boldsymbol{x}, y, h(\boldsymbol{x}, y)).$$

Note. When $n = 0$ (i.e. the parameters $\boldsymbol{x}$ do not appear) the recursion equations take the form

$$h(0) = a,$$
$$h(y+1) \simeq g(y, h(y)),$$

where $a \in \mathbb{N}$.

4.3. *Examples*

(*a*) Addition: for any x, y we have

$$x + 0 = x,$$
$$x + (y+1) = (x+y) + 1.$$

Thus addition (i.e. the function $h(x, y) = x + y$) is defined by recursion from the functions $f(x) = x$ and $g(x, y, z) = z + 1$.

(*b*) $y!$: with the convention that $0! = 1$, we have that

$$0! = 1,$$
$$(y+1)! = y!(y+1).$$

Thus the function $y!$ is defined by recursion from 1 and the function $g(y, z) = z(y+1)$.

There are forms of definition by recursion that are more general than the one we have discussed; we shall encounter an example of this in § 5, and a fuller discussion of this topic is included in chapter 10. In contexts

where general kinds of recursion are being considered, the particularly simple kind of definition given by (4.1) is called *primitive recursion.*

Many commonly occurring functions have easy definitions by (primitive) recursion, so for establishing computability the next theorem is extremely useful. Briefly, it shows that $\mathscr{C}$ is closed under definition by recursion.

4.4. *Theorem*

Suppose that $f(\boldsymbol{x})$ and $g(\boldsymbol{x}, y, z)$ are computable functions, where $\boldsymbol{x} = (x_1, \ldots, x_n)$; then the function $h(\boldsymbol{x}, y)$ obtained from f and g by recursion is computable.

Proof. Let F and G be programs in standard form which compute the functions $f(\boldsymbol{x})$ and $g(\boldsymbol{x}, y, z)$. We will devise a program H for the function $h(\boldsymbol{x}, y)$ given by the recursion equations 4.1. Given an initial configuration $x_1, \ldots, x_n, y, 0, 0, 0 \ldots$ H will first compute $h(\boldsymbol{x}, 0)$ (using F); then, if $y \neq 0$, H will use G to compute successively $h(\boldsymbol{x}, 1)$, $h(\boldsymbol{x}, 2), \ldots, h(\boldsymbol{x}, y)$, and then stop.

Let $m = \max(n+2, \rho(F), \rho(G))$; we begin by storing $\boldsymbol{x}, y$ in $R_{m+1}, \ldots, R_{m+n+1}$; the next two registers will be used to store the current value of the numbers k and $h(\boldsymbol{x}, k)$ for $k = 0, 1, 2, \ldots, y$. Writing t for $m+n$, a typical configuration during the procedure will thus be

Storage registers

	$R_1 \ldots R_m$	$R_{m+1} \ldots R_t$	R_{t+1}	R_{t+2}	R_{t+3}	
(*)	$\ldots$	$\boldsymbol{x}$	y	k	$h(\boldsymbol{x}, k)$	$\ldots$

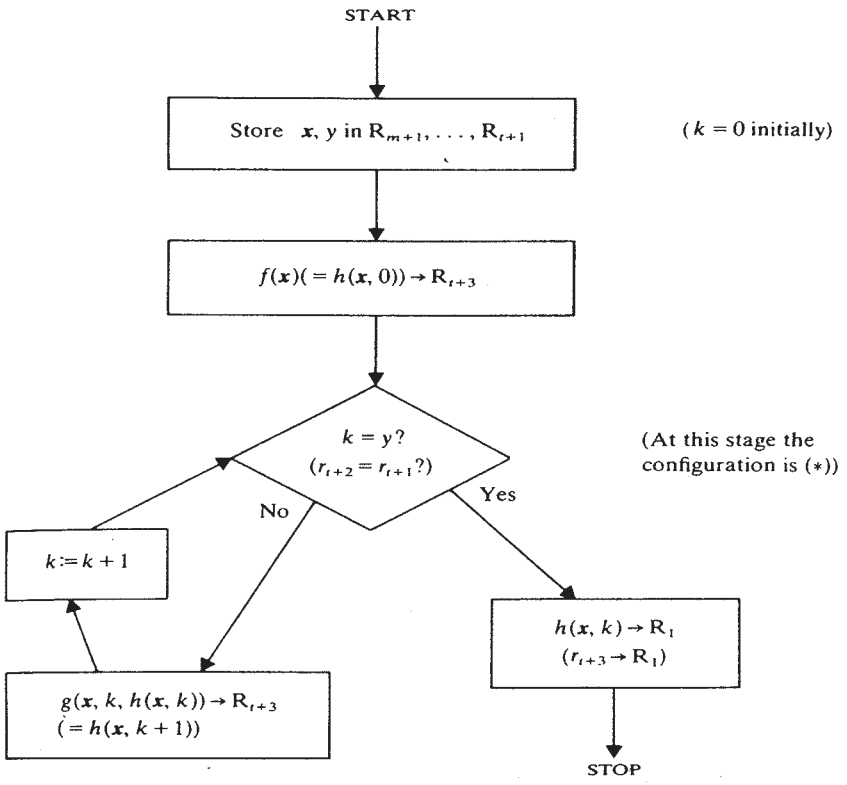
with $k = 0$ initially.

An informal flow diagram for the procedure is given in fig. 2*c*. This flow diagram translates easily into the following program H that computes h:

$$
\begin{array}{ll}
 & \mathrm{T}(1, m+1) \\
 & \vdots \\
 & \mathrm{T}(n+1, m+n+1) \\
 & F[1, 2, \ldots, n \to t+3] \\
I_q & \mathrm{J}(t+2, t+1, p) \\
 & G[m+1, \ldots, m+n, t+2, t+3 \to t+3] \\
 & \mathrm{S}(t+2) \\
 & \mathrm{J}(1, 1, q) \\
I_p & \mathrm{T}(t+3, 1)
\end{array}
$$

Hence h is computable. □

Fig. 2*c*. Recursion (theorem 4.4).



We now proceed to use theorems 3.1 and 4.4 to compile a collection of computable functions. The collection is potentially infinite, so our choice is influenced by (i) the needs of subsequent development of our theory, and (ii) the desire to give credence to the thesis that all functions that we would regard as computable in the informal sense are indeed URM-computable. For reasons which will become apparent later we shall include some functions such as $x + y$ and $x \dot{-} 1$ for which we have already written programs.

We shall use repeatedly the fact that,by theorem 3.2, in a definition by recursion such as (4.1), the computable functions f and g need not be functions of all of the named variables for the function h to be computable.

4.5. *Theorem*

The following functions are computable. (Proofs are given as the functions are listed.)

(a) $x+y$ *Proof.* Example 4.3(a) gives a definition by recursion from the computable functions x and $z+1$.

(b) xy *Proof.* $x0=0$,
$x(y+1)=xy+x$,
is a definition by recursion from the computable functions $\mathbf{0}(x)$ and $z+x$.

(c) x^y *Proof.* $x^0=1$,
$x^{y+1}=x^y x$; by recursion and (b).

(d) $x \dot{-} 1$ *Proof.* $0 \dot{-} 1=0$,
$(x+1) \dot{-} 1=x$; by recursion.

(e) $x \dot{-} y=\begin{cases} x-y & \text{if } x \geq y, \\ 0 & \text{otherwise.} \end{cases}$ (*cut-off subtraction*)
Proof. $x \dot{-} 0=x$,
$x \dot{-} (y+1)=(x \dot{-} y) \dot{-} 1$; by recursion and ($d$).

(f) $\mathrm{sg}(x)=\begin{cases} 0 & \text{if } x=0, \\ 1 & \text{if } x \neq 0. \end{cases}$ (cf. exercises 1-3.3(1a))
Proof. $\mathrm{sg}(0)=0$,
$\mathrm{sg}(x+1)=1$; by recursion.

(g) $\overline{\mathrm{sg}}(x)=\begin{cases} 1 & \text{if } x=0, \\ 0 & \text{if } x \neq 0. \end{cases}$ (cf. example 1-4.2(b))
Proof. $\overline{\mathrm{sg}}(x)=1 \dot{-} \mathrm{sg}(x)$; by substitution, ($e$) and ($f$).

(h) $|x-y|$ *Proof.* $|x-y|=(x \dot{-} y)+(y \dot{-} x)$; by substitution, ($a$) and ($e$).

(i) $x!$ *Proof.* Example 4.3(b) gives a definition by recursion from computable functions.

(j) $\min(x, y)=$ *minimum of x and y.*
Proof. $\min(x, y)=x \dot{-} (x \dot{-} y)$; by substitution.

(k) $\max(x, y)=$ *maximum of x and y.*
Proof. $\max(x, y)=x+(y \dot{-} x)$; by substitution.

(l) $\mathrm{rm}(x, y)=$ *remainder when y is divided by x (to obtain a total function, we adopt the convention* $\mathrm{rm}(0, y)=y$).
Proof. We have
$$\mathrm{rm}(x, y+1)=\begin{cases} \mathrm{rm}(x, y)+1 & \text{if } \mathrm{rm}(x, y)+1 \neq x, \\ 0 & \text{if } \mathrm{rm}(x, y)+1=x. \end{cases}$$
This gives the following definition by recursion:
$\mathrm{rm}(x, 0)=0$,
$\mathrm{rm}(x, y+1)=(\mathrm{rm}(x, y)+1)\,\mathrm{sg}(|x-(\mathrm{rm}(x, y)+1)|)$.

The second equation can be written
$\mathrm{rm}(x, y+1)=g(x, \mathrm{rm}(x, y))$
where $g(x, z)=(z+1)\,\mathrm{sg}(|x-(z+1)|)$; and g is computable by several applications of substitution. Hence $\mathrm{rm}(x, y)$ is computable.

(m) $\mathrm{qt}(x, y)=$ *quotient when y is divided by x (to obtain a total function we define* $\mathrm{qt}(0, y)=0$).
Proof. Since
$$\mathrm{qt}(x, y+1)=\begin{cases} \mathrm{qt}(x, y)+1 & \text{if } \mathrm{rm}(x, y)+1=x, \\ \mathrm{qt}(x, y) & \text{if } \mathrm{rm}(x, y)+1 \neq x, \end{cases}$$
we have the following definition by recursion from computable functions:
$\mathrm{qt}(x, 0)=0$,
$\mathrm{qt}(x, y+1)=\mathrm{qt}(x, y)+\overline{\mathrm{sg}}(|x-(\mathrm{rm}(x, y)+1)|)$.

(n) $\mathrm{div}(x, y)=\begin{cases} 1 & \text{if } x|y \text{ (} x \text{ divides } y\text{)}, \\ 0 & \text{if } x \nmid y. \end{cases}$
(*We adopt the convention that* $0|0$ *but* $0 \nmid y$ *if* $y \neq 0$.) *Hence $x|y$ is decidable* (recall definition 1-4.1).
Proof. $\mathrm{div}(x, y)=\overline{\mathrm{sg}}(\mathrm{rm}(x, y))$, computable by substitution. □

The following are useful corollaries involving decidable predicates.

4.6. *Corollary* (Definition by cases)

Suppose that $f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x})$ are total computable functions, and $M_1(\boldsymbol{x}), \ldots, M_k(\boldsymbol{x})$ are decidable predicates, such that for every $\boldsymbol{x}$ exactly one of $M_1(\boldsymbol{x}), \ldots, M_k(\boldsymbol{x})$ holds. Then the function $g(\boldsymbol{x})$ given by
$$g(\boldsymbol{x})=\begin{cases} f_1(\boldsymbol{x}) & \text{if } M_1(\boldsymbol{x}) \text{ holds}, \\ f_2(\boldsymbol{x}) & \text{if } M_2(\boldsymbol{x}) \text{ holds}, \\ \vdots & \vdots \\ f_k(\boldsymbol{x}) & \text{if } M_k(\boldsymbol{x}) \text{ holds}, \end{cases}$$
is computable.

Proof. $g(\boldsymbol{x})=c_{M_1}(\boldsymbol{x})f_1(\boldsymbol{x})+\ldots+c_{M_k}(\boldsymbol{x})f_k(\boldsymbol{x})$, computable by substitution using addition and multiplication. □

4.7. *Corollary* (Algebra of decidability)

Suppose that $M(\boldsymbol{x})$ and $Q(\boldsymbol{x})$ are decidable predicates; then the following are also decidable.

(a) '*not $M(\boldsymbol{x})$*'

(*b*) '$M(\boldsymbol{x})$ *and* $Q(\boldsymbol{x})$'
(*c*) '$M(\boldsymbol{x})$ *or* $Q(\boldsymbol{x})$'

Proof. The characteristic functions of these predicates are as follows:
(*a*) 'not $M(\boldsymbol{x})$': $1 \dot{-} c_M(\boldsymbol{x})$,
(*b*) '$M(\boldsymbol{x})$ and $Q(\boldsymbol{x})$': $c_M(\boldsymbol{x})\, c_Q(\boldsymbol{x})$,
(*c*) '$M(\boldsymbol{x})$ or $Q(\boldsymbol{x})$': $\max(c_M(\boldsymbol{x}), c_Q(\boldsymbol{x}))$ (where we take 'or' in the inclusive sense).
Each of the functions on the right is computable provided c_M and c_Q are, by substitution in functions from theorem 4.5. □

Recursion can be used to establish the computability of functions obtained by other function building techniques, which we now describe. First, we introduce some notation.

Suppose that $f(\boldsymbol{x}, z)$ is any function; the *bounded sum* $\sum_{z<y} f(\boldsymbol{x}, z)$ and the *bounded product* $\prod_{z<y} f(\boldsymbol{x}, z)$ are the functions of $\boldsymbol{x}$, y given by the following recursion equations.

$$(4.8)\qquad \begin{cases} \sum\limits_{z<0} f(\boldsymbol{x}, z) = 0, \\ \sum\limits_{z<y+1} f(\boldsymbol{x}, z) = \sum\limits_{z<y} f(\boldsymbol{x}, z) + f(\boldsymbol{x}, y), \end{cases}$$

$$(4.9)\qquad \begin{cases} \prod\limits_{z<0} f(\boldsymbol{x}, z) = 1, \\ \prod\limits_{z<y+1} f(\boldsymbol{x}, z) = \left(\prod\limits_{z<y} f(\boldsymbol{x}, z)\right) \cdot f(\boldsymbol{x}, y). \end{cases}$$

4.10. *Theorem*
Suppose that $f(\boldsymbol{x}, z)$ is a total computable function; then the functions $\sum_{z<y} f(\boldsymbol{x}, z)$ and $\prod_{z<y} f(\boldsymbol{x}, z)$ are computable.

Proof. The equations 4.8 and 4.9 are definitions by recursion from computable functions. □

It is easily seen that if the bound on z in a bounded sum or product is given by any computable function, the result is still computable, as follows.

4.11. *Corollary*
Suppose that $f(\boldsymbol{x}, z)$ and $k(\boldsymbol{x}, \boldsymbol{w})$ are total computable functions; then so are the functions $\sum_{z<k(\boldsymbol{x},\boldsymbol{w})} f(\boldsymbol{x}, z)$ and $\prod_{z<k(\boldsymbol{x},\boldsymbol{w})} f(\boldsymbol{x}, z)$ (both functions of $\boldsymbol{x}$, $\boldsymbol{w}$).

Proof. By substitution of $k(\boldsymbol{x}, \boldsymbol{w})$ for y in the bounded sum $\sum_{z<y} f(\boldsymbol{x}, z)$ and the bounded product $\prod_{z<y} f(\boldsymbol{x}, z)$. □

We now describe another useful function building technique which yields computable functions. We write

$$\mu z{<}y(\ldots)$$

for 'the least z less than y such that ...'. In order that this expression be totally defined, we give it the value y when no such z exists. Then, for example, given a function $f(\boldsymbol{x}, z)$ we can define a new function g by

$$g(\boldsymbol{x}, y) = \mu z{<}y(f(\boldsymbol{x}, z) = 0)$$
$$= \begin{cases} \text{the least } z < y & \text{such that } f(\boldsymbol{x}, z) = 0, \text{ if such a } z \text{ exists;} \\ y & \text{if there is no such } z. \end{cases}$$

The operator $\mu z{<}y$ is called a *bounded minimalisation operator*, or *bounded μ-operator.*

4.12. *Theorem*
Suppose that $f(\boldsymbol{x}, y)$ is a total computable function; then so is the function $\mu z{<}y(f(\boldsymbol{x}, z) = 0)$.

Proof. Consider the function

$$h(\boldsymbol{x}, v) = \prod_{u \le v} \mathrm{sg}(f(\boldsymbol{x}, u)),$$

which is computable by corollary 4.11. For a given $\boldsymbol{x}$, y, suppose that $z_0 = \mu z{<}y(f(\boldsymbol{x}, z) = 0)$. It is easy to see that

if $v < z_0$, then $h(\boldsymbol{x}, v) = 1$;
if $z_0 \le v < y$, then $h(\boldsymbol{x}, v) = 0$.

Thus

$$z_0 = \text{the number of } v\text{s less than } y \text{ such that } h(\boldsymbol{x}, v) = 1,$$
$$= \sum_{v<y} h(\boldsymbol{x}, v).$$

Hence

$$\mu z{<}y(f(\boldsymbol{x}, z) = 0) = \sum_{v<y} \left(\prod_{u \le v} \mathrm{sg}(f(\boldsymbol{x}, u))\right),$$

and is computable by theorem 4.10. □

As with bounded sums and products, the bound in bounded minimalisation can be given by any computable function:

4.13. *Corollary*

If $f(x, z)$ and $k(x, w)$ are total computable functions, then so is the function

$$\mu z<k(x, w)(f(x, z)=0).$$

Proof. By substitution of $k(x, w)$ for y in the computable function $\mu z<y(f(x, z)=0)$. □

Theorems 4.10 and 4.12 give us the following applications involving decidable predicates.

4.14. *Corollary*

Suppose that $R(x, y)$ is a decidable predicate: then

(*a*) *the function $f(x, y)=\mu z<y\, R(x, z)$ is computable.*

(*b*) *the following predicates are decidable:*

(i) $M_1(x, y)\equiv\forall z<y\, R(x, z)$,

(ii) $M_2(x, y)\equiv\exists z<y\, R(x, z)$.

Proof.

(*a*) $f(x, y)=\mu z<y(\overline{\mathrm{sg}}(c_R(x, z))=0)$.

(*b*) (i) $c_{M_1}(x, y)=\prod_{z<y} c_R(x, z)$.

(ii) $M_2(x, y)\equiv$ not $(\forall z<y$ (not $R(x, z)))$

which is decidable by (*b*)(i) and 4.7(*a*). □

Note. As in 4.11 and 4.13, the bound on z in this corollary could be any total computable function.

We now use the above techniques to enlarge our collection of particular computable functions and decidable properties.

4.15. *Theorem*

The following functions are computable.

(*a*) $\mathrm{D}(x)=$ *the number of divisors of* x (*convention:* $\mathrm{D}(0)=1$),

(*b*) $\mathrm{Pr}(x)=\begin{cases}1 & \text{if } x \text{ is prime,}\\ 0 & \text{if } x \text{ is not prime}\end{cases}$

(*i.e. 'x is prime' is decidable*).

(*c*) $\mathrm{p}_x=$ *the* x*th prime number* (*as a convention we set* $\mathrm{p}_0=0$, *then* $\mathrm{p}_1=2$, $\mathrm{p}_2=3$, *etc.*)

(*d*) $(x)_y=\begin{cases}\text{the exponent of } \mathrm{p}_y \text{ in the prime factorisation of } x, \text{ for } x, y>0,\\ 0 \quad \text{if } x=0 \text{ or } y=0.\end{cases}$

Proof.

(*a*) $\mathrm{D}(x)=\sum_{y\le x}\mathrm{div}(y, x)$ (where div is as in theorem 4.5(*n*)).

(*b*) $\mathrm{Pr}(x)=\begin{cases}1 & \text{if } \mathrm{D}(x)=2 \text{ (i.e. } x>1 \text{ and the only divisors of } x\\ & \text{are 1 and } x),\\ 0 & \text{otherwise}\end{cases}$

$=\overline{\mathrm{sg}}(|\mathrm{D}(x)-2|)$.

(*c*) $\mathrm{p}_0=0$,

$\mathrm{p}_{x+1}=\mu z\le(\mathrm{p}_x!+1)(z>\mathrm{p}_x$ and z is prime),

which is a definition by recursion; the predicate '$z>y$ and z is prime' is decidable, so using corollary 4.14 (and the note following) we have a computable function.

(*d*) $(x)_y=\mu z<x(\mathrm{p}_y^{z+1}\nmid x)$, which is computable since the predicate '$\mathrm{p}_y^{z+1}\nmid x$' is decidable. □

Note. The function $(x)_y$ is needed in the following kind of situation. A sequence $s=(a_1, a_2, a_3, \ldots, a_n)$ from $\mathbb{N}$ can be coded by the single number $b=\mathrm{p}_1^{a_1+1}\mathrm{p}_2^{a_2+1}\ldots\mathrm{p}_n^{a_n+1}$; then the length n of s and the numbers a_i can be recovered effectively from b as follows:

$$n=\mu z<b((b)_{z+1}=0),$$

$$a_i=(b)_i\dot{-}1 \text{ for } 1\le i\le n.$$

Alternative ways of coding pairs and sequences are indicated in exercises 4.16 (2, 5) below.

4.16. *Exercises*

1. Show that the following functions are computable:
   (*a*) Any polynomial function $a_0+a_1x+\ldots+a_nx^n$, where $a_0, a_1, \ldots, a_n\in\mathbb{N}$,
   (*b*) $[\sqrt{x}]$,
   (*c*) $\mathrm{LCM}(x, y)=$ the least common multiple of x and y,
   (*d*) $\mathrm{HCF}(x, y)=$ the highest common factor of x and y,
   (*e*) $f(x)=$ number of prime divisors of x,
   (*f*) $\phi(x)=$ the number of positive integers less than x which are relatively prime to x. (*Euler's function*) (We say that x, y are *relatively prime* if $\mathrm{HCF}(x, y)=1$.)
2. Let $\pi(x, y)=2^x(2y+1)-1$. Show that π is a computable bijection from $\mathbb{N}^2$ to $\mathbb{N}$, and that the functions π_1, π_2 such that $\pi(\pi_1(z), \pi_2(z))=z$ for all z are computable.
3. Suppose $f(x)$ is defined by
   $f(0)=1$,

$f(1) = 1$,
$f(x+2) = f(x) + f(x+1)$.
($f(x)$ is the *Fibonacci* sequence.)
Show that f is computable. (*Hint:* first show that the function $g(x) = 2^{f(x)}3^{f(x+1)}$ is computable, using recursion.)

4. Show that the following problems are decidable:
   (*a*) x is odd,
   (*b*) x is a power of a prime number,
   (*c*) x is a perfect cube.

5. Any number $x \in \mathbb{N}$ has a unique expression as
   (1) $x = \sum_{i=0}^{\infty} \alpha_i 2^i$, with $\alpha_i = 0$ or 1, all i. Hence, if $x > 0$, there are unique expressions for x in the forms
   (2) $x = 2^{b_1} + 2^{b_2} + \ldots + 2^{b_l}$, with $0 \leq b_1 < b_2 < \ldots < b_l$ and $l \geq 1$.
   and
   (3) $x = 2^{a_1} + 2^{a_1+a_2+1} + \ldots + 2^{a_1+a_2+\ldots+a_k+k-1}$
   Putting
   $\alpha(i, x) = \alpha_i$ as in the expression (1);

$$l(x) = \begin{cases} l \text{ as in (2)}, & \text{if } x > 0, \\ 0 & \text{otherwise;} \end{cases}$$

$$b(i, x) = \begin{cases} b_i, \text{ as in (2)}, & \text{if } x > 0 \text{ and } 1 \leq i \leq l, \\ 0 & \text{otherwise;} \end{cases}$$

$$a(i, x) = \begin{cases} a_i, \text{ as in (3)}, & \text{if } x > 0 \text{ and } 1 \leq i \leq l, \\ 0 & \text{otherwise;} \end{cases}$$

   show that each of the functions α, l, b, a is computable. (The expression (3) is a way of regarding x as coding the sequence $(a_1, a_2, \ldots, a_l)$ of numbers, and will be used in chapter 5.)

5. Minimalisation

In the previous section we have seen that a large collection of functions can be shown to be computable using the operations of substitution and recursion, and operations derived from these. There is a third important operation which generates further computable functions, namely *unbounded minimalisation*, or just *minimalisation*, which we now describe.

Suppose that $f(\boldsymbol{x}, y)$ is a function (not necessarily total) and we wish to define a function $g(\boldsymbol{x})$ by

$g(\boldsymbol{x})$ = the least y such that $f(\boldsymbol{x}, y) = 0$,

in such a way that if f is computable then so is g. Two problems can arise. First, for some $\boldsymbol{x}$ there may not be any y such that $f(\boldsymbol{x}, y) = 0$. Second, assuming that f is computable, consider the following natural algorithm for computing $g(\boldsymbol{x})$. 'Compute $f(\boldsymbol{x}, 0)$, $f(\boldsymbol{x}, 1)$, ... until y is found such that $f(\boldsymbol{x}, y) = 0$'. This procedure may not terminate if f is not total, even if such a y exists; for instance, if $f(\boldsymbol{x}, 0)$ is undefined but $f(\boldsymbol{x}, 1) = 0$.

Thus we are led to the following definition of the *minimalisation operator* μ, which yields computable functions from computable functions.

5.1. *Definition*
For any function $f(\boldsymbol{x}, y)$

$$\mu y(f(\boldsymbol{x}, y) = 0) = \begin{cases} \text{the least } y \text{ such that} \\ \quad \text{(i) } f(\boldsymbol{x}, z) \text{ is defined, all } z \leq y, \text{ and} \\ \quad \text{(ii) } f(\boldsymbol{x}, y) = 0, & \text{if such a } y \text{ exists,} \\ \text{undefined,} & \text{if there is no such } y. \end{cases}$$

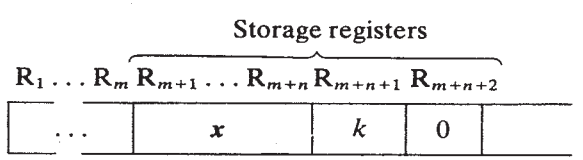
$\mu y(\ldots)$ is read 'the least y such that ...'. This operator is sometimes called simply the *μ-operator.*

The next theorem shows that $\mathscr{C}$ is closed under minimalisation.

5.2. *Theorem*
Suppose that $f(\boldsymbol{x}, y)$ is computable; then so is the function $g(\boldsymbol{x}) = \mu y(f(\boldsymbol{x}, y) = 0)$.

Proof. Suppose that $\boldsymbol{x} = (x_1, \ldots, x_n)$ and that F is a program in standard form that computes the function $f(\boldsymbol{x}, y)$. Let $m = \max(n+1, \rho(F))$. We write a program G that embodies the natural algorithm for g: for $k = 0, 1, 2 \ldots$, compute $f(\boldsymbol{x}, k)$ until a value of k is found such that $f(\boldsymbol{x}, k) = 0$; this value of k is the required output.

The value of $\boldsymbol{x}$ and the current value of k will be stored in registers $R_{m+1}, \ldots, R_{m+n+1}$ before computing $f(\boldsymbol{x}, k)$: thus the typical configuration will be



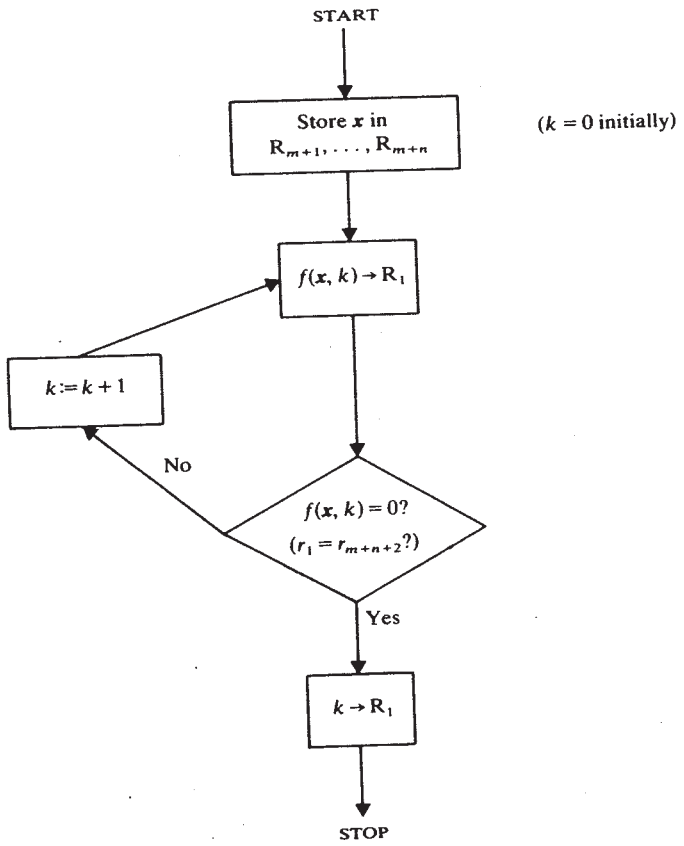
with $k = 0$ initially. Note that r_{m+n+2} is always 0.

A flow diagram that carries out the above procedure for g is given in fig. 2*d*. This translates easily into the following program G for g:

$$
\begin{array}{ll}
 & \mathrm{T}(1, m+1) \\
 & \vdots \\
 & \mathrm{T}(n, m+n) \\
I_p & F[m+1, m+2, \ldots, m+n+1 \to 1] \\
 & \mathrm{J}(1, m+n+2, q) \\
 & \mathrm{S}(m+n+1) \\
 & \mathrm{J}(1, 1, p) \\
I_q & \mathrm{T}(m+n+1, 1)
\end{array}
$$

(I_p is the first instruction of the subroutine $F[m+1, m+2, \ldots \to 1]$.) □

Fig. 2*d*. Minimalisation (theorem 5.2).



5.3. *Corollary*
Suppose that $R(x, y)$ is a decidable predicate; then the function

$$g(x) = \mu y\, R(x, y) = \begin{cases} \text{the least } y \text{ such that } R(x, y) \text{ holds}, & \text{if there is such a } y, \\ \text{undefined} & \text{otherwise}, \end{cases}$$

is computable

Proof. $g(x) = \mu y(\overline{\mathrm{sg}}(c_R(x, y)) = 0)$. □

In view of this corollary, the μ-operator is often called a *search operator*. Given a decidable predicate $R(x, y)$ the function $g(x)$ searches for a y such that $R(x, y)$ holds, and moreover, finds the least such y if there is one.

The μ-operator may generate a non-total computable function from a total computable function; for instance, putting $f(x, y) = |x - y^2|$, and $g(x) \simeq \mu y(f(x, y) = 0)$, we have that g is the non-total function

$$g(x) = \begin{cases} \sqrt{x} & \text{if } x \text{ is a perfect square}, \\ \text{undefined} & \text{otherwise}. \end{cases}$$

Thus, in a trivial sense, using the μ-operator together with substitution and recursion, we can generate from the basic functions more functions than can be obtained using only substitution and recursion (since these operations always yield total functions from total functions). There are also, however, *total* functions for which the use of the μ-operator is essential. Example 5.5 below gives one such function; we present another example in chapter 5. Thus we see that, in a strong sense, minimalisation, unlike bounded minimalisation, cannot be defined in terms of substitution and recursion. It turns out, nevertheless, that most commonly occurring computable total functions can be built up from the basic functions using substitution and recursion only: such functions are called *primitive recursive*, and are discussed further in chapter 3 § 3. In practice, of course, we might establish the computability of these functions by what amounts to a non-essential use of minimalisation, if this makes the task easier.

5.4. *Exercises*

1. Suppose that $f(x)$ is a total injective computable function; prove that f^{-1} is computable.
2. Suppose that $p(x)$ is a polynomial with integer coefficients; show that the function

$f(a)$ = least non-negative integral root of $p(x)-a (a \in \mathbb{N})$ is computable ($f(a)$ is undefined if there is no such root).

3. Show that the function

$$f(x, y) = \begin{cases} x/y & \text{if } y \neq 0 \text{ and } y \mid x, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

is computable.

We conclude this chapter with an example of a function that makes essential use of the μ-operator; it also shows how this operator can be used not only to search for a single number possessing a given property, but to search for finite sequences or sets of numbers, or other objects coded by a single number. The function is a modification by Péter of an example due to Ackermann, after whom it is named. It is rather more complicated than any function we have considered so far.

5.5. *Example* (The Ackermann function)

The function $\psi(x, y)$ given by the following equations is computable:

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

This definition involves a kind of double recursion that is stronger than the primitive recursion discussed in § 3. To see, nevertheless, that these equations do unambiguously define a function, notice that any value $\psi(x, y)$ $(x > 0)$ is defined in terms of 'earlier' values $\psi(x_1, y_1)$ with $x_1 < x$ or $x_1 = x$ and $y_1 < y$. In fact, $\psi(x, y)$ can be obtained by using only a *finite* number of such earlier values: this is easily established by induction on x and y. Hence ψ is computable in the informal sense. For instance, it is easy to calculate that $\psi(1, 1) = 3$ and $\psi(2, 1) = 5$.

To show rigorously that ψ is computable is quite difficult. We sketch a proof using the idea of a *suitable* set of triples S. The essential property of a suitable set S (defined below) is that if $(x, y, z) \in S$, then

(5.6) (i) $z = \psi(x, y)$,

(ii) S contains all the earlier triples $(x_1, y_1, \psi(x_1, y_1))$ that are needed to calculate $\psi(x, y)$.

Definition

A finite set of triples S is said to be *suitable* if the following conditions are satisfied:

(a) if $(0, y, z) \in S$ then $z = y + 1$,

(b) if $(x+1, 0, z) \in S$ then $(x, 1, z) \in S$,

(c) if $(x+1, y+1, z) \in S$ then there is u such that $(x+1, y, u) \in S$ and $(x, u, z) \in S$.

These three conditions correspond to the three clauses in the definition of ψ: for instance, (a) corresponds to the statement: if $z = \psi(0, y)$, then $z = y + 1$; (c) corresponds to the statement: if $z = \psi(x+1, y+1)$, then there is u such that $u = \psi(x+1, y)$ and $z = \psi(x, u)$.

The definition of a suitable set S ensures that (5.6) is satisfied. Moreover, for any particular pair of numbers (m, n) there is a suitable set S such that $(m, n, \psi(m, n)) \in S$; for example, let S be the set of triples $(x, y, \psi(x, y))$ that are used in the calculation of $\psi(m, n)$.

Now a triple (x, y, z) can be coded by the single positive number $u = 2^x 3^y 5^z$; a finite set of positive numbers $\{u_1, \ldots, u_k\}$ can be coded by the single number $p_{u_1} p_{u_2} \ldots p_{u_k}$. Hence a finite set of triples can be coded by a single number v say. Let S_v denote the set of triples coded by the number v. Then we have

$$(x, y, z) \in S_v \Leftrightarrow p_{2^x 3^y 5^z} \text{ divides } v,$$

so '$(x, y, z) \in S_v$' is a decidable predicate of x, y, z, v; and if it holds, then $x, y, z < v$. Hence, using the techniques and functions of earlier sections we can show that the following predicate is decidable:

$$R(x, y, v) \equiv \text{'}v \text{ is the code number of a suitable set of triples and } \exists z < v\ ((x, y, z) \in S_v)\text{.'}$$

Thus the function

$$f(x, y) = \mu v R(x, y, v)$$

is a computable function that searches for the code of a suitable set containing (x, y, z) for some z. Hence

$$\psi(x, y) = \mu z ((x, y, z) \in S_{f(x,y)})$$

which shows that ψ is computable.

A more sophisticated proof that ψ is computable will be given in chapter 10 as an application of more advanced theoretical results.

We do not prove here that ψ cannot be shown to be computable using substitution and recursion alone. This matter is further discussed in § 3 of the next chapter.

3
Other approaches to computability: Church's thesis

Over the past fifty years there have been many proposals for a precise mathematical characterisation of the intuitive idea of effective computability. The URM approach is one of the more recent of these. In this chapter we pause in our investigation of URM-computability itself to consider two related questions.

1. How do the many different approaches to the characterisation of computability compare with each other, and in particular with URM-computability?
2. How well do these approaches (particularly the URM approach) characterise the informal idea of effective computability?

The first question will be discussed in §§ 1–6; the second will be taken up in § 7. The reader interested only in the technical development of the theory in this book may omit §§ 3–6; none of the development in later chapters depends on these sections.

1. Other approaches to computability

The following are some of the alternative characterisations that have been proposed:

(*a*) *Gödel–Herbrand–Kleene* (1936). General recursive functions defined by means of an equation calculus. (Kleene [1952], Mendelson [1964].)

(*b*) *Church* (1936). λ-definable functions. (Church [1936] or [1941].)

(*c*) *Gödel–Kleene* (1936). μ-recursive functions and partial recursive functions (§ 2 of this chapter.).

(*d*) *Turing* (1936). Functions computable by finite machines known as Turing machines. (Turing [1936]; § 4 of this chapter.)

(*e*) *Post* (1943). Functions defined from canonical deduction systems. (Post [1943], Minsky [1967]; § 5 of this chapter.)

(*f*) *Markov* (1951). Functions given by certain algorithms over a finite alphabet. (Markov [1954], Mendelson [1964]; § 5 of this chapter.)

(*g*) *Shepherdson–Sturgis* (1963). URM-computable functions. (Shepherdson & Sturgis [1963].)

There is great diversity among these various approaches; each has its own rationale for being considered a plausible characterisation of computability. The remarkable result of investigation by many researchers is the following:

1.1. *The Fundamental result*

Each of the above proposals for a characterisation of the notion of effective computability gives rise to the same class of functions, the class that we have denoted $\mathscr{C}$.

Thus we have the simplest possible answer to the first question posed above. Before discussing the second question, we shall examine briefly the approaches of Gödel–Kleene, Turing, Post and Markov, mentioned above, and we will sketch some of the proofs of the equivalence of these with the URM approach. The reader interested to discover full details of these and other approaches, and proofs of all the equivalences in the Fundamental result, may consult the references indicated.

2. Partial recursive functions (Gödel–Kleene)

2.1. *Definition*

The class $\mathscr{R}$ of *partial recursive functions* is the smallest class of partial functions that contains the basic functions $\mathbf{0}$, $x+1$, U_i^n (lemma 2-1.1) and is closed under the operations of substitution, recursion and minimalisation. (Equivalently, $\mathscr{R}$ is the class of partial functions that can be built up from the basic functions by a finite number of operations of substitution, recursion or minimalisation.)

Note that in the definition of the class $\mathscr{R}$, no restriction is placed on the use of the μ-operator, so that $\mathscr{R}$ contains non-total functions. Gödel and Kleene originally confined their attention to *total* functions; the class of functions first considered was the class $\mathscr{R}_0$ of *μ-recursive functions*, defined like $\mathscr{R}$ above, except that applications of the μ-operator are allowed only if a *total* function results. Thus $\mathscr{R}_0$ is a class of total functions, and clearly $\mathscr{R}_0 \subseteq \mathscr{R}$. In fact, $\mathscr{R}_0$ contains all of the total

functions that are in $\mathscr{R}$, although this is not immediately obvious; see corollary 2.3 below for a proof. Hence $\mathscr{R}$ is a natural extension of $\mathscr{R}_0$ to a class of partial functions.

The term *recursive function* is used nowadays to describe μ-recursive functions; so a recursive function is always total – it is a totally defined partial recursive function. The term *general recursive* function is sometimes used to describe μ-recursive functions, although historically, this was the name Kleene gave to the total functions given by his equation calculus approach ((*a*) in § 1). It was Kleene who proved the equivalence of general recursive functions (given by the equation calculus) and μ-recursive functions.

We now outline a proof of

2.2. *Theorem*
$\mathscr{R} = \mathscr{C}$.

Proof. From the main results of chapter 2 (lemma 1.1, theorems 3.1, 4.4, 5.2) it follows that $\mathscr{R} \subseteq \mathscr{C}$.

For the converse, suppose that $f(\boldsymbol{x})$ is a URM-computable function, computed by a program $P = I_1, \ldots, I_s$. By a *step* in a computation $P(\boldsymbol{x})$ we mean the implementation of one instruction. Consider the following functions connected with computations under P.

$$c(\boldsymbol{x}, t) = \begin{cases} \text{contents of } R_1 \text{ after } t \text{ steps in the computation} \\ P(\boldsymbol{x}), \text{ if } P(\boldsymbol{x}) \text{ has not already stopped;} \\ \\ \text{the final contents of } R_1 \text{ if } P(\boldsymbol{x}) \text{ has stopped} \\ \text{after fewer than } t \text{ steps.} \end{cases}$$

$$j(\boldsymbol{x}, t) = \begin{cases} \text{number of the next instruction, when } t \text{ steps of} \\ \text{the computation } P(\boldsymbol{x}) \text{ have been performed,} \\ \quad \text{if } P(\boldsymbol{x}) \text{ has not stopped after } t \text{ steps or fewer;} \\ \\ 0 \quad \text{if } P(\boldsymbol{x}) \text{ has stopped after } t \text{ steps or fewer.} \end{cases}$$

Clearly c and j are total functions.

If $f(\boldsymbol{x})$ is defined, then $P(\boldsymbol{x})$ converges after exactly t_0 steps, where

$$t_0 = \mu t(j(\boldsymbol{x}, t) = 0),$$

and then

$$f(\boldsymbol{x}) = c(\boldsymbol{x}, t_0).$$

If, on the other hand, $f(\boldsymbol{x})$ is not defined, then $P(\boldsymbol{x})$ diverges, and so $j(\boldsymbol{x}, t)$ is never zero. Thus $\mu t(j(\boldsymbol{x}, t) = 0)$ is undefined. Hence, in either case, we

have

$$f(\boldsymbol{x}) \simeq c(\boldsymbol{x}, \mu t(j(\boldsymbol{x}, t) = 0)).$$

So, to show that f is partial recursive, it is sufficient to show that c and j are recursive functions. It is clear that these functions are computable in the informal sense – we can simply simulate the computation $P(\boldsymbol{x})$ for up to t steps. By a detailed analysis of computations $P(\boldsymbol{x})$ and utilising many of the functions obtained in chapter 2, it is not difficult, though rather tedious, to show that c and j are recursive; in fact, they can be obtained from the basic functions without the use of minimalisation (so they are *primitive recursive* – see § 3 of this chapter). (A detailed proof of rather more than this will be given in chapter 5 – theorem 1.2 and Appendix). Hence f is partial recursive. □

2.3. *Corollary*
Every total function in $\mathscr{R}$ belongs to $\mathscr{R}_0$.

Proof. Suppose that $f(\boldsymbol{x})$ is a total function in $\mathscr{R}$; then f is URM-computable by a program P. Let c and j be the functions defined in the proof of theorem 2.2; as noted there, these can be obtained without any use of minimalisation, so in particular they are in $\mathscr{R}_0$. Further, since f is total, $P(\boldsymbol{x})$ converges for every $\boldsymbol{x}$, so the function $\mu t(j(\boldsymbol{x}, t) = 0)$ is total and belongs to $\mathscr{R}_0$. Now

$$f(\boldsymbol{x}) = c(\boldsymbol{x}, \mu t(j(\boldsymbol{x}, t) = 0)),$$

so f also is in $\mathscr{R}_0$. □

A predicate $M(\boldsymbol{x})$ whose characteristic function c_M is recursive is called a *recursive predicate*. In view of theorem 2.2, a recursive predicate is the same as a decidable predicate.

3. A digression: the primitive recursive functions

This is a natural point to mention an important subclass of $\mathscr{R}$, the class of *primitive recursive functions*, although they do not form part of the main line of thought in this chapter. These functions were referred to in chapter 2 § 5.

3.1. *Definition*
(*a*) The class $\mathscr{PR}$ of *primitive recursive functions* is the smallest class of functions that contains the basic functions $\mathbf{0}$, $x+1$, U_i^n, and is closed under the operations of substitution and recursion.

(*b*) A *primitive recursive predicate* is one whose characteristic function is primitive recursive.

All of the particular computable functions obtained in §§ 1, 3, 4 of chapter 2 are primitive recursive, since minimalisation was not used there. We have already noted that the functions c and j used in the proof of theorem 2.2 are primitive recursive. Further, from theorems 2-4.10 and 2-4.12 we see that $\mathscr{PR}$ is closed under bounded sums and products, and under bounded minimalisation. Thus the class of primitive recursive functions is quite extensive.

There are nevertheless recursive functions (or, equivalently, total computable functions) that are not primitive recursive. Indeed, the Ackermann function ψ of example 2-5.5 was given as an instance of such a function. A detailed proof that the Ackermann function is not primitive recursive is rather lengthy, and we refer the reader to Péter [1967, chapter 9] or Mendelson [1964, p. 250, exercise 11]. Essentially one shows that ψ grows faster than any given primitive recursive function. (To see how fast ψ grows try to calculate a few simple values.)

In chapter 5 we will be able to give an example of a total computable (i.e. recursive) function that we shall prove is not primitive recursive.

Our conclusion is that although the primitive recursive functions form a natural and very extensive class, they do *not* include all computable functions and thus fall short as a possible characterisation of the informal notion of computability.

4. Turing-computability

The definition of computability proposed by A. M. Turing [1936] is based on an analysis of a human agent's implementation of an algorithm, using pen and paper. Turing viewed this as a succession of very simple acts of the following kinds

(*a*) writing or erasing a single symbol.

(*b*) transferring attention from one part of the paper to another.

At each stage the algorithm specifies the action to be performed next. This depends only on (i) the symbol on the part of the paper currently being scrutinised by the agent, and (ii) the current state (of mind) of the agent. For the purposes of implementing the algorithm this is assumed to be determined entirely by the algorithm and the history of the operation so far. It may incorporate a partial record of what has happened to date, but it will not reflect the mood or intelligence of the agent, or the state of his indigestion. Moreover, there are only finitely many distinguishable

states in which the agent can be, because he is finite. The state of the agent may, of course, change as a result of the action taken at this stage.

Turing devised finite machines that carry out algorithms conceived in this way. There is a different machine for each algorithm. We shall briefly describe these machines, which have become known as *Turing machines*.

4.1. *Turing machines.*

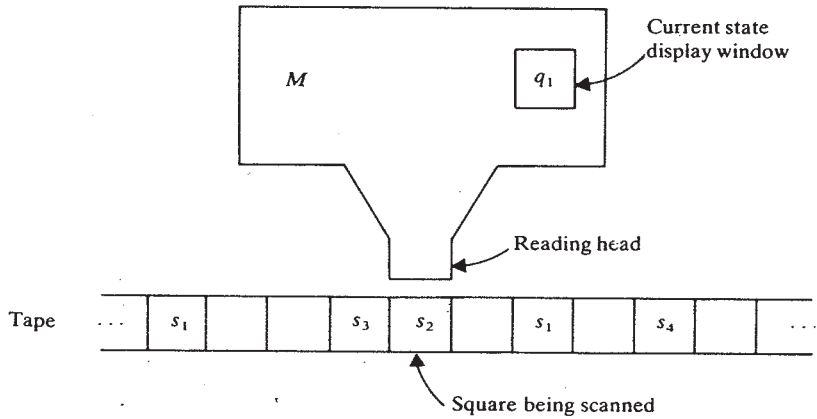
A Turing machine M is a finite device, which performs operations on a paper tape. This tape is infinite in both directions, and is divided into single squares along its length. (The tape represents the paper used by a human agent implementing an algorithm; each square represents a portion of the paper capable of being viewed in a given instant. In any particular terminating computation under M only a finite part of the tape will be used, although we may not know in advance how much will be needed. The tape is nevertheless infinite, corresponding to the human situation where we envisage an unlimited supply of clean paper.)

At any given time each square of the tape is either blank or contains a single symbol from a fixed finite list of symbols $s_1, s_2, \ldots, s_n$, the *alphabet* of M. We will let B denote a blank, and count it as the symbol s_0 belonging to M's alphabet.

M has a *reading head* which at any given time scans or reads a single square of the tape. We can visualise this as shown in fig. 3*a*.

M is capable of three kinds of simple operation on the tape, namely:

Fig. 3*a*. A Turing machine.



(a) erase the symbol in the square being scanned and replace it by another symbol from the alphabet of M;

(b) move the reading head one square to the right of that being scanned (or, equivalently, move the tape one square to the left);

(c) move the reading head one square to the left of that being scanned (or, equivalently, move the tape one square to the right).

At any given time M is in one of a fixed finite number of states, represented by symbols $q_1, \ldots, q_m$. During operation the state of M can change. We may envisage the symbol for the current state as being displayed in a window on the exterior of M (as in fig. 3a), and think of this as a partial guide to what has happened to date and what will happen in the future.

The action that M takes at any instant depends on the current state of M and on the symbol currently being scanned. This dependence is described in M's *specification* which consists of a finite set Q of quadruples, each of which takes one of the following forms

$$q_i s_j s_k q_l$$

$$q_i s_j R q_l \qquad \begin{pmatrix} 1 \leq i, l \leq m \\ 0 \leq j, k \leq n \end{pmatrix}$$

$$q_i s_j L q_l$$

A quadruple $q_i s_j \alpha q_l$ in Q specifies the action to be taken by M when it is in the state q_i and scanning the symbol s_j, as follows:

1. Operate on the tape thus:
   (a) if $\alpha = s_k$, erase s_j and write s_k in the square being scanned;
   (b) if $\alpha = R$, move the reading head one square to the right;
   (c) if $\alpha = L$, move the reading head one square to the left;
2. Change into state q_l.

The specification Q is such that for every pair $q_i s_j$ there is at most one quadruple of the form $q_i s_j \alpha \beta$; otherwise there could be ambiguity about what M does next.

To begin a computation, M must be provided with a tape and positioned so that a specified square is being scanned; further, M must be set in some prescribed initial state. Then, if M is in the state q_i and scans the symbol s_j, it acts as described above provided that there is a quadruple of the form $q_i s_j \alpha q_l$ in Q. This kind of action is then repeated for the new state and symbol scanned, and so on. M continues in this way for as long as possible. The operation of M terminates only when it is in a state q_i scanning a symbol s_j such that there is no quadruple of the form $q_i s_j \alpha \beta$ in Q; i.e. there is no quadruple in Q that specifies what to do next. (It is possible that this never happens.)

4.2. *Example*

Let M be a Turing machine whose alphabet consists of the symbols 0, 1, (and a blank of course) and whose possible states are q_1 and q_2. The specification of M is

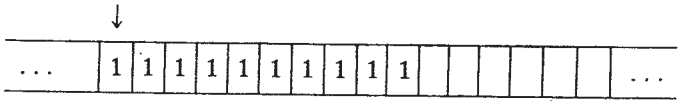
$q_1 0 R q_1$

$q_1 1 0 q_2$

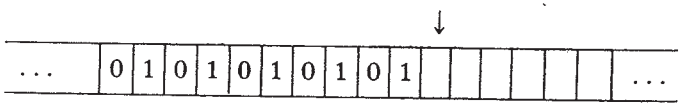
$q_2 0 R q_2$

$q_2 1 R q_1$

Suppose that M is provided with the tape



scanning the square marked ↓, and initially in state q_1. It is easy to see that M's action is to work from left to right along the tape, replacing alternate 1s by the symbol 0; M stops when it scans the first blank square since there is no quadruple that specifies what it should do. The resulting tape is



with the square marked ↓ being scanned, and M is in state q_1.

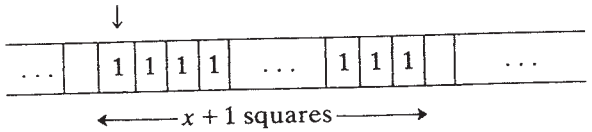
On the other hand, if M is provided with a tape such that every square contains the symbol 0 or 1, then the operation of M never stops.

It is clear from this example that a Turing machine M is a device for effecting an algorithm that operates on tapes. Complete details of the algorithm are contained in the specification Q of M. Thus, for the mathematician, a Turing machine is *defined* to be the set of quadruples that specify it. It is not usual to build physical Turing machines, except for illustrative purposes.

4.3. *Turing-computable functions*

In order to regard a Turing machine M as computing a numerical function, we must use some convention for representing numbers on a tape. One way is as follows: suppose, for convenience of exposition, that the symbol s_1 of M's alphabet is 1. We use 1 as a 'tally symbol', and then

represent a number x on the tape as follows (ignore the marker ↓ for the moment):

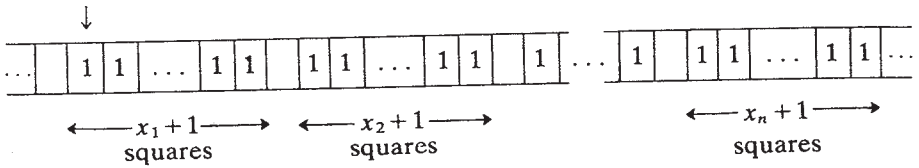


We use $x+1$ tallys to represent x, so as to distinguish 0 from a blank tape.

The *partial function* $f(x)$ *computed by* M is defined as follows. Consider the computation by M on the above tape, starting in state q_1, and initially scanning the square marked ↓. Then

$$f(x) = \begin{cases} \text{the total number of occurrences of the symbol 1} \\ \text{on the final tape,} \quad \text{if this computation eventually stops;} \\ \text{undefined} \quad \text{otherwise.} \end{cases}$$

Similarly, the n-ary *partial function* $f(x_1, \ldots, x_n)$ *computed by* M is defined by counting the number of 1s on the final tape when M is started in state q_1 and scanning the square marked ↓ on the following tape:



4.4. *Definition*

A partial function is *Turing-computable* if there is a Turing machine that computes it. The class of all Turing-computable functions is denoted $\mathscr{TC}$.

4.5. *Example*

The function $x+y$ is Turing-computable; the Turing machine given by the following specification Turing-computes this function.

$q_1 1 B q_1$

$q_1 B R q_2$

$q_2 1 B q_3$

$q_2 B R q_2$

The tape representation of (x, y) contains $x+y+2$ occurrences of the symbol 1, so the machine M is designed to erase the first two of these occurrences starting from the left. The details are easy to check by trying a few particular values for x, y.

4.6. *Exercises*

1. What unary function is Turing-computed by the machine in example 4.2?
2. Devise Turing machines that will Turing-compute the functions (*a*) $x \dot{-} 1$ (*b*) $2x$.

It is not our purpose here to develop the theory of Turing machines and Turing-computable functions; the interested reader should consult the books by Davis [1958] or Minsky [1967] listed in the bibliography.

The fundamental result linking Turing-computability with partial recursive functions and URM-computability is the following.

4.7. *Theorem*

$\mathscr{R} = \mathscr{TC} = \mathscr{C}$.

Proof. There are various ways of establishing this result, which we indicate in barest outline.

A direct proof that $\mathscr{TC} \subseteq \mathscr{R}$ is somewhat similar to the proof that $\mathscr{C} \subseteq \mathscr{R}$ (theorem 2.2). The tape configurations and states of a Turing machine during a computation can be coded by natural numbers, and the operation of the machine is then represented by recursive functions of these numbers.

For the converse inclusion, $\mathscr{R} \subseteq \mathscr{TC}$, one can show directly that $\mathscr{TC}$ contains the basic functions and is closed under substitution, recursion, and minimalisation. This is done in detail in Davis [1958]. Alternatively, one can show that $\mathscr{C} \subseteq \mathscr{TC}$ by showing that URMs are equivalent in power to a succession of simpler machines, ending with Turing machines. This is the proof given in their original paper by Shepherdson & Sturgis [1963]. □

5. Symbol manipulation systems of Post and Markov

E. L. Post and A. A. Markov formulated their ideas of effectiveness in terms of strings of symbols. They recognised that objects (including numbers) to which effective processes apply can always be represented as strings of symbols; in fact, in contexts such as symbolic logic, abstract algebra and the analysis of languages the objects actually

are strings of symbols. Both Post and Markov, from different points of view, considered that effective operations on strings of symbols are those that are built up from very simple manipulations of the strings themselves.

Post's central idea was that of a *canonical system*, which we describe below. Such systems do not compute functions; they generate sets of strings. This is because Post aimed to characterise *formal logical systems*; i.e. systems that generate *theorems* from *axioms* by the mechanical application of rules of logic. Thus a notion of effectiveness emerges from Post's work, initially in the guise of *effectively* (or *mechanically*) *generated sets*. We shall see how a notion of a Post-computable function can be derived from this.

In paragraph 5.17 below we explain the way in which Markov's approach is related to that of Post.

We must now define some notation to aid our discussion. Let $\Sigma = \{a_1, \ldots, a_k\}$ be a finite set of symbols, called an *alphabet*. A *string* from Σ is any sequence $a_{i_1} \ldots a_{i_n}$ of symbols from Σ. Strings are sometimes called *words*, by analogy with ordinary language. For any alphabet Σ, we write Σ^* to denote the set of all strings from Σ. Included in Σ^* is the *empty string*, denoted Λ, that has no symbols. If $\sigma = b_1 b_2 \ldots b_m$ and $\tau = c_1 \ldots c_n$ are strings then $\sigma\tau$ denotes the string $b_1 \ldots b_m c_1 \ldots c_n$. The empty string Λ has the property that for any string σ, $\sigma\Lambda = \sigma = \Lambda\sigma$.

5.1. *Post-systems*

In elementary algebra a common operation is to replace the string $(x-y)(x+y)$ whenever it occurs by the string (x^2-y^2). This string manipulation may be denoted by writing

$$S_1(x-y)(x+y)S_2 \to S_1(x^2-y^2)S_2$$

where S_1 and S_2 are arbitrary strings.

A more general manipulation of a string, yet still regarded by Post as elementary, takes the form

(5.2) $$g_0 S_1 g_1 S_2 \ldots g_{m-1} S_m g_m \to h_0 S_{i_1} h_1 S_{i_2} h_2 \ldots S_{i_n} h_n$$

where

(i) $g_0, \ldots, g_m, h_0, \ldots, h_n$ are fixed strings (and may be null),
(ii) the subscripts $i_1, \ldots, i_n$ are all from $1, 2, \ldots, m$, and need not be distinct.

Post called an operation of the form (5.2) a *production*; it may be applied to any string σ that can be analysed as

$$\sigma = g_0\sigma_1 g_1 \sigma_2 \ldots \sigma_m g_m \quad (\sigma_1, \ldots, \sigma_m \text{ are strings}),$$

to produce the string $h_0\sigma_{i_1}h_1\sigma_{i_2}h_2 \ldots \sigma_{i_n}h_n$.

5.3. *Example*

Let $\Sigma = \{a, b\}$; consider the production

(π) $aS_1bS_2 \to S_2aS_2a$.

Then the effect of (π) on some strings is given in the table

σ	Strings produced by (π)
aba	*aaaa*
abba	*aaaa* and *baabaa*
ba	(π) does not apply.

(The entries in the second line correspond to the two possible analyses of *abba*: as $\underset{S_1}{a}\underset{S_2}{b\,b\,a}$ and $\underset{S_1}{a\,\Lambda\,b}\underset{S_2}{b\,a}$.)

5.4. *Exercise*

Examine the ways in which the production

$S_1bS_2aaS_3b \to S_3abS_1$

applies to the string *babaabbaab*.

Productions form the main ingredient of Post's *canonical systems*:

5.5. *Definition*

A *Post-(canonical) system* $\mathcal{G}$ consists of
(a) a finite alphabet Σ,
(b) a finite subset A of Σ^*, the *axioms* of $\mathcal{G}$,
(c) a finite set Q of productions of the form (5.2), whose fixed strings are in Σ^*.

We say that $\mathcal{G}$ is a *Post-system over* Σ.

We write $\sigma \overset{Q}{\Rightarrow} \tau$ if the string τ can be obtained from the string σ by a finite succession of applications of productions in Q; then we write $\vdash_{\mathcal{G}} \tau$ if there is an axiom $\sigma \in A$ such that $\sigma \overset{Q}{\Rightarrow} \tau$. In this case we say that τ is *generated* by the Post-system; the set of all strings in Σ^* generated by $\mathcal{G}$ is denoted $T_{\mathcal{G}}$, i.e.

$$T_{\mathcal{G}} = \{\sigma \in \Sigma^* : \vdash_{\mathcal{G}} \sigma\}.$$

The set $T_{\mathcal{G}}$ is also called the set of *theorems* of $\mathcal{G}$, reflecting the original motivation of Post.

5.6. *Example*

Let $\mathcal{G}$ be the Post-system with alphabet $\Sigma = \{a, b\}$, axioms Λ, a, b and productions $S \to aSa$ and $S \to bSb$. Then $T_{\mathcal{G}}$ is the set of *palindromes* – the strings reading the same in either direction, such as *aba*, *bbabb*, *abba*.

Sometimes, in order to generate a particular set of strings in Σ^* it is necessary to use auxiliary symbols in the generation process. This leads to the following definition:

5.7. *Definition*

Let Σ be an alphabet and let $X \subseteq \Sigma^*$. Then X is *Post-generable* if there is an alphabet $\Sigma_1 \supseteq \Sigma$ and a Post-system $\mathcal{G}$ over Σ_1 such that X is the set of strings in Σ^* that are generated by $\mathcal{G}$, i.e. $X = T_{\mathcal{G}} \cap \Sigma^*$.

Post proved a remarkable theorem showing that really only very simple productions are needed to generate Post-generable sets. A set of productions Q (and any system in which they occur) is said to be *normal* if all the productions have the form $gS \to Sh$.

Post proved

5.8. *Theorem* (Post's normal form theorem)

Any Post-generable set can be generated by a normal system.

For an excellent proof consult Minsky [1967].

Post-systems having only productions of the kind

$$S_1 g S_2 \to S_1 h S_2$$

give models of grammars and languages. They reflect the way in which complex sentences of a language are built up from certain basic units according to the rules of grammar. Restrictions on the nature of g and h provide the *context-sensitive* and *context-free* languages of Chomsky, which provide useful models of languages used in computer programming. We cannot pursue this interesting topic here: the reader may consult the books of Arbib [1969] and Manna [1974] for further information.

5.9. *Post-systems and other approaches to computability*

As we have seen, Post-systems give a characterisation of the notion of an *effectively generated set*. We may compare this with the corresponding notion that emerges from the other approaches to computability. For URM-computability (or Turing-computability, etc.) effectively generated sets of numbers are called *recursively enumerable* (r.e.); these are the sets that are the range of some URM-computable function. (We shall study r.e. sets in chapter 7.)

To compare sets of *strings* with sets of numbers, we choose an (intuitively) effective coding function $\hat{}: \Sigma^* \to \mathbb{N}$ under which the string $\sigma \in \Sigma^*$ is coded by the number $\hat{\sigma}$. A convenient method for an alphabet $\Sigma = \{a_1, \ldots, a_k\}$ is by the *k-adic coding* where $\hat{}: \Sigma^* \to \mathbb{N}$ is defined by $\hat{\Lambda} = 0$; $\widehat{a_{r_0} \ldots a_{r_m}} = r_0 + r_1 k + \ldots + r_m k^m$. It is easily seen that $\hat{}$ is actually a bijection, so if the inverse of $\hat{}$ is denoted by $\tilde{}: \mathbb{N} \to \Sigma^*$ we also have a *representation* of each number n by a string $\tilde{n}$.

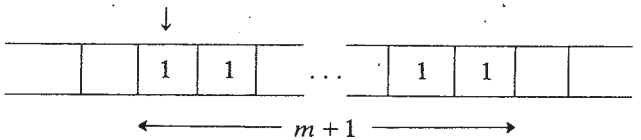
Suppose now that X is a set of strings: let $\hat{X} = \{\hat{\sigma}: \sigma \in X\}$, the set of numbers coding X. We have the equivalence result:

5.10. *Theorem*

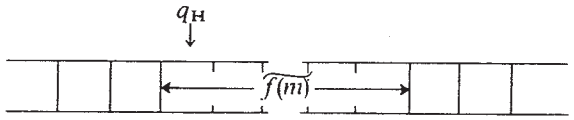
X is Post-generable iff $\hat{X}$ is r.e.

Proof. We sketch one proof of this result. Let $X \subseteq \Sigma^*$ and suppose first that $\hat{X}$ is r.e. Let $\hat{X} = \mathrm{Ran}(f)$, where f is URM-computable. Using earlier equivalences we can design a Turing machine M whose symbols include the alphabet Σ, so that when in state q_1 and given initial tape

(*)



M halts if and only if $m \in \mathrm{Dom}(f)$, and does so in the following configuration

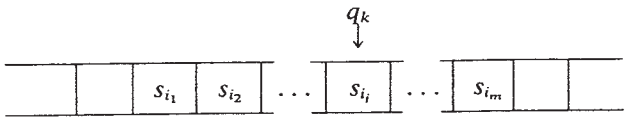


i.e. the symbols on the non-blank part of the tape indicated constitute the string $\widetilde{f(m)}$ from Σ, and M is in a special halting state q_H scanning the square marked $\downarrow$. Now devise a Post-system $\mathcal{G}$ to simulate the behaviour of M on its tape; the alphabet of $\mathcal{G}$ will include Σ and the symbols and

states of M; $\mathcal{G}$ will generate strings of the form

$$s_{i_1}s_{i_2}\ldots q_k s_{i_j}\ldots s_{i_m}$$

to represent any situation



that occurs during the computation by M from an initial tape of the form (*). To get things going, $\mathcal{G}$ will generate all strings of the form $q_1\overbrace{111\ldots11}^{m+1}$, which represent such initial tapes. If we include in $\mathcal{G}$ the production $q_H S \to S$, then the strings from Σ^* generated by $\mathcal{G}$ will be the set

$$\{\widetilde{f(m)}: m \in \mathrm{Dom}(f)\} = \{\tilde{n}: n \in \mathrm{Ran}(f)\} = \{\tilde{n}: n \in \hat{X}\} = X.$$

Thus $X = T_{\mathcal{G}} \cap \Sigma^*$, so X is Post generable.

Conversely, if X is Post-generable by a Post-system $\mathcal{G}$, show that the relation

'$\tilde{n}$ is generated by $\mathcal{G}$ using at most m productions'

is decidable; from the theory of r.e. sets (chapter 7) it follows easily that $\hat{X}$ is r.e. □

5.11. *Post-computability*

We now explain two ways to derive a concept of computable function from Post-systems. In both cases the concept is defined first for functions on Σ^*, and then extended to $\mathbb{N}$ by coding or representation.

Suppose that $f: \Sigma^* \to \Sigma^*$ is a partial function. Select a symbol $\cdot$ not in Σ and consider the set of strings

$$G(f) = \{\sigma \cdot f(\sigma): \sigma \in \mathrm{Dom}(f)\}$$

from the alphabet $\Sigma \cup \{\cdot\}$. The set $G(f)$ contains all information about f, and we define:

5.12. *Definition*

$f: \Sigma^* \to \Sigma^*$ is *Post-computable* if $G(f)$ is Post-generable.

5.13. *Example*

Let $\Sigma = \{1\}$ and consider the function $f: \Sigma^* \to \Sigma^*$ given by

$f(\Lambda) = \Lambda$,

$f(11\ldots11) = 1111\ldots1111$.

$\longleftarrow n \longrightarrow \quad \longleftarrow n^2 \longrightarrow$

The set $G(f)$ is generated by the following Post-system:

Alphabet $\{1, \cdot\}$

Axiom $\cdot\,(= \Lambda \cdot f(\Lambda))$

Production $S_1 \cdot S_2 \to S_1 1 \cdot S_2 S_1 S_1 1$.

The single production of this system applies to a string of the form

$11\ldots11 \cdot 111\ldots111$

$\longleftarrow n \longrightarrow \quad \longleftarrow n^2 \longrightarrow$

to produce the string

$11\ldots111 \cdot 111\ldots11111\ldots1111\ldots111.$

$\longleftarrow n+1 \longrightarrow \quad \longleftarrow n^2 \longrightarrow \longleftarrow n \longrightarrow \longleftarrow n \longrightarrow$

Hence f is Post-computable.

Suppose that $G(f)$ is generated by a Post-system $\mathcal{G}$. To see that f is computable in the informal sense, consider the following algorithm for finding $f(\sigma)$ (where $\sigma \in \Sigma^*$).

'Generate the strings $T_{\mathcal{G}}$ in some systematic fashion; examine these as they appear, looking for a string of the form $\sigma \cdot \tau$ with $\tau \in \Sigma^*$. Such a string will eventually be produced if, and only if, $\sigma \in \mathrm{Dom}(f)$, and then $\tau = f(\sigma)$.'

The definition (5.12) extends in an obvious way to partial functions $(\Sigma^*)^n \to \Sigma^*$. Post-computability on $\mathbb{N}$ is then defined using any effective representation $\tilde{}: \mathbb{N} \to \Sigma^*$ in the natural way:

5.14. *Definition*

Let $g: \mathbb{N}^n \to \mathbb{N}$ be a partial function, and let

$\tilde{g}: (\Sigma^*)^n \to \Sigma^*$ be the function defined by

$\tilde{g}(\tilde{m}_1, \tilde{m}_2, \ldots, \tilde{m}_n) = \widetilde{g(m_1, \ldots, m_n)} \quad (m_1, \ldots, m_n \in \mathbb{N})$.

Then g is *Post-computable* if $\tilde{g}$ is Post-computable.

If we let $\mathcal{PC}$ denote the class of Post-computable functions of natural numbers, we have the equivalence:

5.15. *Theorem*

$\mathcal{PC} = \mathcal{C} = \mathcal{TC} = \mathcal{R}$. (We omit a proof. When the reader has studied chapters 6 and 7 he should be able to see that this follows from theorem 5.10 and the results of chapters 6 and 7 linking r.e. sets with URM-computable functions.)

An alternative way to derive a notion of computability from Post-systems is to simulate computations by a machine directly. This can be done by using sets of productions Q such that *at most* one production in Q can apply to any given string. Such sets, and the systems in which they occur, are called *monogenic*. A monogenic system operates sequentially like a machine. It is convenient for the following to write $\sigma \overset{Q}{\Rightarrow}| \tau$ to mean that $\sigma \overset{Q}{\Rightarrow} \tau$ but no production in Q applies to τ.

We have the following characterisation of Post-computability.

5.16 *Theorem*

Let $f: \Sigma^* \to \Sigma^*$ be a partial function, and let X, Y be symbols not in Σ. Then the following are equivalent.

(*a*) *f is Post-computable;*

(*b*) *there is a monogenic set of productions Q over an alphabet $\Sigma_1 \supseteq \Sigma \cup \{X, Y\}$ such that for $\sigma \in \Sigma^*$ and $\tau \in \Sigma_1^*$*

$$X\sigma \overset{Q}{\Rightarrow}| \tau \text{ iff } \sigma \in \mathrm{Dom}(f) \text{ and } \tau \text{ is the string } Yf(\sigma).$$

(*Note.* The symbols X and Y are needed to distinguish 'input' strings from 'output' strings. Otherwise a desired output string $f(\sigma)$ would be regarded as a new input string to which further productions might apply.)

Proof. The implication $(a) \Rightarrow (b)$ may be obtained by first showing that f is computable by a Turing machine M, and then devising a monogenic set of productions to simulate M. In fact it is possible to obtain a *normal* monogenic set of productions for this task.

For $(b) \Rightarrow (a)$ it is quite straightforward to show that the function $\hat{f}: \mathbb{N} \to \mathbb{N}$ coding f is partial recursive, then apply theorem 5.15. □

This equivalence extends to functions $f: (\Sigma^*)^n \to \Sigma^*$ by considering 'inputs' of the form $X\sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n$.

For a fuller discussion of Post systems the reader is referred to the excellent chapters in Minsky [1967].

5.17. *Markov-computability*

Markov's notion of computability is very similar to that derived from Post-systems by the second method above. Rather than restrict attention to monogenic systems, however, Markov gave rules to determine uniquely which of the available productions to apply next. Details are, briefly, as follows:

A *Markov normal algorithm* over an alphabet Σ is essentially a list Q of productions over Σ with the following features:

(i) every production in Q has the form $S_1 g S_2 \to S_1 h S_2$,

(ii) certain productions in Q are singled out as being *terminal.*

Given an input string σ, a Markov algorithm applies productions in Q sequentially, according to the following rules.

(*a*) if more than one production in Q applies to a string σ, use the first one in the list Q,

(*b*) if a production $S_1 g S_2 \to S_1 h S_2$ applies to σ in more than one way, apply it to the leftmost occurrence of g in σ,

(*c*) the process halts having produced a string τ *either* when a terminal production is used *or* if no production in Q applies to τ.

With these rules, the definition of *Markov-computable function* $f: \Sigma^* \to \Sigma^*$ is given in the obvious way. A Markov normal algorithm to compute f may use an alphabet extending Σ. It is quite straightforward to establish:

5.18. *Theorem*

Let $f: \Sigma^* \to \Sigma^*$. Then f is Markov-computable iff f is Post-computable.

Markov-computability on $\mathbb{N}$ is defined by using some system of representing numbers in the usual way, and thus coincides with the other approaches to computability.

6. Computability on domains other than $\mathbb{N}$

In chapter 1 § 5 we showed how any notion of computability on $\mathbb{N}$ can be extended to other domains by the device of coding. By contrast, in the previous section the definition of Post-computability on the domain of strings on a finite alphabet was given directly in terms of the objects (strings) and their intrinsic structure. A variety of such direct approaches to computability on other domains is possible: we give two examples.

6.1. *Example*

$D = \mathbb{Z}$. The URM idea may be extended to handle integers by making the following modifications:

(*a*) each register contains an integer,

(*b*) there is an additional instruction $S^-(n)$ for each $n = 1, 2, 3, \ldots$ that has the effect of *subtracting* 1 from the contents of register R_n.

6.2. *Example*

$D = \Sigma^*$, where $\Sigma = \{a, b\}$. The class $\mathscr{R}^D$ of *partial recursive functions* on Σ^* is the smallest class of partial functions such that

(a) the basic functions

(i) $f(\sigma) = \Lambda$

(ii) $f(\sigma) = \sigma a$

(iii) $f(\sigma) = \sigma b$

(iv) the projection functions $U_i^n(\sigma_1, \ldots, \sigma_n) = \sigma_i$,

are in $\mathscr{R}^D$,

(b) $\mathscr{R}^D$ is closed under substitution,

(c) $\mathscr{R}^D$ is closed under primitive recursive definitions of the following form:

$$\begin{cases} h(\boldsymbol{\sigma}, \Lambda) \simeq f(\boldsymbol{\sigma}) \\ h(\boldsymbol{\sigma}, \tau a) \simeq g_1(\boldsymbol{\sigma}, \tau, h(\boldsymbol{\sigma}, \tau)) \\ h(\boldsymbol{\sigma}, \tau b) \simeq g_2(\boldsymbol{\sigma}, \tau, h(\boldsymbol{\sigma}, \tau)) \end{cases}$$

where $f, g_1, g_2 \in \mathscr{R}^D$,

(d) $\mathscr{R}^D$ is closed under *minimalisation*: if $f(\boldsymbol{\sigma}, \tau)$ is in $\mathscr{R}^D$, so is the function h given by

$$h(\boldsymbol{\sigma}) \simeq \mu\tau(f(\boldsymbol{\sigma}, \tau) = \Lambda)$$

where $\mu\tau$ means the first τ in the natural ordering $\Lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots$.

For each of these, and other approaches to computability on a domain D that utilise the intrinsic structure of D, we find as expected that they are equivalent to the approach that transfers the notion of computability from $\mathbb{N}$ by using coding. And, vice versa, any natural notion of computability on a domain D induces an alternative (but equivalent) notion of computability on $\mathbb{N}$ via coding, as with Post-computability in § 5.

6.3. *Exercises*

1. Prove that URM-computability on $\mathbb{Z}$ as outlined in example 6.1 is equivalent to URM-computability via coding (example 1-5.1).
2. Prove that the class $\mathscr{R}^D$ of partial recursive functions on Σ^*, as defined in example 6.2, is identical to the Post-computable functions on Σ^*.
3. Suggest natural definitions of computability on the domains (a) 3×3 matrices, (b) $\mathbb{Q}$ (rational numbers).
4. Give a natural definition of Turing-computability on Σ^*, where Σ is any finite alphabet.

7. Church's thesis

We now turn our attention to the second question of the introduction to this chapter: how well is the informal and intuitive idea of effectively computable function captured by the various formal characterisations?

In the light of their investigations, Church, Turing and Markov each put forward the claim that the class of functions he had defined coincides with the informally defined class of effectively computable functions. In view of the Fundamental result (1.1), these claims are all mathematically equivalent. The name *Church's thesis* (sometimes the *Church–Turing thesis*) is now used to describe any of these other claims. Thus, in terms of the URM approach, we can state:

Church's thesis

The intuitively and informally defined class of effectively computable partial functions coincides exactly with the class $\mathscr{C}$ of URM-computable functions.

Note immediately that this thesis is not a *theorem* which is susceptible to mathematical proof; it has the status of a *claim* or *belief* which must be substantiated by evidence. The evidence for Church's thesis, which we summarise below, is impressive.

1. The Fundamental result: many independent proposals for a precise formulation of the intuitive idea have led to the same class of functions, which we have called $\mathscr{C}$.
2. A vast collection of effectively computable functions has been shown explicitly to belong to $\mathscr{C}$; the particular functions of chapter 2 constitute the beginning of such a collection, which can be enlarged *ad infinitum* by the techniques of that chapter, and other more sophisticated methods.
3. The implementation of a program P on the URM to compute a function is clearly an example of an algorithm; thus, directly from the definition of the class $\mathscr{C}$, we see that all functions in $\mathscr{C}$ are computable in the informal sense. Similarly with all the other equivalent classes, the very definitions are such as to demonstrate that the functions involved are effectively computable.
4. No one has ever found a function that would be accepted as computable in the informal sense, that does not belong to $\mathscr{C}$.

On the basis of this evidence, and that of their own experience, most mathematicians are led to accept Church's thesis. For our part, we

propose to accept and use Church's thesis throughout the rest of this book, in a way that we now explain.

Suppose that we have an informally described algorithm for computing the values of a function f. Such an algorithm may be described in English, or by means of diagrams, or in semi-formal mathematical terms, or by any other means that communicate unambiguously how to effectively calculate the values of f, where defined, in a finite amount of time. In such a situation we may wish to prove that f is URM-computable. There are, broadly, two methods open to us.

Method 1. Write a program that URM-computes f (and prove that it does so), or prove by indirect means that such a program exists. This could be done, for instance, by the methods of chapter 2, or by showing that f belongs to one of the many classes shown by the Fundamental result to be equivalent to $\mathscr{C}$.

Such a full and formal proof that f is URM-computable may be a long and rather technical process. Essentially it would involve translation of the informally described algorithm into a program or into the language of one of the other formal characterisations. Probably there would be various flow diagrams as intermediate translations.

Method 2. Give an informal (though rigorous) proof that the given informal algorithm is indeed an algorithm that serves to compute f. Then appeal to Church's thesis and conclude immediately that f is URM-computable.

We propose to accept method 2 as a valid method of proof, which we call *proof by Church's thesis.*

7.1. *Examples*

1. Let P be a URM program; define a function f by

$$f(x, y, t) = \begin{cases} 1 & \text{if } P(x)\downarrow y \text{ after } t \text{ or fewer steps} \\ & \text{of the computation } P(x), \\ 0 & \text{otherwise.} \end{cases}$$

An informal algorithm for computing f is as follows.

'Given (x, y, t), simulate the computation $P(x)$ (on a piece of paper, for example, as in example 1-2.1), carrying out t steps of $P(x)$ unless this computation stops after fewer than t steps. If $P(x)$ stops after t or fewer steps, with y finally in R_1, then $f(x, y, t) = 1$. Otherwise (i.e. if $P(x)$ stops in t or fewer steps with some number other than y in R_1, or if $P(x)$ has not stopped after t steps) we have $f(x, y, t) = 0$.'

Simulation of $P(x)$ for at most t steps is clearly a mechanical procedure, which can be completed in a finite amount of time. Thus, f *is* effectively computable. Hence, by Church's thesis, f is URM-computable.

2. Suppose that f and g are unary effectively computable functions. Define a function h by

$$h(x) = \begin{cases} 1 & \text{if } x \in \mathrm{Dom}(f) \text{ or } x \in \mathrm{Dom}(g), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

An algorithm for h can be described in terms of given algorithms for the effectively computable functions f and g as follows:

'Given x, start the algorithms for computing $f(x)$ and $g(x)$ simultaneously. (Envisage two agents or machines working simultaneously, or one agent who does one step of each algorithm alternately.) If and when one of these computations terminates, then stop altogether, and set $h(x) = 1$. Otherwise, continue indefinitely.'

This algorithm gives $h(x) = 1$ for any x such that either $f(x)$ or $g(x)$ is defined; and it goes on for ever if neither is defined. Thus we have an algorithm for computing h, so by Church's thesis h is URM-computable.

3. Let $f(n) =$ the nth digit in the decimal expansion of $\pi = 3.14159\ldots$ (so we have $f(0) = 3$, $f(1) = 1$, $f(2) = 4$, etc.). We can obtain an informal algorithm for computing $f(n)$ as follows. Consider Hutton's series for π

$$\pi = \frac{12}{5}\left\{1 + \frac{2}{3}\left(\frac{1}{10}\right) + \frac{2\times 4}{3\times 5}\left(\frac{1}{10}\right)^2 + \ldots\right\} + \frac{14}{25}\left\{1 + \frac{2}{3}\left(\frac{1}{50}\right) + \frac{2\times 4}{3\times 5}\left(\frac{1}{50}\right)^2 + \ldots\right\} = \sum_{n=0}^{\infty} \frac{(n!2^n)^2}{(2n+1)!}\left\{\frac{12}{5}\left(\frac{1}{10}\right)^n + \frac{14}{25}\left(\frac{1}{50}\right)^n\right\} = \sum_{n=0}^{\infty} h_n, \text{ say.}$$

Let $s_k = \sum_{n=0}^{k} h_n$; by the elementary theory of infinite series $s_k < \pi < s_k + 1/10^k$.

Now s_k is rational, so the decimal expansion of s_k can be effectively calculated to any desired number of places using long division. Thus the following is an effective method for calculating $f(n)$ (given a number n):

'Find the first $N \geq n+1$ such that the decimal expansion

$s_N = a_0 \cdot a_1 a_2 \ldots a_n a_{n+1} \ldots a_N \ldots$

does *not* have all of $a_{n+1}, \ldots, a_N$ equal to 9. (Such an N exists, for otherwise the decimal expansion of π would end in recurring 9, making π rational.) Then put $f(n) = a_n$.'

To see that this gives the required value, suppose that $a_m \neq 9$ with $n < m \leq N$. Then by the above

$$s_N < \pi < s_N + 1/10^N \leq s_N + 1/10^m.$$

Hence $a_0 \cdot a_1 \ldots a_n \ldots a_m \ldots < \pi < a_0 \cdot a_1 \ldots a_n \ldots (a_m + 1) \ldots$ so the nth decimal place of π is indeed a_n.

Hence by Church's thesis, f is computable.

The student should try to provide complete formal proofs (method 1) that the functions in these examples are URM-computable (assuming, for example 2, that f and g are URM-computable). For all of them it is a lengthy and tedious task.

Note that in using Church's thesis we are not proposing to abandon all thought of proof, as if Church's thesis is a magic wand which we can wave instead. A proof by Church's thesis will always involve proof that is careful, and sometimes complicated, although informal. Moreover, anyone using Church's thesis in the way we propose should be able to provide a formal proof if challenged. (As if to anticipate such a challenge, we provide in the appendix to chapter 5 an alternative formal proof of one fundamental theorem in that chapter (theorem 5-1.2) on which almost all later development depends. This then serves to substantiate further Church's thesis; incidentally, it is a simple formal corollary that the functions in the first two examples above are URM-computable also.)

Church's thesis not only keeps proofs shorter, but also prevents the main idea of a proof or construction from being obscured by a mass of technical details. It remains, however, an expression of faith or confidence. The validity of faith depends on the evidence that can be mustered. In the case of Church's thesis, there is the mathematical evidence already outlined. For the practised student there is the additional evidence of his own experience in translating informal algorithms into formal counterparts. For the beginner, our use of Church's thesis in subsequent chapters may call on his willingness to place confidence in the ability of others until self confidence is developed.

To conclude: *for the remainder of this book, we accept Church's thesis and use it in the manner described above, often without explicit reference.*

7.2. *Exercises*

1. Suppose that $f(x)$ and $g(x)$ are effectively computable functions. Prove, using Church's thesis, that the function h given by

$$h(x) = \begin{cases} x & \text{if } x \in \mathrm{Dom}(f) \cap \mathrm{Dom}(g), \\ \text{undefined} & \text{otherwise} \end{cases}$$

is URM-computable.

2. Suppose that f is a total unary computable function. Prove, by Church's thesis, that the following function h is URM-computable

$$h(x) = \begin{cases} 1 & \text{if } x \in \mathrm{Ran}(f), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

3. Give a detailed proof by Church's thesis that the Ackermann function (example 2-5.5) is computable.
4. Prove by Church's thesis that the function g given by

$g(n) = n$th digit in the decimal expansion of e

is computable (where the number e is the basis for natural logarithms).

4
Numbering computable functions

We return now to the study of URM-computable functions. Henceforth the term *computable* standing alone means URM-computable, and *program* means URM program.

The key fact established in this chapter is that the set of all programs is *effectively denumerable*: in other words there is an effective coding of programs by the set of all natural numbers. Among other things, it follows that the class $\mathscr{C}$ is denumerable, which implies that there are many functions that are not computable. In § 3 we discuss Cantor's diagonal method, whereby this is established.

The numbering or coding of programs, and particularly its effectiveness, is absolutely fundamental to the development of the theory of computability. We cannot overemphasise its importance. From it we obtain codes or indices for computable functions, and this means that we are able to pursue the idea of effective operations involving such codes.

In § 4 we prove the first of two important theorems involving codes of functions: the so-called *s–m–n* theorem of Kleene. (The second theorem is the main result of chapter 5.)

1. Numbering programs

We first explain the terminology that we shall use.

1.1. *Definitions*

(a) A set X is *denumerable* if there is a bijection $f: X \to \mathbb{N}$.

(*Note.* The term *countable* is normally used to mean finite or denumerable; thus, for infinite sets, countable means the same as denumerable. The term *countably infinite* is used by some authors instead of denumerable.)

(b) An *enumeration* of a set X is a surjection $g: \mathbb{N} \to X$; this is often represented by writing

$$X = \{x_0, x_1, x_2, \ldots\}$$

where $x_n = g(n)$. This is an enumeration *without repetitions* if g is injective.

(c) Let X be a set of finite objects (for example a set of integers, or a set of instructions, or a set of programs); then X is *effectively denumerable* if there is a bijection $f: X \to \mathbb{N}$ such that both f and f^{-1} are effectively computable functions.

(*Note.* We mean here the *informal* notion of effectively computable. This is compelled on us since, in general, there is no available formal notion of computability of functions from X to $\mathbb{N}$.[1] In cases where some formal notion does apply, we take this to be the meaning, as for example in theorem 1.2(a).)

Clearly, a set is denumerable if and only if it can be enumerated without repetitions.

For the main result of this section we need the following (recall that $\mathbb{N}^+$ denotes the set of all positive natural numbers):

1.2. *Theorem*

The following sets are effectively denumerable.

(a) $\mathbb{N} \times \mathbb{N}$,

(b) $\mathbb{N}^+ \times \mathbb{N}^+ \times \mathbb{N}^+$,

(c) $\bigcup_{k>0} \mathbb{N}^k$, *the set of all finite sequences of natural numbers.*

Proof

(a) A bijection $\pi: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is defined by

$$\pi(m, n) = 2^m(2n+1) - 1.$$

It is clear from the definition that π is in fact a computable function; to see that the inverse is effectively computable observe that π^{-1} is given by

$$\pi^{-1}(x) = (\pi_1(x), \pi_2(x)),$$

where π_1, π_2 are the computable functions defined by $\pi_1(x) = (x+1)_1$, $\pi_2(x) = \frac{1}{2}((x+1)/2^{\pi_1(x)} - 1)$. (Cf. exercise 2-4.16(2).)

(b) An explicit bijection $\zeta: \mathbb{N}^+ \times \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}$ is given, using the function π of (a), by

$$\zeta(m, n, q) = \pi(\pi(m-1, n-1), q-1).$$

Then we have

$$\zeta^{-1}(x) = (\pi_1(\pi_1(x)) + 1, \pi_2(\pi_1(x)) + 1, \pi_2(x) + 1).$$

[1] To say that we should use a notion of computability based on some coding begs the whole question, since a coding is an effective (in the informal sense) function.

Since the functions π, π_1, π_2 are effectively computable, then so are ζ and ζ^{-1}.

(c) A bijection $\tau: \bigcup_{k>0} \mathbb{N}^k \to \mathbb{N}$ is defined by

$$\tau(a_1, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \ldots + 2^{a_1+a_2+\ldots a_k+k-1} - 1.$$

Clearly τ is an effectively computable function. To see that τ is a bijection, and to calculate $\tau^{-1}(x)$, we use the fact that each natural number has a unique expression as a binary decimal. Thus, given x, we can effectively find unique numbers $k \geq 1$ and $0 \leq b_1 < b_2 \ldots < b_k$ such that

$$x + 1 = 2^{b_1} + 2^{b_2} + \ldots + 2^{b_k}$$

from which we obtain

$$\tau^{-1}(x) = (a_1, \ldots, a_k),$$

where $a_1 = b_1$ and $a_{i+1} = b_{i+1} - b_i - 1$ $(1 \leq i < k)$. (Cf. exercise 2-4.16(5), where functions closely connected with the calculation of τ^{-1} are to be proved computable.) □

Let us now denote the set of all URM instructions by $\mathscr{I}$, and the set of all programs by $\mathscr{P}$. A program consists of a finite list of instructions, so we next consider the set $\mathscr{I}$.

1.3. *Theorem*

$\mathscr{I}$ is effectively denumerable.

Proof. We define an explicit bijection $\beta: \mathscr{I} \to \mathbb{N}$ that maps the four kinds of instruction onto natural numbers of the forms $4u$, $4u+1$, $4u+2$, $4u+3$ respectively; we use the functions π and ζ defined in the proof of theorem 1.2.

$$\beta(\mathrm{Z}(n)) = 4(n-1),$$
$$\beta(\mathrm{S}(n)) = 4(n-1)+1,$$
$$\beta(\mathrm{T}(m, n) = 4\pi(m-1, n-1)+2,$$
$$\beta(\mathrm{J}(m, n, q)) = 4\zeta(m, n, q)+3.$$

This explicit definition shows that β is effectively computable. To find $\beta^{-1}(x)$, first find u, r such that $x = 4u + r$ with $0 \leq r < 4$. The value of r indicates which kind of instruction $\beta^{-1}(x)$ is, and from u we can effectively find the particular instruction of that kind. Specifically:

if $r = 0$, then $\beta^{-1}(x) = \mathrm{Z}(u+1)$;

if $r = 1$, then $\beta^{-1}(x) = \mathrm{S}(u+1)$;

if $r = 2$, then $\beta^{-1}(x) = \mathrm{T}(\pi_1(u)+1, \pi_2(u)+1)$;

if $r = 3$, then $\beta^{-1}(x) = \mathrm{J}(m, n, q)$, where $(m, n, q) = \zeta^{-1}(u)$.

Hence β^{-1} is also effectively computable. □

Now we can prove:

1.4. *Theorem*

$\mathscr{P}$ is effectively denumerable.

Proof. Define an explicit bijection $\gamma: \mathscr{P} \to \mathbb{N}$ as follows, using the bijections τ and β of theorems 1.2 and 1.3: if $P = I_1, I_2, \ldots, I_s$ then

$$\gamma(P) = \tau(\beta(I_1), \ldots, \beta(I_s)).$$

Since τ and β are bijections, so is γ; the fact that τ, β and their inverses are effectively computable ensures that γ and γ^{-1} are also effectively computable. □

The bijection γ will play an important role in subsequent development. For a program P, the number $\gamma(P)$ is called the *code number* of P, or the *Gödel*[2] *number* of P, or just the *number* of P. We define

$$P_n = \text{the program with (code) number } n$$
$$= \gamma^{-1}(n),$$

and say that P_n is the nth program. By construction of γ, if $m \neq n$, then P_m differs from P_n, although these programs may compute the same functions.

It is of the utmost importance for later results that the functions γ and γ^{-1} are effectively computable; i.e.

(a) Given a particular program P, we can effectively find the code number $\gamma(P)$;

(b) given a number n, we can effectively find the program $P_n = \gamma^{-1}(n)$.

In order to emphasise this we give two simple illustrations.

1.5. *Examples*

(a) Let P be the program T(1, 3), S(4), Z(6). We will calculate $\gamma(P)$.

$$\beta(\mathrm{T}(1, 3)) = 4\pi(0, 2)+2 = 4(2^0(2\times 2+1)-1)+2 = 18,$$
$$\beta(\mathrm{S}(4)) = 4\times 3+1 = 13,$$
$$\beta(\mathrm{Z}(6)) = 4\times 5 = 20.$$

[2] The term *Gödel number* is used after K. Gödel who first exploited the idea of coding non-numerical objects by numbers in his famous paper (Gödel [1931]).

Hence $\gamma(P) = 2^{18} + 2^{32} + 2^{53} - 1$
$= 9007203549970431.$

(*b*) Let $n = 4127$; we will find P_{4127}.
$4127 = 2^5 + 2^{12} - 1$; thus P_{4127} is a program with two instructions I_1, I_2 where
$\beta(I_1) = 5 = 4 \times 1 + 1$,
$\beta(I_2) = 12 - 5 - 1 = 6 = 4 \times 1 + 2 = 4\pi(1, 0) + 2$.
Hence from the definition of β, $I_1 = \mathrm{S}(2)$ and $I_2 = \mathrm{T}(2, 1)$, so P_{4127} is
S(2)
T(2, 1)

There are, of course, many other possible effective bijections from $\mathscr{P}$ to $\mathbb{N}$; our choice in defining the details of γ was somewhat arbitrary. What is vital, we again emphasise, is that γ and γ^{-1} are effectively computable. The particular details of γ are not so important. For subsequent theory, any other bijection γ' would suffice, provided that γ' and its inverse are effectively computable. However, we have to fix on one particular numbering of programs, and we have chosen that given by γ. ***For the rest of this book, γ remains fixed, so that for each particular number n, the meaning of P_n does not change.*** Thus, for instance, P_{4127} always means the program S(2), T(2, 1).

1.6. *Exercise*
Find
(*a*) $\beta(\mathrm{J}(3, 4, 2))$,
(*b*) $\beta^{-1}(503)$,
(*c*) the code number of the following program:
T(3, 4), S(3), Z(1),
(*d*) P_{100}.

2. Numbering computable functions

Using our fixed numbering of programs, we can now number computable functions and their domains and ranges. We introduce some important notation which is basic to the rest of the book.

2.1. *Definition*
For each $a \in \mathbb{N}$, and $n \geq 1$:

(*a*) $\phi_a^{(n)}$ = the n-ary function computed by P_a
$= f_{P_a}^{(n)}$ in the notation of chapter 1 § 3,
(*b*) $W_a^{(n)}$ = domain of $\phi_a^{(n)} = \{(x_1, \ldots, x_n) : P_a(x_1, \ldots, x_n)\downarrow\}$,
$E_a^{(n)}$ = range of $\phi_a^{(n)}$.

We shall be mainly concerned with unary computable functions in later chapters, so for convenience we omit the superscript (1) when it occurs; thus we write ϕ_a for $\phi_a^{(1)}$, W_a for $W_a^{(1)}$, and E_a for $E_a^{(1)}$.

2.2. *Example*
Let $a = 4127$; from the previous section we know that P_{4127} is S(2), T(2, 1). Hence

$$\phi_{4127}(x) = 1 \qquad (\text{all } x)$$

and

$$\phi_{4127}^{(n)}(x_1, \ldots, x_n) = x_2 + 1 \qquad \text{if} \quad n > 1.$$

Thus

$$W_{4127} = \mathbb{N}, \qquad E_{4127} = \{1\};$$
$$W_{4127}^{(n)} = \mathbb{N}^n, \quad E_{4127}^{(n)} = \mathbb{N}^+ \quad \text{if } n > 1.$$

Suppose that f is a unary computable function. Then there is a program P, say, that computes f, so $f = \phi_a$, where $a = \gamma(P)$. We say then that a is an *index* for f. Since there are many different programs that compute a given function, we cannot say that a is *the* index for f; in fact, each computable function has infinitely many indices.

We conclude that every unary computable function appears in the enumeration

$$\phi_0, \phi_1, \phi_2, \ldots .$$

and that this is an enumeration *with* repetitions.

Similar remarks apply to n-ary functions and their enumeration.

2.3. *Exercise*
Prove that every computable function has infinitely many indices.

Recall that we denoted the set of all n-ary computable function by $\mathscr{C}_n$.

2.4. *Theorem*
$\mathscr{C}_n$ is denumerable.

Proof. We use the enumeration $\phi_0^{(n)}, \phi_1^{(n)}, \phi_2^{(n)}, \ldots$ (which has repetitions) to construct one without repetitions.

$$\text{Let } \begin{cases} f(0) = 0, \\ f(m+1) = \mu z(\phi_z^{(n)} \neq \phi_{f(0)}^{(n)}, \ldots, \phi_{f(m)}^{(n)}). \end{cases}$$

Then

$$\phi_{f(0)}^{(n)}, \phi_{f(1)}^{(n)}, \phi_{f(2)}^{(n)}, \ldots$$

is an enumeration of $\mathscr{C}_n$ without repetitions. □

Note. We are not claiming that f as defined in this proof is computable; in fact, we will be able to show later that this is not the case. It is possible, nevertheless, to give a complicated construction of a total computable function h such that $\phi_{h(0)}^{(n)}, \phi_{h(1)}^{(n)}, \ldots$ is an enumeration of $\mathscr{C}_n$ without repetitions. This was proved by Friedberg [1958].

2.5. *Corollary*

$\mathscr{C}$ is denumerable.

Proof. Since $\mathscr{C} = \bigcup_{n \geq 1} \mathscr{C}_n$, this follows from the fact that a denumerable union of denumerable sets is denumerable.

Explicitly, for each n let f_n be the function used in theorem 2.4 to give an enumeration of $\mathscr{C}_n$ without repetitions. Let π be the bijection $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ of theorem 1.2. Define $\theta: \mathscr{C} \to \mathbb{N}$ by

$$\theta(\phi_{f_n(m)}^{(n)}) = \pi(m, n-1).$$

Clearly θ is a bijection. □

The next theorem shows that there are functions that are not computable. The idea of the proof is as important as the result itself.

2.6. *Theorem*

There is a total unary function that is not computable.

Proof. We shall construct a total function f that is simultaneously different from every function in the enumeration $\phi_0, \phi_1, \phi_2, \ldots$ of $\mathscr{C}_1$. Explicitly, define

$$f(n) = \begin{cases} \phi_n(n) + 1 & \text{if } \phi_n(n) \text{ is defined,} \\ 0 & \text{if } \phi_n(n) \text{ is undefined.} \end{cases}$$

Notice that we have constructed f so that for each n, f differs from ϕ_n at n:

if $\phi_n(n)$ is defined, then f differs from ϕ_n in that $f(n) \neq \phi_n(n)$;

if $\phi_n(n)$ is undefined, then f differs from ϕ_n in that $f(n)$ is defined.

Since f differs from every unary computable function ϕ_n, f does not appear in the enumeration of $\mathscr{C}_1$ and is thus not itself computable. Clearly f is total. □

3. Discussion: the diagonal method

The method of constructing the function f in theorem 2.6 is an example of the *diagonal method* of construction, due to Cantor. Many readers will be familiar with this method as used in proofs of the uncountability of the set of real numbers. The underlying idea is applicable in a wide variety of situations, and is central in the proofs of many results concerning computability and decidability.

To see why the term *diagonal* is used, consider again the construction of f in theorem 2.6. Complete details of the functions $\phi_0, \phi_1, \ldots$ can be represented by the following infinite table:

	0	1	2	3	4
ϕ_0	$\phi_0(0)$	$\phi_0(1)$	$\phi_0(2)$	$\phi_0(3)$	...
ϕ_1	$\phi_1(0)$	$\phi_1(1)$	$\phi_1(2)$	$\phi_1(3)$	...
ϕ_2	$\phi_2(0)$	$\phi_2(1)$	$\phi_2(2)$	$\phi_2(3)$	...
ϕ_3	$\phi_3(0)$	$\phi_3(1)$	$\phi_3(2)$	$\phi_3(3)$	...
⋮	⋮	⋮	⋮	⋮	

We suppose that in this table the word 'undefined' is written whenever $\phi_n(m)$ is not defined.

The function f was constructed by taking the diagonal entries on this table (circled)

$$\phi_0(0), \phi_1(1), \phi_2(2), \ldots$$

and systematically changing them, obtaining

$$f(0), f(1), f(2), \ldots$$

such that $f(n)$ differs from $\phi_n(n)$, for each n. Note that there was considerable freedom in choosing the value of $f(n)$; we only had to ensure that it differed from $\phi_n(n)$. Thus

$$g(n) = \begin{cases} \phi_n(n) + 27^n & \text{if } \phi_n(n) \text{ is defined,} \\ n^2 & \text{if } \phi_n(n) \text{ is undefined,} \end{cases}$$

is another non-computable total function.

We can summarise the diagonal method as we shall be using it, in the following way. Suppose that $\chi_0, \chi_1, \chi_2, \ldots$ is an enumeration of objects of a certain kind (functions or sets of natural numbers). Then we can construct an object χ of the same kind that is different from every χ_n, using the following motto:

'*Make* χ *and* χ_n *differ at* n.'

The interpretation of the phrase *differ at n* depends on the kind of object involved. Functions may differ at n over whether they are defined, or in their values at n if defined there; with functions, there is usually freedom to construct χ so as to meet specific extra requirements; for instance, that χ be computable, or that its domain (or range) should differ from that of each χ_n.

In the case of sets, the question at n is whether or not n is a member. We illustrate the diagonal construction when sets are involved.

3.1. *Example*

Suppose that $A_0, A_1, A_2, \ldots$ is an enumeration of subsets of $\mathbb{N}$. We can define a new set B, using the diagonal motto, by

$n \in B$ if and only if $n \notin A_n$.

Clearly, for each n, $B \neq A_n$.

There are important applications of the diagonal method in the next two chapters.

3.2. *Exercises*

1. Suppose that $f(x, y)$ is a total computable function. For each m, let g_m be the computable function given by

   $g_m(y) = f(m, y)$.

   Construct a total computable function h such that for each m, $h \neq g_m$.
2. Let $f_0, f_1, \ldots$ be an enumeration of partial functions from $\mathbb{N}$ to $\mathbb{N}$. Construct a function g from $\mathbb{N}$ to $\mathbb{N}$ such that $\mathrm{Dom}(g) \neq \mathrm{Dom}(f_i)$ for each i.
3. Let f be a partial function from $\mathbb{N}$ to $\mathbb{N}$, and let $m \in \mathbb{N}$. Construct a non-computable function g such that

   $g(x) \simeq f(x)$ for $x \leq m$.

4.(*a*) (Cantor) Show that the set of all functions from $\mathbb{N}$ to $\mathbb{N}$ is not denumerable.

(*b*) Show that the set of all non-computable total functions from $\mathbb{N}$ to $\mathbb{N}$ is not denumerable.

4 The *s–m–n* theorem

In the final section of this chapter we prove a theorem that has many important uses, especially in conjunction with the main theorem of the next chapter.

Suppose that $f(x, y)$ is a computable function (not necessarily total). Then for each fixed value a of x, f gives rise to a unary computable function g_a, where

$g_a(y) \simeq f(a, y)$.

Since g_a is computable, it has an index e, say, so that

$f(a, y) \simeq \phi_e(y)$.

The next theorem shows that such an index e can be obtained *effectively* from a. This is a particular case of a more general theorem, known as the *s–m–n* theorem, which we prove below. (The reason for this name will be explained after theorem 4.3.) For most purposes in this book, the following suffices.

4.1. *Theorem* (The *s–m–n* theorem, simple form)

Suppose that $f(x, y)$ *is a computable function. There is a total computable function* $k(x)$ *such that*

$f(x, y) \simeq \phi_{k(x)}(y)$.

Proof. For each fixed a, $k(a)$ will be the code number of a program Q_a which, given initial configuration

(*)

	R_1				
	y	0	0	0	...

computes $f(a, y)$.

Let F be a program that computes f. Then for Q_a we write down F prefaced by instructions that transform the configuration (*) to

R_1	R_2				
a	y	0	0	0	...

Thus, define Q_a to be the following program

$$\left.\begin{array}{l} \mathrm{T}(1,2) \\ \mathrm{Z}(1) \\ \left.\begin{array}{l}\mathrm{S}(1)\\ \vdots \\ \mathrm{S}(1)\end{array}\right\} a \text{ times} \\ F \end{array}\right.$$

Now define

$k(a)$ = the code number of the program Q_a.

Since F is fixed, and from the fact that our numbering γ of programs is effective, we see that k is an effectively computable function. Hence, by Church's thesis, k is computable. By construction

$$\phi_{k(a)}(y) \simeq f(a, y)$$

for each a. □

The *s–m–n* theorem is sometimes called the *Parametrisation theorem* because it shows that an index for a computable function (such as g_a in the discussion above) can be found effectively from a parameter (such as a) on which it effectively depends.

Before giving the full *s–m–n* theorem we give some simple illustrations of the use of theorem 4.1 in effectively indexing certain sequences of computable functions or their domains or ranges.

4.2. *Examples*

1. Let $f(x, y) = y^x$. By theorem 4.1 there is a total computable k such that $\phi_{k(x)}(y) = y^x$. Hence, for each fixed n, $k(n)$ is an index for the function y^n.
2. Let $f(x, y) = \begin{cases} y & \text{if } y \text{ is a multiple of } x \\ \text{undefined} & \text{otherwise.} \end{cases}$

   Then f is computable, so let k be a computable function such that $\phi_{k(x)}(y) \simeq f(x, y)$. Then, for each fixed n

   $\phi_{k(n)}(y)$ is defined iff y is a multiple of n

   iff y is in the range of $\phi_{k(n)}$.

   i.e.

   $W_{k(n)} = n\mathbb{N}$ (= the set of all multiples of n)

   $= E_{k(n)}$.

   So we have an effective indexing of the sequence of sets $(n\mathbb{N})$ as (i) the domains of computable functions, (ii) the ranges of computable functions.

One obvious way to generalise theorem 4.1 is to replace the single variables x, y by m- and n-tuples $\boldsymbol{x}$ and $\boldsymbol{y}$ respectively. We can also reflect the fact that the function k defined in the proof of theorem 4.1 depended effectively on a particular program for the original function f. Thus, instead of considering a fixed computable function $f(x, y)$ we consider a general computable function $\phi_e^{(m+n)}(\boldsymbol{x}, \boldsymbol{y})$, and the question of effectively finding, for each e and $\boldsymbol{x}$, a number z such that

$$\phi_e^{(m+n)}(\boldsymbol{x}, \boldsymbol{y}) \simeq \phi_z^{(n)}(\boldsymbol{y}).$$

4.3. *Theorem* (The *s–m–n* theorem)

For each $m, n \geq 1$ there is a total computable $(m+1)$-ary function $s_n^m(e, \boldsymbol{x})$ such that

$$\phi_e^{(m+n)}(\boldsymbol{x}, \boldsymbol{y}) \simeq \phi_{s_n^m(e,\boldsymbol{x})}^{(n)}(\boldsymbol{y}).$$

Proof. We generalise the proof of theorem 4.1.

For any $i \geq 1$ let $Q(i, x)$ be the subroutine

$$\left.\begin{array}{l}\mathrm{Z}(i)\\ \mathrm{S}(i)\\ \vdots \\ \mathrm{S}(i)\end{array}\right\} x \text{ times}$$

that replaces the current contents of R_i by x. Then for fixed m, n define $s_n^m(e, \boldsymbol{x})$ to be the code number of the following program:

$$\left.\begin{array}{l} \mathrm{T}(n, m+n) \\ \vdots \\ \mathrm{T}(2, m+2) \\ \mathrm{T}(1, m+1) \\ Q(1, x_1) \\ Q(2, x_2) \\ \vdots \\ Q(m, x_m) \end{array}\right\}$$

This part of the program transforms any configuration

R_1		R_n			
y_1	...	y_n	0	0	...

into

R_1		R_m	R_{m+1}		R_{m+n}		
x_1	...	x_m	y_1	...	y_n	0	...

P_e

From this explicit definition, and the effectiveness of γ and γ^{-1}, we get that s_n^m is effectively computable, hence computable, by Church's thesis. □

The notation s_n^m for the function given by theorem 4.3 has given rise to the standard description of this result as the *s–m–n* theorem. We will also use this name to describe the simpler version given in theorem 4.1.

It is not hard to see that the function s_n^m as defined above is in fact primitive recursive. With a little thought it is also possible to see that for each m there is a function s^m (also primitive recursive) that suffices in theorem 4.3 for all n. See the exercises 4.4(5) below.

4.4. *Exercises*

1. Show that there is a total computable function k such that for each n, $k(n)$ is an index of the function $[\sqrt[n]{x}]$.
2. Show that there is a total computable function k such that for each n, $W_{k(n)}$ = the set of perfect nth powers.
3. Let $n \geq 1$. Show that there is a total computable function s such that
   $W_{s(x)}^{(n)} = \{(y_1, \ldots, y_n) : y_1 + y_2 + \ldots + y_n = x\}$
4. Show that the functions s_n^m defined in theorem 4.3 are all primitive recursive.
5. Show that for each m there is a total $(m+1)$-ary computable function s^m such that for all n
   $\phi_e^{(m+n)}(\boldsymbol{x}, \boldsymbol{y}) \simeq \phi_{s^m(e,\boldsymbol{x})}^{(n)}(\boldsymbol{y})$
   where $\boldsymbol{x}$, $\boldsymbol{y}$ are m- and n-tuples respectively.
   (*Hint.* Consider the definition of $s_n^m(e, \boldsymbol{x})$ given in the proof of theorem 4.3. The only way in which n was used was in determining how many of the $r_1, r_2, \ldots$ to transfer to $R_{m+1}, R_{m+2}, \ldots$. Now recall that the effect of P_e depends only on the original contents of $R_1, \ldots, R_{\rho(P_e)}$, where ρ is the function defined in chapter 2 § 2; $\rho(P_e)$ is independent of n.) Show further that there is such a function s^m that is primitive recursive.

5
Universal programs

In this chapter we establish the somewhat surprising result that there are *universal programs*; i.e. programs that in a sense embody all other programs. This result is one of the twin pillars that support computability theory (the other is the *s–m–n* theorem); both rest on the numbering of programs given in chapter 4.

Important among the applications of universal programs is the construction of specific non-computable functions and undecidable predicates, a topic pursued in chapter 6. We give a foretaste of such applications in § 2 of this chapter; we also use a universal program to construct a total computable function that is not primitive recursive, as promised in chapter 3.

The final section of this chapter is devoted to some illustrations of the use of the *s–m–n* theorem in conjunction with universal programs to show that certain operations on the indices of computable functions are effective (a foretaste of the topic of chapter 10).

1. Universal functions and universal programs

Consider the function $\psi(x, y)$ defined by

$$\psi(x, y) \simeq \phi_x(y).$$

There is an obvious sense in which the single function ψ embodies all the unary computable functions $\phi_0, \phi_1, \phi_2, \ldots$, since for any particular m, the function g given by

$$g(y) \simeq \psi(m, y)$$

is just the computable function ϕ_m. Thus we describe ψ as the *universal function* for unary computable functions. Generally, we make the following definition.

1.1. *Definition*

The *universal function for n-ary computable functions* is the $(n+1)$-ary function $\psi_U^{(n)}$ defined by

$$\psi_U^{(n)}(e, x_1, \ldots, x_n) \simeq \phi_e^{(n)}(x_1, \ldots, x_n).$$

We write ψ_U for $\psi_U^{(1)}$.

The question arises, is ψ_U (or, generally, $\psi_U^{(n)}$) a computable function? If so, then any program P that computes ψ_U would appear to embody all other programs, and P would be aptly called a *universal program*. At first, perhaps, the existence of a universal program seems unlikely. Nevertheless, it is not hard to see that ψ_U is indeed computable. The point is that a universal program P does *not* need to contain all other programs P_e in itself; P only needs the ability to *decode* any number e and hence mimic P_e.

1.2. *Theorem*

For each n, the universal function $\psi_U^{(n)}$ is computable.

Proof. Fix n, and suppose that we are given an index e and an n-tuple $\boldsymbol{x}$. An informal procedure for computing $\psi_U^{(n)}(e, \boldsymbol{x})$ is as follows:

'Decode the number e and write out the program P_e. Now mimic the computation $P_e(\boldsymbol{x})$ step by step, at each step writing down the configuration of the registers and the next instruction to be obeyed (as was done in example 1-2.1). If and when this computation stops, then the required value $\psi_U^{(n)}(e, \boldsymbol{x})$ is the number currently in R_1.'

We could conclude immediately (using Church's thesis) that $\psi_U^{(n)}(e, \boldsymbol{x})$ is computable. Because of the importance of this theorem, however, we prefer to outline the beginnings of a formal proof and then make a rather less sweeping appeal to Church's thesis. (For the sake of completeness of our exposition we shall provide the rest of the formal proof in an appendix to this chapter.)

The plan for a formal proof is to show first how to use a single number σ to code the current situation during a computation; then we show that there is a *computable* function expressing the dependence of σ on (*a*) the program number e, (*b*) the input $\boldsymbol{x}$, (*c*) the number of steps of the computation that have been completed. We will see that this suffices to prove the theorem.

Let us return, then, to the computation $P_e(\boldsymbol{x})$ considered above. As we have seen in examples, the current situation during a computation is completely specified by (i) the current configuration of the registers

$r_1, r_2, r_3, \ldots$ and (ii) the number j of the next instruction in the computation. Since only finitely many of the numbers r_i are not zero, the current configuration can be specified by the single number

$$c = 2^{r_1}3^{r_2}\ldots = \prod_{i \geq 1} p_i^{r_i}.$$

(Recall that p_i is the ith prime number.) We call this number the *configuration code* or just the *configuration* if there is no ambiguity. Note that the contents r_i of R_i can be easily recovered from c; in fact $r_i = (c)_i$ (using the function of theorem 2-4.15(*d*)).

The complete description of the current situation can now be coded by the single number $\sigma = \pi(c, j)$, which we call the current *state* of the computation $P_e(\boldsymbol{x})$. (Here π is the pairing function used in the proof of theorem 4-1.2.) We will make the convention that if the computation has stopped, then $j = 0$ and c is the final configuration. Note that $c = \pi_1(\sigma)$ and $j = \pi_2(\sigma)$ where π_1, π_2 are the computable functions defined in theorem 4-1.2.

Now c, j, σ change during the computation; their dependence on the program number e, the input $\boldsymbol{x}$ and the number t of steps completed is expressed by defining the following $(n+2)$-ary functions:

(1) $c_n(e, \boldsymbol{x}, t)$ = the configuration after t steps of $P_e(\boldsymbol{x})$ have been completed (= the final configuration if $P_e(\boldsymbol{x})\downarrow$ in t or fewer steps).

(2) $$j_n(e, \boldsymbol{x}, t) = \begin{cases} \text{the number}^1 \text{ of the next instruction for } P_e(\boldsymbol{x}) \text{ when } t \text{ steps have been completed,} & \text{if } P_e(\boldsymbol{x}) \text{ has not stopped after } t \text{ or fewer steps,} \\ 0 & \text{if } P_e(\boldsymbol{x})\downarrow \text{ in } t \text{ or fewer steps.} \end{cases}$$

(3) $\sigma_n(e, \boldsymbol{x}, t)$ = the state of the computation $P_e(\boldsymbol{x})$ after t steps $= \pi(c_n(e, \boldsymbol{x}, t), j_n(e, \boldsymbol{x}, t))$.

The aim now is to show that σ_n (and hence c_n and j_n) are computable functions. To see why this is sufficient, suppose that this has been done. Clearly, if the computation $P_e(\boldsymbol{x})$ stops it does so in $\mu t(j_n(e, \boldsymbol{x}, t) = 0)$ steps; then the final configuration is $c_n(e, \boldsymbol{x}, \mu t(j_n(e, \boldsymbol{x}, t) = 0))$, and so we

[1] We mean here the number j such that the next instruction I is the jth instruction of P_e; we do *not* mean the code number $\beta(I)$.

have

$$\psi_U^{(n)}(e, \boldsymbol{x}) \simeq (c_n(e, \boldsymbol{x}, \mu t(j_n(e, \boldsymbol{x}, t) = 0)))_1.$$

Thus, if c_n and j_n are computable, so is $\psi_U^{(n)}$ (using substitution and minimalisation) and our proof is complete.

We now use Church's thesis to show that σ_n (and hence c_n and j_n) are computable. We have the following informal algorithm for obtaining $\sigma_n(e, \boldsymbol{x}, t+1)$ effectively from $\sigma_n(e, \boldsymbol{x}, t)$ and e:

'Decode $\sigma_n(e, \boldsymbol{x}, t)$ to find the numbers $c = c_n(e, \boldsymbol{x}, t)$ and $j = j_n(e, \boldsymbol{x}, t)$. If $j = 0$, then $\sigma_n(e, \boldsymbol{x}, t+1) = \sigma_n(e, \boldsymbol{x}, t)$. Otherwise, write out the configuration coded by c,

(*)

$(c)_1$	$(c)_2$	$(c)_3$	...	$(c)_m$	0	0	...

say, and by decoding e write out the program P_e. Now find the jth instruction in P_e and operate with it on the configuration (*), producing a new configuration with code c' say. Find also the number j' of the new next instruction (with $j' = 0$ if the computation has now terminated). Then we have

$$\sigma_n(e, \boldsymbol{x}, t+1) = \pi(c', j').$$

This shows informally that $\sigma_n(e, \boldsymbol{x}, t)$ is computable by recursion in t, since for $t = 0$ we have

$$\sigma_n(e, \boldsymbol{x}, 0) = \pi(2^{x_1}3^{x_2} \ldots p_n^{x_n}, 1)$$

to start the recursion off. Hence, by Church's thesis, σ_n is computable, and our theorem is now proved. □

Note. Since this theorem is so basic to further development, we provide in the appendix a complete formal proof that σ_n (and hence $\psi_U^{(n)}$) is computable. This then provides further evidence for Church's thesis. (Our formal proof also gives us the extra information that σ_n is actually *primitive* recursive.)

From the proof of this theorem we obtain:

1.3. *Corollary*

For each $n \geq 1$, the following predicates are decidable.

(*a*) $S_n(e, \boldsymbol{x}, y, t) \equiv$ '$P_e(\boldsymbol{x})\downarrow y$ *in* t *or fewer steps*',

(*b*) $H_n(e, \boldsymbol{x}, t) \equiv$ '$P_e(\boldsymbol{x})\downarrow$ *in* t *or fewer steps*'.

Proof. (*a*) $S_n(e, \boldsymbol{x}, y, t) \equiv$ '$j_n(e, \boldsymbol{x}, t) = 0$ and $(c_n(e, \boldsymbol{x}, t))_1 = y$'.

(*b*) $H_n(e, \boldsymbol{x}, t) \equiv$ '$j_n(e, \boldsymbol{x}, t) = 0$'. □

The significance of the next corollary is discussed in the first note below.

1.4. *Corollary* (Kleene's normal form theorem)

There is a total computable function $U(x)$ and for each $n \geq 1$ a decidable predicate $T_n(e, \boldsymbol{x}, z)$ such that

(*a*) $\phi_e^{(n)}(\boldsymbol{x})$ *is defined if and only if* $\exists z\, T_n(e, \boldsymbol{x}, z)$,

(*b*) $\phi_e^{(n)}(\boldsymbol{x}) \simeq U(\mu z\, T_n(e, \boldsymbol{x}, z))$.

Proof. To discover whether $\phi_e^{(n)}(\boldsymbol{x})$ is defined, and the value if it is, we need to search for a pair of numbers y, t such that $S_n(e, \boldsymbol{x}, y, t)$. We have the μ-operator that enables us to search effectively for a single number having a given property. To use this in searching for a pair of numbers, we can think of a single number z as coding the pair of numbers $(z)_1$ and $(z)_2$. Then, as z runs through $\mathbb{N}$, the pair $((z)_1, (z)_2)$ runs through $\mathbb{N} \times \mathbb{N}$. So we define

$$T_n(e, \boldsymbol{x}, z) \equiv S_n(e, \boldsymbol{x}, (z)_1, (z)_2).$$

For (*a*), suppose that $\phi_e^{(n)}(\boldsymbol{x})$ is defined; then there are y, t such that $S_n(e, \boldsymbol{x}, y, t)$, so putting $z = 2^y 3^t$ we have $T_n(e, \boldsymbol{x}, z)$.

Conversely, if there is z such that $T_n(e, \boldsymbol{x}, z)$, then from the definition of T_n, $P_e(\boldsymbol{x})\downarrow$; i.e. $\phi_e^{(n)}(\boldsymbol{x})$ is defined.

For (*b*), it is clear from the definition of T_n that if $\phi_e^{(n)}(\boldsymbol{x})$ is defined, then for any z such that $T_n(e, \boldsymbol{x}, z)$, we have $\phi_e^{(n)}(\boldsymbol{x}) = (z)_1$. So if we put $U(z) = (z)_1$ then

$$\phi_e^{(n)}(\boldsymbol{x}) \simeq U(\mu z\, T_n(e, \boldsymbol{x}, z)). \quad \square$$

Notes

1. From the appendix to this chapter it follows that the functions c_n and j_n are primitive recursive. Hence, the predicates S_n, H_n, T_n in corollaries 1.3 and 1.4 are also primitive recursive. Thus, in particular, the Kleene normal form theorem shows that every computable function (or partial recursive function) can be obtained from primitive recursive functions by using at most one application of the μ-operator. The theorem gives, moreover, a standard way of doing this.

2. The technique of searching for pairs of numbers by thinking of a single number z as coding the pair $(z)_1$, $(z)_2$ (as used in the proof of corollary 1.4) is often used in computability theory. We give an exercise needing this technique below (exercise 1.5(1)).

The technique can also be used in searching for sequences $(x_1, x_2, \ldots, x_n)$ for any $n > 1$.

1.5. *Exercises*

1. (i) Show that there is a decidable predicate $Q(x, y, z)$ such that
   (*a*) $y \in E_x$ if and only if $\exists z\, Q(x, y, z)$,
   (*b*) if $y \in E_x$, and $Q(x, y, z)$, then $\phi_x((z)_1) = y$.
   (ii) Deduce that there is a computable function $g(x, y)$ such that
   (*a*) $g(x, y)$ is defined if and only if $y \in E_x$.
   (*b*) if $y \in E_x$, then $g(x, y) \in W_x$ and $\phi_x(g(x, y)) = y$; i.e. $g(x, y) \in \phi_x^{-1}(\{y\})$.
   (iii) Deduce that if f is a computable injective function (not necessarily total or surjective) then f^{-1} is computable. (cf. exercise 2-5.4 (1)).
2. (cf. example 3-7.1(2)) Suppose that f and g are unary computable functions; assuming that T_1 has been formally proved to be decidable, prove formally that the function $h(x)$ defined by

$$h(x) = \begin{cases} 1 \text{ if } x \in \mathrm{Dom}(f) \text{ or } x \in \mathrm{Dom}(g), \\ \text{undefined otherwise,} \end{cases}$$

   is computable.

2. Two applications of the universal program

We illustrate now the use of the computability of universal functions in diagonal constructions. This kind of application will be explored more thoroughly in the next chapter.

2.1. *Theorem*

The problem 'ϕ_x is total' is undecidable.

Proof. Let g be the characteristic function of this problem; i.e.

$$g(x) = \begin{cases} 1 & \text{if } \phi_x \text{ is total,} \\ 0 & \text{if } \phi_x \text{ is not total.} \end{cases}$$

We must show that g is not computable. To achieve this, we use the diagonal method to construct a total function f that is different from every computable function, yet such that if g is computable, then so is f. Explicitly, define f by

$$f(x) = \begin{cases} \phi_x(x) + 1 & \text{if } \phi_x \text{ is total,} \\ 0 & \text{if } \phi_x \text{ is not total.} \end{cases}$$

Clearly, f is total and differs from every computable function ϕ_x. Now, using g and ψ_U we can write f as follows:

$$f(x) = \begin{cases} \psi_U(x, x) + 1 & \text{if } g(x) = 1, \\ 0 & \text{if } g(x) = 0. \end{cases}$$

Now suppose that g is computable; since ψ_U is computable, then, by Church's thesis, so is f, which is a contradiction. Hence g is not computable. □

Our second application here fulfils the promise made in chapter 3 § 3.

2.2. *Theorem*

There is a total computable function that is not primitive recursive.

Proof. We give an informal proof. Recall that the primitive recursive functions are those functions that can be built up from the basic functions by a sequence of applications of the operations of substitution and recursion. Thus each primitive recursive function can be specified by a *plan* that indicates the basic functions used and the exact sequence of operations performed in its construction. To describe such a plan it is convenient to adopt some notation such as the following:

$\mathrm{Sub}(f; g_1, g_2, \ldots, g_m)$ denotes the function obtained by substituting $g_1, \ldots, g_m$ into f (assuming that f is m-ary, and $g_1, \ldots, g_m$ are n-ary for some n);

$\mathrm{Rec}(f, g)$ denotes the function obtained from f and g by recursion (assuming that f is n-ary and g is $(n+2)$-ary for some n).

If we write S for the function $x + 1$, then we have, for example, the following plan for the function $f(x) = x^2$. We use letters $g_1, \ldots, g_4$ to denote intermediate functions.

Plan	Explanation of the steps
Step 1. $g_1 = \mathrm{Sub}(\mathrm{S}; \mathrm{U}_3^3)$.	$g_1(x, y, z) = \mathrm{U}_3^3(x, y, z) + 1 = z + 1$.
Step 2. $g_2 = \mathrm{Rec}(\mathrm{U}_1^1, g_1)$.	$\begin{cases} g_2(x, 0) = \mathrm{U}_1^1(x) = x, \\ g_2(x, y+1) = g_1(x, y, g_2(x, y)) \\ \qquad = g_2(x, y) + 1. \end{cases}$
	So $g_2(x, y) = x + y$.
Step 3. $g_3 = \mathrm{Sub}(g_2; \mathrm{U}_1^3, \mathrm{U}_3^3)$.	$g_3(x, y, z) = g_2(x, z) = x + z$.
Step 4. $g_4 = \mathrm{Rec}(\mathbf{0}, g_3)$.	$\begin{cases} g_4(x, 0) = 0, \\ g_4(x, y+1) = g_3(x, y, g_4(x, y)) \\ \qquad = x + g_4(x, y). \end{cases}$
	So $g_4(x, y) = xy$.
Step 5. $f = \mathrm{Sub}(g_4; \mathrm{U}_1^1, \mathrm{U}_1^1)$.	$f(x) = g_4(x, x) = x^2$.

Thus a plan is somewhat akin to a program, in that it is a finite and explicit specification of a function.

We now restrict our attention to plans for unary primitive recursive functions. As with programs, we can number these plans in an effective

way, so that we may then define

θ_n = the unary primitive recursive function defined by plan number n.

Then $\theta_0, \theta_1, \theta_2, \ldots$ is an effective enumeration of all unary primitive recursive functions.

From chapter 2 we know that every primitive recursive function is computable. Hence there is a total function p such that for each n, $p(n)$ is the number of a program that computes θ_n; i.e.

$$\theta_n = \phi_{p(n)}.$$

Now the crucial point is that we can find such a function p that is *computable*. We argue informally using Church's thesis.

Recall the proofs of theorems 2-3.1 and 2-4.4. There we showed explicitly how to obtain a program for the function

$$\mathrm{Sub}(f; g_1, \ldots, g_m)$$

given programs for $f, g_1, \ldots, g_m$; and also, how to obtain a program for the function

$$\mathrm{Rec}(f, g)$$

given programs for f and g. (In the next section (example 3.1(5)) we use the s–m–n theorem to show in detail that for each n there is a computable function r such that for any e_1, e_2 an index for $\mathrm{Rec}(\phi_{e_1}^{(n)}, \phi_{e_2}^{(n+2)})$ is given by $r(e_1, e_2)$; we can do a similar thing for substitution (see exercise 3.2(5a)).) We also have explicit programs for the basic functions. Hence, given a plan for a primitive recursive function f involving intermediate functions $g_1, \ldots, g_k$, say, we can effectively find programs for $g_1, g_2, \ldots, g_k$, and finally f. Thus there is an effectively computable function p such that

$$\theta_n = \phi_{p(n)}.$$

By Church's thesis, p is computable.

Now for the payoff! From p and the universal function ψ_U we can define a total computable function g that differs from every primitive recursive function θ_n. We use a diagonal construction as follows:

$$\begin{aligned} g(x) &= \theta_x(x) + 1 \\ &= \phi_{p(x)}(x) + 1 \\ &= \psi_U(p(x), x) + 1. \end{aligned}$$

From this we see immediately that g is a total function that is not primitive recursive; but g *is* computable, by the computability of ψ_U and p. □

3. Effective operations on computable functions

In this section we illustrate another important application of the computability of the universal functions, this time in conjunction with the s–m–n theorem.

Consider the following operations on computable functions or their domains:

(a) combining ϕ_x and ϕ_y to form the product $\phi_x\phi_y$;

(b) forming the union $W_x \cup W_y$ from W_x and W_y.

We are all familiar with a wide variety of operations of a similar kind, usually defined explicitly like these. Is there any sense in which these operations can be thought of as *effective* operations? Inasmuch as these are operations involving infinite objects (functions or sets), they seem to lie outside the scope of even our informal notion of computability, which implicitly applies only to finite objects. Nevertheless, we will see, for instance, that an *index* for the function $\phi_x\phi_y$ can be obtained effectively from the indices x, y. In the following examples and exercises we see that many other operations are effective when viewed thus as operations on indices of the objects involved. (We will return to the topic of effective operations on functions in chapter 10.)

3.1. *Examples*

1. There is a total computable function $s(x, y)$ such that for all x, y $\phi_{s(x,y)} = \phi_x\phi_y$.

   Proof. Let $f(x, y, z) \simeq \phi_x(z)\phi_y(z)$
   $\simeq \psi_U(x, z)\psi_U(y, z)$.

   Thus f is computable, so by the s–m–n theorem there is a total computable function $s(x, y)$ such that $f(x, y, z) \simeq \phi_{s(x,y)}(z)$; hence $\phi_{s(x,y)} = \phi_x\phi_y$.
2. Taking $g(x) = s(x, x)$, with s as an example 1, we have $(\phi_x)^2 = \phi_{g(x)}$.
3. There is a total computable function $s(x, y)$ such that $W_{s(x,y)} = W_x \cup W_y$.

   Proof. Let $f(x, y, z) = \begin{cases} 1 & \text{if } z \in W_x \text{ or } z \in W_y, \\ \text{undefined} & \text{otherwise.} \end{cases}$

   By Church's thesis and the computability of ψ_U, f is computable; so there is a total computable function $s(x, y)$ such that $f(x, y, z) \simeq \phi_{s(x,y)}(z)$. Then clearly $W_{s(x,y)} = W_x \cup W_y$.

4. *Effectiveness of taking inverses.* Let $g(x,y)$ be a computable function as described in exercise 1.5; i.e. such that
   (a) $g(x, y)$ is defined if and only if $y \in E_x$,
   (b) if $y \in E_x$, then $g(x, y) \in W_x$ and $\phi_x(g(x, y)) = y$.
   By the s–m–n theorem there is a total computable function k such that $g(x, y) \simeq \phi_{k(x)}(y)$. Then from (a) and (b) we have
   (a') $W_{k(x)} = E_x$,
   (b') (i) $E_{k(x)} \subseteq W_x$,
   (ii) if $y \in E_x$, then $\phi_x(\phi_{k(x)}(y)) = y$.
   Hence, if ϕ_x is injective, then $\phi_{k(x)} = \phi_x^{-1}$ and $E_{k(x)} = W_x$.
5. *Effectiveness of recursion.* Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ and consider the $(n+3)$-ary function f defined by
   $$f(e_1, e_2, \boldsymbol{x}, 0) \simeq \phi_{e_1}^{(n)}(\boldsymbol{x}),$$
   $$f(e_1, e_2, \boldsymbol{x}, y+1) \simeq \phi_{e_2}^{(n+2)}(\boldsymbol{x}, y, f(e_1, e_2, \boldsymbol{x}, y)).$$
   Then using the universal functions $\psi_U^{(n)}$ and $\psi_U^{(n+2)}$ to rewrite the expressions on the right, this is a definition by recursion from computable functions, so f is computable. Moreover, for fixed e_1, e_2 the function $g(\boldsymbol{x}, y) \simeq f(e_1, e_2, \boldsymbol{x}, y)$ is the function obtained from $\phi_{e_1}^{(n)}$ and $\phi_{e_2}^{(n+2)}$ by recursion.
   By the s–m–n theorem there is a total computable function $r(e_1, e_2)$ such that
   $$\phi_{r(e_1,e_2)}^{(n+1)}(\boldsymbol{x}, y) \simeq f(e_1, e_2, \boldsymbol{x}, y).$$
   Hence $r(e_1, e_2)$ is an index for the $(n+1)$-ary function obtained from $\phi_{e_1}^{(n)}$ and $\phi_{e_2}^{(n+2)}$ by recursion. In the notation of theorem 2.2
   $$\phi_{r(e_1,e_2)}^{(n+1)} = \mathrm{Rec}(\phi_{e_1}^{(n)}, \phi_{e_2}^{(n+2)})$$
   for all e_1, e_2.

The following exercises give more examples of the use of the s–m–n theorem in showing that operations are effective on indices.

3.2. *Exercises*

1. Show that there is a total computable function $k(e)$ such that for any e, if ϕ_e is the characteristic function for a decidable predicate $M(x)$, then $\phi_{k(e)}$ is the characteristic function for 'not $M(x)$'.
2. Show that there is a total computable function $k(x)$ such that for every x, $E_{k(x)} = W_x$.
3. Show that there is a total computable function $s(x, y)$ such that for all x, y, $E_{s(x,y)} = E_x \cup E_y$.
4. Suppose that $f(x)$ is computable; show that there is a total computable function $k(x)$ such that for all x, $W_{k(x)} = f^{-1}(W_x)$.

5. Prove the equivalent of example 5 above for the operations of substitution and minimalisation, namely:
   (a) Fix $m, n \geq 1$; there is a total computable function $s(e, e_1, \ldots, e_m)$ such that (in the notation of theorem 2.2)
   $$\phi_{s(e,e_1,\ldots,e_m)}^{(n)} = \mathrm{Sub}(\phi_e^{(m)}; \phi_{e_1}^{(n)}, \phi_{e_2}^{(n)}, \ldots, \phi_{e_m}^{(n)}).$$
   (b) Fix $n \geq 1$; there is a total computable function $k(e)$ such that for all e,
   $$\phi_{k(e)}^{(n)}(\boldsymbol{x}) \simeq \mu y(\phi_e^{(n+1)}(\boldsymbol{x}, y) = 0).$$
   (We could extend the notation of theorem 2.2 in the obvious way and write $\phi_{k(e)}^{(n)} = \mathrm{Min}(\phi_e^{(n+1)})$.)

Appendix
Computability of the function σ_n

In this appendix we give a formal proof that the function σ_n defined in the proof of theorem 1.2 is computable (in fact, primitive recursive) thus completing a formal proof of the computability of the universal function $\psi_U^{(n)}$.

Theorem.
The function σ_n is primitive recursive.

Proof. For the definition of σ_n and the functions c_n and j_n coded by σ_n, refer to the proof of theorem 1.2.

We define two functions 'config' and 'nxt' that describe the changes in c_n and j_n during computations. Suppose that at some stage during computation under P_e the current state is $\sigma = \pi(c, j)$, and suppose that P_e has s instructions. We can describe the effect of the jth instruction of P_e on the state σ by defining

$$\mathrm{config}(e, \sigma) = \begin{cases} \text{the new configuration after the } j\text{th instruction of } P_e \text{ has been obeyed,} & \text{if } 1 \leq j \leq s, \\ c & \text{otherwise.} \end{cases}$$

$$\mathrm{nxt}(e, \sigma) = \begin{cases} \text{the number of the next instruction for the computation, after the } j\text{th instruction of } P_e \text{ has been executed on the configuration } c, & \text{if } 1 \leq j \leq s \text{ and this next instruction exists in } P_e, \\ 0 & \text{otherwise.} \end{cases}$$

Now σ_n can be obtained from config and nxt by the following recursion equations:

$$\sigma_n(e, \boldsymbol{x}, 0) = \pi(2^{x_1}3^{x_2}\ldots p_n^{x_n}, 1),$$

$$\sigma_n(e, \boldsymbol{x}, t+1) = \pi(\text{config}(e, \sigma_n(e, \boldsymbol{x}, t)), \text{nxt}(e, \sigma_n(e, \boldsymbol{x}, t))).$$

Thus, σ_n is primitive recursive if config and nxt are primitive recursive; we proceed to show that they are.

We must be careful now to distinguish between the code number $\beta(I)$ of an instruction I and its number in any program in which it occurs (i.e. the number j such that I is the jth instruction). We will always use the term code number when $\beta(I)$ is intended.

It is sufficient to establish that the following four functions are primitive recursive:

(1) $\ln(e)$ = the number of instructions in program P_e;

(2) $$\text{gn}(e, j) = \begin{cases} \text{the code number of the } j\text{th} \\ \text{instruction in } P_e, & \text{if } 1 \le j \le \ln(e), \\ 0 & \text{otherwise;} \end{cases}$$

(3) $\text{ch}(c, z)$ = the configuration resulting when the configuration c is operated on by the instruction with code number z;

(4) $$\nu(c, j, z) = \begin{cases} \text{the number } j' \text{ of the} \\ \text{'next instruction for the} \\ \text{computation' when the} \\ \text{configuration } c \text{ is operated on} \\ \text{by the instruction with code} & \text{if } j > 0, \\ z, \text{ and this occurs as the} \\ j\text{th instruction in a program,} \\ 0 & \text{otherwise} \end{cases}$$

(The 'next instruction for the computation' here is as defined in chapter 1 § 2, so $j' = j+1$ or, if I_j is a jump instruction $\text{J}(m_1, m_2, q)$ we may have $j' = q$.)

If these four functions are primitive recursive, then remembering that $\sigma = \pi(c, j)$ where $c = \pi_1(\sigma)$ and $j = \pi_2(\sigma)$ we have

$$\text{config}(e, \sigma) = \begin{cases} \text{ch}(\pi_1(\sigma), \text{gn}(e, \pi_2(\sigma))) & \text{if } 1 \le \pi_2(\sigma) \le \ln(e), \\ \pi_1(\sigma) & \text{otherwise.} \end{cases}$$

$$\text{nxt}(e, \sigma) = \begin{cases} \nu(\pi_1(\sigma), \pi_2(\sigma), \text{gn}(e, \pi_2(\sigma))) \\ \qquad \text{if this number is} \le \ln(e), \\ 0 \qquad \text{otherwise.} \end{cases}$$

Thus config and nxt are primitive recursive, by the methods of chapter 2.

It remains to show that the functions (1)–(4) above are primitive recursive. A sequence of auxiliary functions is needed to decode the code numbers of programs and instructions. We use freely the standard functions and techniques of chapter 2 §§ 1–4, together with the functions defined in chapter 4 § 1 for coding instructions and programs.

(5) The functions $\alpha(i, x)$, $l(x)$, $b(i, x)$ and $a(i, x)$ of exercise 2-4.16(5) are primitive recursive.

Proof. (i) $x = \sum_{i=0}^{\infty} \alpha(i, x)2^i$; so we have $\text{qt}(2^i, x) = \alpha(i, x) + \alpha(i+1, x)2 + \ldots$ and hence $\alpha(i, x) = \text{rm}(2, \text{qt}(2^i, x))$.

(ii) $l(x)$ = number of is such that $\alpha(i, x) = 1$; hence

$$l(x) = \sum_{i<x} \alpha(i, x).$$

(iii) If $x > 0$, $x = 2^{b(1,x)} + 2^{b(2,x)} + \ldots + 2^{b(l(x),x)}$; thus, if $1 \le i \le l(x)$, then $b(i, x)$ is the ith index k such that $\alpha(k, x) = 1$. Hence

$$b(i, x) = \begin{cases} \mu y < x \left(\sum_{k \le y} \alpha(k, x) = i \right) & \text{if } 1 \le i \le l(x) \text{ and } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

(iv) From the definition:

$$a(i, x) = b(i, x) \qquad (i = 0, 1)$$

$$a(i+1, x) = (b(i+1, x) \dot{-} b(i, x)) \dot{-} 1 \qquad (i \ge 1)$$

From the above explicit formulae, using the techniques of chapter 2, these functions are all primitive recursive.

(6) The functions $\ln(e)$ and $\text{gn}(e, j)$ are primitive recursive.

Proof. From the definitions of the coding function γ:

$$\ln(e) = l(e+1),$$

$$\text{gn}(e, j) = a(j, e+1),$$

where l and a are the functions in (5).

(7) There are primitive recursive functions $u, u_1, u_2, v_1, v_2, v_3$ such that:

if $z = \beta(\text{Z}(m))$, then $u(z) = m$,

if $z = \beta(\text{S}(m))$, then $u(z) = m$,

if $z = \beta(\text{T}(m_1, m_2))$, then $u_1(z) = m_1$ and $u_2(z) = m_2$,

if $z = \beta(\text{J}(m_1, m_2, q))$, then $v_1(z) = m_1$, $v_2(z) = m_2$, and $v_3(z) = q$.

Proof. From the definition of β, and writing $(z/4)$ for $\mathrm{qt}(4, z)$, take

$$u(z) = (z/4)+1,$$

$$u_1(z) = \pi_1(z/4)+1,$$

$$u_2(z) = \pi_2(z/4)+1,$$

$$v_1(z) = \pi_1(\pi_1(z/4))+1,$$

$$v_2(z) = \pi_2(\pi_1(z/4))+1,$$

$$v_3(z) = \pi_2(z/4)+1.$$

(8) The following functions are primitive recursive:

(i) $\mathrm{zero}(c, m) =$ the change in the configuration c effected by instruction $\mathrm{Z}(m)$,
$= \mathrm{qt}(\mathrm{p}_m^{(c)_m}, c)$.

(ii) $\mathrm{suc}(c, m) =$ the change in the configuration c effected by instruction $\mathrm{S}(m)$,
$= c\mathrm{p}_m$.

(iii) $\mathrm{transfer}(c, m, n) =$ the change in the configuration c effected by instruction $\mathrm{T}(m, n)$,
$= \mathrm{qt}(\mathrm{p}_n^{(c)_n}, c\mathrm{p}_n^{(c)_m})$.

(9) The function $\mathrm{ch}(c, z)$ (defined in (3) above) is primitive recursive.

Proof.

$$\mathrm{ch}(c, z) = \begin{cases} \mathrm{zero}(c, u(z)) & \text{if } \mathrm{rm}(4, z) = 0 \text{ (i.e. } z \text{ is the code of a zero instruction),} \\ \mathrm{suc}(c, u(z)) & \text{if } \mathrm{rm}(4, z) = 1 \text{ (i.e. } z \text{ is the code of a successor instruction),} \\ \mathrm{transfer}(c, u_1(z), u_2(z)) & \text{if } \mathrm{rm}(4, z) = 2 \text{ (i.e. } z \text{ is the code of a transfer instruction),} \\ c & \text{otherwise.} \end{cases}$$

(10) The function $\nu(c, j, z)$ (defined in (4) above) is primitive recursive.

Proof. We have

$$\nu(c, j, z) = \begin{cases} j+1 & \text{if } \mathrm{rm}(4, z) \neq 3 \text{ (i.e. } z \text{ is the code of an arithmetic instruction),} \\ \left.\begin{array}{ll} j+1 & \text{if } (c)_{v_1(z)} \neq (c)_{v_2(z)} \\ v_3(z) & \text{if } (c)_{v_1(z)} = (c)_{v_2(z)} \end{array}\right\} & \text{if } \mathrm{rm}(4, z) = 3 \text{ (i.e. } z \text{ is the code of a jump instruction).} \end{cases}$$

From this definition by cases, we see that ν is primitive recursive.

We have now shown that the functions (1)–(4) above are primitive recursive, so the proof of the theorem is complete. □

6
Decidability, undecidability and partial decidability

In previous chapters we have noted several decidable problems, but so far we have encountered only one explicit example of undecidability: the problem 'ϕ_x is total' (theorem 5-2.1). It is of considerable interest to identify decidable and undecidable problems; the latter, particularly, indicate the limitations of computability, and hence demonstrate the theoretical limits to the power of real world computers.

In this chapter the emphasis is largely on undecidability. In § 1 we give a survey of undecidable problems arising in the theory of computability itself, and discuss some methods for establishing undecidability. Sections 2–5 are devoted to a sample of decidable and undecidable problems from other areas of mathematics: these sections will not be needed in later chapters and may be omitted. In the final section we discuss *partial decidability*, a notion closely related to decidability.

Let us recall from chapter 1 that a predicate $M(\boldsymbol{x})$ is said to be *decidable* if its characteristic function c_M, given by

$$c_M(\boldsymbol{x}) = \begin{cases} 1 & \text{if } M(\boldsymbol{x}) \text{ holds,} \\ 0 & \text{if } M(\boldsymbol{x}) \text{ does not hold,} \end{cases}$$

is computable. This is the same as saying that $M(\boldsymbol{x})$ is recursive (see chapter 3 § 2). The predicate $M(\boldsymbol{x})$ is *undecidable* if it is not decidable. In the literature all of the following phrases are used to mean that $M(\boldsymbol{x})$ is decidable.

$M(\boldsymbol{x})$ is *recursively decidable*,

$M(\boldsymbol{x})$ has *recursive decision problem*,

$M(\boldsymbol{x})$ is *solvable*,

$M(\boldsymbol{x})$ is *recursively solvable*,

$M(\boldsymbol{x})$ is *computable*.

An algorithm for computing c_M is called a *decision procedure* for $M(\boldsymbol{x})$.

1. Undecidable problems in computability

Most proofs of undecidability rest on a diagonal construction, as in the following important example.

1.1. *Theorem*

'$x \in W_x$' (or, equivalently, '$\phi_x(x)$ is defined', or '$P_x(x)\downarrow$', or '$\psi_U(x, x)$ is defined') is undecidable.

Proof. The characteristic function f of this problem is given by

$$f(x) = \begin{cases} 1 & \text{if } x \in W_x, \\ 0 & \text{if } x \notin W_x. \end{cases}$$

Suppose that f is computable; we shall obtain a contradiction. Specifically, we make a diagonal construction of a *computable* function g such that $\mathrm{Dom}(g) \neq W_x (= \mathrm{Dom}(\phi_x))$ for every x; this is obviously contradictory.

The diagonal motto tells us to ensure that $\mathrm{Dom}(g)$ differs from W_x at x; so we aim to make

$$x \in \mathrm{Dom}(g) \Leftrightarrow x \notin W_x.$$

Let us define g, then, by

$$g(x) = \begin{cases} 0 & \text{if } x \notin W_x \text{ (i.e. if } f(x) = 0), \\ \text{undefined} & \text{if } x \in W_x \text{ (i.e. if } f(x) = 1). \end{cases}$$

Since f is computable, then so is g (by Church's thesis); so we have our contradiction. (To see this in detail: since g is computable take m such that $g = \phi_m$; then $m \in W_m \Leftrightarrow m \in \mathrm{Dom}(g) \Leftrightarrow m \notin W_m$, a contradiction).

We conclude that f is *not* computable, and so the problem '$x \in W_x$' is undecidable. □

Note that this theorem does *not* say that we cannot tell for any *particular* number a whether $\phi_a(a)$ is defined. For some numbers this is quite simple; for instance, if we have written a program P that computes a total function, and $P = P_a$, then we know immediately that $\phi_a(a)$ is defined. What the theorem says is that there is no single *general* method for deciding whether $\phi_x(x)$ is defined; i.e. there is no method that works for *every* x.

An easy corollary to the above result is

1.2. *Corollary*

There is a computable function h such that the problems '$x \in \mathrm{Dom}(h)$' and '$x \in \mathrm{Ran}(h)$' are both undecidable.

Proof. Let

$$h(x) = \begin{cases} x & \text{if } x \in W_x, \\ \text{undefined} & \text{if } x \notin W_x. \end{cases}$$

Then h is computable, by Church's thesis and the computability of the universal function ψ_U (or, formally, we have that $h(x) \simeq x\ \mathbf{1}(\psi_U(x, x))$ which is computable by substitution). Clearly we have $x \in \mathrm{Dom}(h) \Leftrightarrow x \in W_x \Leftrightarrow x \in \mathrm{Ran}(h)$, so the problems '$x \in \mathrm{Dom}(h)$' and '$x \in \mathrm{Ran}(h)$' are undecidable. □

Another important undecidable problem is derived easily from theorem 1.1:

1.3. *Theorem* (the Halting problem)
The problem '$\phi_x(y)$ is defined' (or, equivalently '$P_x(y)\downarrow$' or '$y \in W_x$') is undecidable.

Proof. Arguing informally, if the problem '$\phi_x(y)$ is defined' is decidable then so is the problem '$\phi_x(x)$ is defined', which is if anything easier. But this contradicts theorem 1.1.

Giving this argument in full detail, let g be the characteristic function for '$\phi_x(y)$ is defined'; i.e.

$$g(x, y) = \begin{cases} 1 & \text{if } \phi_x(y) \text{ is defined,} \\ 0 & \text{if } \phi_x(y) \text{ is not defined.} \end{cases}$$

If g is computable, then so is the function $f(x) = g(x, x)$; but f is the characteristic function of '$x \in W_x$', and is not computable by theorem 1.1. Hence g is not computable; so '$\phi_x(y)$ is defined' is undecidable. □

Theorem 1.3 is often described as the Unsolvability of the Halting problem (for URM programs): there is no effective general method for discovering whether a given program running on a given input eventually halts. The implication of this for the theory of computer programming is obvious: there can be no perfectly general method for checking programs to see if they are free from possible infinite loops.

The undecidable problem '$x \in W_x$' of theorem 1.1 is important for several reasons. Among these is the fact that many problems can be shown to be undecidable by showing that they are at least as difficult as this one. We have already done this in a simple way in showing that the Halting problem is undecidable (theorem 1.3): this process is known as *reducing* one problem to another.

Speaking generally, consider a problem $M(x)$. Often we can show that a solution to the general problem $M(x)$ would lead to a solution to the general problem '$x \in W_x$'. Then we say that the problem '$x \in W_x$' is *reduced* to the problem $M(x)$. In other words, we can give a decision procedure for '$x \in W_x$' if only we could find one for $M(x)$. In this case, the decidability of $M(x)$ implies the decidability of '$x \in W_x$', from which we conclude immediately that $M(x)$ is undecidable.

The *s–m–n* theorem is often useful in reducing '$x \in W_x$' to other problems, as illustrated in the proof of the next result.

1.4. *Theorem*
The problem '$\phi_x = \mathbf{0}$' is undecidable.

Proof. Consider the function f defined by

$$f(x, y) = \begin{cases} 0 & \text{if } x \in W_x, \\ \text{undefined} & \text{if } x \notin W_x. \end{cases}$$

We have defined f anticipating that we shall use the *s–m–n* theorem; thus we are thinking of x as a parameter, and are concerned about the functions g_x where $g_x(y) \simeq f(x, y)$. We have actually designed f so that $g_x = \mathbf{0} \Leftrightarrow x \in W_x$.

By Church's thesis (or by substitution using $\mathbf{0}$ and ψ_U) f is computable; so there is a total computable function $k(x)$ given by the *s–m–n* theorem such that $f(x, y) \simeq \phi_{k(x)}(y)$; i.e. $\phi_{k(x)} = g_x$. Thus from the definition of f we see that

(*) $\quad x \in W_x \Leftrightarrow \phi_{k(x)} = \mathbf{0}.$

Hence, a particular question Is $x \in W_x$? can be settled by answering the question Is $\phi_{k(x)} = \mathbf{0}$? We have thus reduced the general problem '$x \in W_x$' to the general problem '$\phi_x = \mathbf{0}$'; the former is undecidable, hence so is the latter, as was to be proved.

Let us present the final part of this argument in more detail as it is the first example of its kind. Let g be the characteristic function of '$\phi_x = \mathbf{0}$'; i.e.

$$g(x) = \begin{cases} 1 & \text{if } \phi_x = \mathbf{0}, \\ 0 & \text{if } \phi_x \neq \mathbf{0}. \end{cases}$$

Suppose that g is computable; then so is the function $h(x) = g(k(x))$. But from (*) above we have

$$h(x) = \begin{cases} 1 & \text{if } \phi_{k(x)} = \mathbf{0}; \text{ i.e. } x \in W_x, \\ 0 & \text{if } \phi_{k(x)} \neq \mathbf{0}; \text{ i.e. } x \notin W_x. \end{cases}$$

So by theorem 1.1 h is *not* computable. Hence g is not computable, and the problem '$\phi_x = \mathbf{0}$' is undecidable. □

From theorem 1.4 we can see that there are inherent limitations when it comes to checking the correctness of computer programs; this theorem shows that there can be no perfectly general effective method for checking whether a program will compute the zero function. By adapting the proof of theorem 1.4 we can see that the same is true for any particular computable function (see exercise 1.8(1i) below).

The following easy corollary to theorem 1.4 shows that the question of whether two programs compute the same unary function is undecidable. Again there are obvious implications for computer programming theory.

1.5. *Corollary*

The problem '$\phi_x = \phi_y$' is undecidable.

Proof. We can easily see that this is a harder problem than the problem '$\phi_x = \mathbf{0}$'.

Let c be a number such that $\phi_c = \mathbf{0}$; if $f(x, y)$ is the characteristic function of the problem $\phi_x = \phi_y$, then the function $g(x) = f(x, c)$ is the characteristic function of '$\phi_x = \mathbf{0}$'. By theorem 1.4, g is not computable, so neither is f. Thus '$\phi_x = \phi_y$' is undecidable. □

We use the s–m–n theorem again to reduce the problem '$x \in W_x$' in the following results.

1.6. *Theorem*

Let c be any number. The following problems are undecidable.

(a) (the Input or Acceptance problem) '$c \in W_x$' (*equivalently*, '$P_x(c)\downarrow$' *or* '$c \in \mathrm{Dom}(\phi_x)$'),

(b) (the Output or Printing problem) '$c \in E_x$' (*equivalently*, '$c \in \mathrm{Ran}(\phi_x)$').

Proof. We are able to reduce '$x \in W_x$' to these problems simultaneously. Consider the function $f(x, y)$ given by

$$f(x, y) = \begin{cases} y & \text{if } x \in W_x, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

(With the s–m–n theorem in mind, we are concerned about the functions g_x where $g_x(y) \simeq f(x, y)$: we have designed f so that $c \in \mathrm{Dom}(g_x) \Leftrightarrow$

$x \in W_x \Leftrightarrow c \in \mathrm{Ran}(g_x)$.) By Church's thesis f is computable, and so the s–m–n theorem provides a total computable function k such that $f(x, y) \simeq \phi_{k(x)}(y)$. From the definition of f we see that

$$x \in W_x \Rightarrow W_{k(x)} = E_{k(x)} = \mathbb{N}, \text{ so } c \in W_{k(x)} \text{ and } c \in E_{k(x)};$$

and

$$x \notin W_x \Rightarrow W_{k(x)} = E_{k(x)} = \varnothing, \text{ so } c \notin W_{k(x)} \text{ and } c \notin E_{k(x)}.$$

Thus we have reduced the problem '$x \in W_x$' to each of the problems '$c \in W_x$' and '$c \in E_x$'.

Completing the proof of (a) in detail, we see that if g is the characteristic function of '$c \in W_x$', then

$$g(k(x)) = \begin{cases} 1 & \text{if } x \in W_x, \\ 0 & \text{if } x \notin W_x. \end{cases}$$

This function is not computable (theorem 1.1), so g cannot be computable. Hence '$c \in W_x$' is undecidable.

A detailed proof of (b) is similar. □

We conclude this section with a very general undecidability result, from which theorems 1.4 and 1.6 follow immediately. It is another use of the s–m–n theorem to reduce '$x \in W_x$'.

1.7. *Theorem* (Rice's theorem)

Suppose that $\mathscr{B} \subseteq \mathscr{C}_1$, and $\mathscr{B} \neq \varnothing, \mathscr{C}_1$. Then the problem '$\phi_x \in \mathscr{B}$' is undecidable.

Proof. From the algebra of decidability (theorem 2-4.7) we know that '$\phi_x \in \mathscr{B}$' is decidable iff '$\phi_x \in \mathscr{C}_1 \backslash \mathscr{B}$' is decidable; so we may assume without any loss of generality that the function $f_\varnothing$ that is nowhere defined does not belong to $\mathscr{B}$ (if not, prove the result for $\mathscr{C}_1 \backslash \mathscr{B}$).

Choose a function $g \in \mathscr{B}$. Consider the function $f(x, y)$ defined by

$$f(x, y) \simeq \begin{cases} g(y) & \text{if } x \in W_x, \\ \text{undefined} & \text{if } x \notin W_x. \end{cases}$$

The s–m–n theorem provides a total computable function $k(x)$ such that $f(x, y) \simeq \phi_{k(x)}(y)$. Thus we see that

$$x \in W_x \Rightarrow \phi_{k(x)} = g, \text{ i.e. } \phi_{k(x)} \in \mathscr{B};$$

$$x \notin W_x \Rightarrow \phi_{k(x)} = f_\varnothing, \text{ i.e. } \phi_{k(x)} \notin \mathscr{B}.$$

So we have reduced the problem '$x \in W_x$' to the problem '$\phi_x \in \mathscr{B}$' using the computable function k. In the standard way we conclude that '$\phi_x \in \mathscr{B}$' is undecidable. □

Theorem 1.4, for example, is obtained immediately from Rice's theorem by taking $\mathscr{B} = \{\mathbf{0}\}$, and theorem 1.6($a$) by taking $\mathscr{B} = \{g \in \mathscr{C}_1 : c \in \mathrm{Dom}(g)\}$. Rice's theorem may be similarly applied in several of the exercises below.

1.8. *Exercises*

1. Show that the following problems are undecidable.
   (a) '$x \in E_x$' (*Hint.* Either use a direct diagonal construction, or reduce '$x \in W_x$' to this problem using the s–m–n theorem.),
   (b) '$W_x = W_y$' (*Hint.* Reduce 'ϕ_x is total' to this problem.),
   (c) '$\phi_x(x) = 0$',
   (d) '$\phi_x(y) = 0$',
   (e) '$x \in E_y$',
   (f) 'ϕ_x is total and constant',
   (g) '$W_x = \varnothing$',
   (h) 'E_x is infinite'.
   (i) '$\phi_x = g$', where g is any fixed computable function.
2. Show that there is no total computable function $f(x, y)$ with the following property: if $P_x(y)$ stops, then it does so in $f(x, y)$ or fewer steps. (*Hint.* Show that if such a function exists, then the Halting problem is decidable.)

Decidability and undecidability in other areas of mathematics In many areas of mathematics there arise general problems for which the informal idea of decidability is meaningful. Generally such problems involve finite objects from a particular field of study. The idea of decidability of some property involving these objects can always be made precise using a suitable coding by natural numbers.

Much research has been directed towards identifying both decidable and undecidable problems in a variety of mathematical situations: in the next sections we give a small sample of the results that have been obtained.

2. The word problem for groups[1]

Suppose that G is a group with identity element 1, and that G is generated by a set of elements $S = \{g_1, g_2, g_3, \ldots\} \subseteq G$. A *word* on S is

[1] The reader with no knowledge of group theory should omit this section.

any expression such as $g_2^{-1} g_3^6 g_1 g_2^5 g_8$ involving the elements of S and the group operations. Each word represents an element of G, and to say that G is *generated* by S means that every element of G is represented by some word on S.

The *word problem* for G (relative to S) is the problem of deciding for which words w on S is it the case that $w = 1$.

There are many groups with decidable word problem: for example any finite group (with S finite, of course). For many years mathematicians searched for an example of a *finitely presented*[2] group with *un*decidable word problem. Eventually it was shown by Novikov in 1955 and Boone in 1957 that such groups do exist. Proofs of the Novikov–Boone Theorem are beyond the scope of this survey: the reader is referred to expositions in Rotman [1965] or Manin [1977].

Group theory, and modern algebra in general, abounds with interesting decidable and undecidable problems; a great many of them involve properties of words or generators akin to the basic word problem for groups.

3. Diophantine equations

Suppose that $p(x_1, x_2, \ldots, x_n)$ is a polynomial in the variables $x_1, \ldots, x_n$, with integer coefficients. Then the equation

$$p(x_1, x_2, \ldots, x_n) = 0$$

for which *integer* solutions are sought is called a *diophantine equation.* Diophantine equations do not always have solutions: for instance the equation $x^2 - 2 = 0$.

Hilbert's tenth problem, posed in 1900, asks whether there is an effective procedure that will determine whether any given diophantine equation has a solution. It was shown in 1970 by Y. Matiyasevich that there is no such procedure; his proof was the culmination of earlier work by M. Davis, J. Robinson and H. Putnam.

Actually Matiyasevich established rather more than the unsolvability of Hilbert's tenth problem; the full Matiyasevich theorem and its application to Hilbert's tenth problem are discussed in § 6. For complete details consult Davis [1973] or Manin [1977], or Bell & Machover [1977].

[2] A group G is *finitely presented* if there is a finite set of generators S and a finite set B of *relations* of the form $w = 1$ (where w is a word on S) such that (i) all relations in B are true in G, and (ii) all other relations holding in G can be deduced from those in B by using the group axioms alone.

4. Sturm's algorithm

To redress the emphasis on undecidability in the previous two sections, we now mention a theorem of Sturm that gives us positive results for computability and decidability in connection with the zeros of polynomials.

4.1. *Sturm's theorem*

Let $p(x)$ be a real polynomial, and let $p_0, p_1, \ldots, p_r$ be the sequence of real polynomials given by

(a) $p_0 = 0$,

(b) $p_1 = p'$ (the derivative of p),

(c) for $0 < i < r$, there is a polynomial q_i such that $p_{i-1} = p_i q_i - p_{i+1}$ with $p_{i+1} \neq 0$ and $\text{degree}(p_{i+1}) < \text{degree}(p_i)$ (so that q_i and $-p_{i+1}$ are the quotient and remainder respectively when p_{i-1} is divided by p_i),

(d) $p_{r-1} = p_r q_r$.

For any real number c denote by $\delta(c)$ the number of sign changes in the sequence $p_0(c), \ldots, p_r(c)$ (ignoring zeros).

Suppose that a, b are real numbers that are not zeros of $p(x)$, and $a < b$. Then the number of zeros of $p(x)$ in the interval $[a, b]$ is $\delta(a) - \delta(b)$, (each zero being counted once only).

This is not the place to give a proof of Sturm's theorem, which the reader may find clearly expounded in Cohn [1977] or Van der Waerden [1949]. From our point of view, Sturm's theorem is interesting because of the algorithm it embodies. It gives us positive results about the *computability* of the number of zeros of a polynomial, and the *decidability* of statements about zeros of polynomials.

To frame such results, we must restrict attention to polynomials over the rational numbers, denoted by $\mathbb{Q}$, so that the objects we are dealing with are finite. Thus we are thinking in terms of computability over the domain $\mathbb{Q}$ (which can be defined in terms of computability on $\mathbb{N}$ by the usual coding device); note that a polynomial $p(x)$ with coefficients in $\mathbb{Q}$ is essentially a sequence of rational numbers.

A sample of the results that follow from Sturm's theorem is the following.

4.2. *Theorem*

(a) There is an effective procedure for calculating the number of real zeros of a polynomial over $\mathbb{Q}$;

(b) The predicate 'p has a zero in $[a, b]$' is decidable, where p denotes a polynomial over $\mathbb{Q}$ and $a, b \in \mathbb{Q}$.

Proof. Given any polynomial p, the polynomials $p_0, p_1, \ldots, p_r$ defined in Sturm's theorem may be found effectively by using the standard rules for differentiation and the division algorithm for polynomials.

For (a), it is a routine matter to find for any polynomial $p(x)$ a rational number $M > 0$ such that all the zeros of p lie in the interval $]-M, M[$. In fact, if $p(x) = a_0 + a_1 x + \ldots + a_n x^n$, the number

$$M = 1 + \frac{1}{|a_n|}(|a_0| + \ldots + |a_{n-1}|)$$

suffices. Then by Sturm's theorem the number of zeros of p is $\delta(-M) - \delta(M)$ which may be calculated effectively.

For (b), suppose that we are given a polynomial p and rationals a, b. To decide whether p has a zero in $[a, b]$, first calculate $p(a)$ and $p(b)$; if neither of these is zero, calculate $\delta(a) - \delta(b)$ and apply Sturm's theorem. □

Of course, Sturm's theorem can be used to show that many other questions about polynomials over $\mathbb{Q}$ are computable or decidable.

4.3. *Exercise*

Show that there is an effective procedure, given a polynomial p and rational numbers a, b, for finding the number of zeros of p in $[a, b]$. (Remember that a or b may be zeros of p.)

5. Mathematical logic

Early investigations into the idea of effective computability were very much linked with the development of mathematical logic, because decidability was regarded as a basic question about any formalisation of mathematics. We shall describe some of the results that have been obtained in this area, in general terms that do not assume any acquaintance with mathematical logic. (The reader interested to learn the basics of this subject may consult one of the many introductory texts, such as Margaris [1966].)

The simplest logical system reflecting something of mathematical reasoning is the *propositional calculus*. In this calculus compound statements are formed from basic propositions using symbols for the logical connectives 'not', 'and'; 'or', and 'implies'. It is quite easy, once the

propositional calculus has been carefully defined, to see that it is *decidable*. By this we mean that there is an effective procedure for deciding whether a statement σ of the calculus is (*universally*) *valid*; i.e. true in all possible situations. The method of truth tables gives an algorithm for this that will be familiar to many readers.

A logical system that has greater expressive power than the propositional calculus is the (*first-order*) *predicate calculus*: using the language of this calculus it is possible to formalise a great deal of mathematics. The basic statements are formed from symbols representing individual objects (or elements) and predicates and functions of them. The compound statements are formed using the logical symbols of the propositional calculus together with $\forall$ and $\exists$.

There is a precise notion of a *proof* of a statement of the predicate calculus, such that a statement is provable if and only if it is valid.[3] In 1936 Church showed that provability (and hence validity) in the predicate calculus is *undecidable*, unlike the simpler propositional calculus. (This result was regarded by Hilbert as the most fundamental undecidability result for the whole of mathematics.)

We can use the URM to give an easy proof of the undecidability of validity, although this calls upon a certain familiarity with the predicate calculus. We advise the reader who does not have a rudimentary knowledge of predicate logic to omit the proof that we now sketch.

5.1. *Theorem*

Validity in the first-order predicate calculus is undecidable.

Proof. (Not advised for strangers to the predicate calculus.)

Let P be a program in standard form having instructions $I_1, \ldots, I_s$ and let $u = \rho(P)$ (as defined in chapter 2 § 2). We use the following symbols of the predicate calculus.

$\mathsf{0}$ a symbol for an individual,

$'$ a symbol for a unary function (whose value at x is x'),

R a symbol for a $(u+1)$-ary relation,

$\mathsf{x}_1, \mathsf{x}_2, \ldots \mathsf{x}_u, \mathsf{y}$ symbols for variable individuals.

The interpretation we have in mind is that $\mathsf{0}$ represents the number 0, $'$ represents the function $x+1$, and R represents the possible states of a computation under P. Thus if we write $\mathsf{1}$ for $\mathsf{0}'$, $\mathsf{2}$ for $\mathsf{0}''$, etc. the statement

$$\mathsf{R}(\mathsf{r}_1, \ldots, \mathsf{r}_u, \mathsf{k})$$

[3] This is described by saying that the notion of provability is *complete*, and is the content of Gödel's completeness theorem.

where $r_1, \ldots r_u, k \in \mathbb{N}$ means that the state

r_1	r_2	...	r_u	0	0	...

; next instruction I_k

occurs in the computation.

Now for each instruction I_i we can write down a statement τ_i of the predicate calculus that describes the effect of I_i on states, using the symbol $\wedge$ for 'and' and $\rightarrow$ for 'implies':

(a) if $I_i = \mathrm{Z}(n)$ let τ_i be the statement

$$\forall \mathsf{x}_1 \ldots \forall \mathsf{x}_u\colon \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_n, \ldots, \mathsf{x}_u, \mathsf{i}) \rightarrow \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{0}, \ldots, \mathsf{x}_u, \mathsf{i}').$$

(b) If $I_i = \mathrm{S}(n)$ let τ_i be the statement

$$\forall \mathsf{x}_1 \ldots \forall \mathsf{x}_u\colon \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_n, \ldots, \mathsf{x}_u, \mathsf{i}) \rightarrow \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_n', \ldots, \mathsf{x}_u, \mathsf{i}').$$

(c) If $I_i = \mathrm{T}(m, n)$ let τ_i be the statement

$$\forall \mathsf{x}_1 \ldots \forall \mathsf{x}_u\colon \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_n, \ldots, \mathsf{x}_u, \mathsf{i}) \rightarrow \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_m, \ldots, \mathsf{x}_u, \mathsf{i}').$$

(d) If $I_i = \mathrm{J}(m, n, q)$ let τ_i be the statement

$$\forall \mathsf{x}_1 \ldots \forall \mathsf{x}_u\colon \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_u, \mathsf{i}) \rightarrow ((\mathsf{x}_m = \mathsf{x}_n \rightarrow \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_u, \mathsf{q})) \wedge (\mathsf{x}_m \neq \mathsf{x}_n \rightarrow \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_u, \mathsf{i}'))).$$

Now for any $a \in \mathbb{N}$ let σ_a be the statement

$$(\tau_0 \wedge \tau_1 \wedge \ldots \wedge \tau_s \wedge \mathsf{R}(\mathsf{a}, \mathsf{0}, \ldots, \mathsf{0}, \mathsf{1})) \rightarrow \exists \mathsf{x}_1 \ldots \exists \mathsf{x}_u\, \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_u, \mathsf{s}+\mathsf{1}),$$

where τ_0 is the statement $\forall \mathsf{x} \forall \mathsf{y}((\mathsf{x}' = \mathsf{y}' \rightarrow \mathsf{x} = \mathsf{y}) \wedge \mathsf{x}' \neq \mathsf{0})$. (This ensures that in any interpretation, if $m, n \in \mathbb{N}$ and $\mathsf{m} = \mathsf{n}$ then $m = n$.)

The statement $\mathsf{R}(\mathsf{a}, \mathsf{0}, \ldots, \mathsf{0}, \mathsf{1})$ corresponds to a starting state

a	0	0	...

; next instruction I_1,

and any statement $\mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_u, \mathsf{s}+\mathsf{1})$ corresponds to a halting state (since there is no instruction I_{s+1}). Thus we shall see that

(*) $\quad P(a)\!\downarrow \Leftrightarrow \sigma_a$ is valid.

Suppose first that $P(a)\!\downarrow$, and that we have a structure in which $\tau_0, \ldots, \tau_s$ and $\mathsf{R}(\mathsf{a}, \mathsf{0}, \ldots, \mathsf{0}, \mathsf{1})$ hold. Using the statements $\tau_0, \ldots, \tau_s$ we find that each of the statements $\mathsf{R}(\mathsf{r}_1, \ldots, \mathsf{r}_u, \mathsf{k})$ corresponding to the successive states in the computation also holds. Eventually we find that a halting statement $\mathsf{R}(\mathsf{b}_1, \ldots, \mathsf{b}_u, \mathsf{s}+\mathsf{1})$ holds, for some $b_1, \ldots, b_u \in \mathbb{N}$, and hence $\exists \mathsf{x}_1 \ldots \exists \mathsf{x}_u\, \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_u, \mathsf{s}+\mathsf{1})$ holds. Thus σ_a is valid.

Conversely, if σ_a is valid, it holds in particular in the structure $\mathbb{N}$ with the predicate symbol R interpreted by the predicate R_a where

$R_a(a_1, \ldots, a_u, k) \equiv$ At some stage in the computation $P(a)$ the registers contain $a_1, a_2, \ldots, a_u, 0, 0, \ldots$ and the next instruction is I_k.

Then $\tau_0, \ldots, \tau_s$ and $\mathsf{R}(\mathsf{a}, 0, \ldots, 0, 1)$ all hold in this structure, hence so does $\exists \mathsf{x}_1 \ldots \exists \mathsf{x}_u\, \mathsf{R}(\mathsf{x}_1, \ldots, \mathsf{x}_u, \mathsf{s}+1)$. Therefore $P(a)\downarrow$.

If we take P to be a program that computes the function $\psi_U(x, x)$, the equivalence (*) gives a reduction of the problem '$x \in W_x$' to the problem 'σ is valid'. Hence the latter is undecidable. □

The field of mathematical logic abounds with decidability and undecidability results. A common type of problem that arises is whether a statement is true in all mathematical structures of a certain kind. It has been shown, for example, that the problem

'σ is a statement that is true in all groups'

is undecidable (here σ is a statement of the first-order predicate language appropriate to groups), whereas the problem

'σ is a statement that is true in all abelian groups'

is decidable. (We say that the first-order theory of groups is *undecidable* whereas the first-order theory of abelian groups is *decidable*.) It was shown by Tarski [1951] that the problem

'σ is true in the field of real numbers'

is decidable. On the other hand, many problems connected with the formalisation of ordinary arithmetic on the natural numbers are undecidable, as we shall see in chapter 8.

For further examples and proofs of decidability and undecidability results in logic the reader should consult books such as Tarski, Mostowski & Robinson [1953], or Boolos & Jeffrey [1974].

6. Partially decidable predicates

Although the predicate '$x \in W_x$' has non-computable characteristic function, the following function connected with this problem *is* computable:

$$f(x) = \begin{cases} 1 & \text{if } x \in W_x, \\ \text{undefined} & \text{if } x \notin W_x. \end{cases}$$

If we continue to think of 1 as a code for Yes, then any algorithm for f is a procedure that gives answer Yes when $x \in W_x$, but goes on for ever when $x \in W_x$ does not hold. Such a procedure is called a *partial decision procedure* for the problem '$x \in W_x$', and we say that this problem or predicate is *partially decidable*.

Many undecidable predicates turn out to be partially decidable: let us formulate the general definition.

6.1. *Definition*

A predicate $M(x)$ of natural numbers is *partially decidable* if the function f given by

$$f(x) = \begin{cases} 1 & \text{if } M(x) \text{ holds}, \\ \text{undefined} & \text{if } M(x) \text{ does not hold}, \end{cases}$$

is computable. (This function is called the *partial characteristic function* for M.) If M is partially decidable, any algorithm for computing f is called a *partial decision procedure* for M.

Note. In the literature the terms *partially solvable*, *semi-computable*, and *recursively enumerable*[4] are used with the same meaning as partially decidable.

6.2. *Examples*

1. The Halting problem (theorem 1.3) is partially decidable, since its partial characteristic function

   $$f(x, y) = \begin{cases} 1 & \text{if } P_x(y)\downarrow, \\ \text{undefined} & \text{otherwise}, \end{cases}$$

   is computable, by Church's thesis (or by observing that $f(x, y) \simeq \mathbf{1}(\psi_U(x, y))$).
2. Any decidable predicate is partially decidable: simply arrange for the decision procedure to enter a loop whenever it gives output 0.
3. For any computable function $g(x)$ the problem '$x \in \mathrm{Dom}(g)$' is partially decidable, since it has the computable partial characteristic function $\mathbf{1}(g(x))$. (Cf. corollary 1.2.)
4. The problem '$x \notin W_x$' is *not* partially decidable: for if f is its partial characteristic function, then

   $$x \in \mathrm{Dom}(f) \Leftrightarrow x \notin W_x.$$

   Thus $\mathrm{Dom}(f)$ differs from the domain of every unary computable function: hence f is not computable.

[4] The reason for the use of this term will be explained in the next chapter.

We proceed to establish some of the important characteristics of partially decidable predicates. First we have the alternative characterisation that is given essentially in example 6.2(3) above.

6.3. *Theorem*

A predicate $M(\boldsymbol{x})$ is partially decidable if and only if there is a computable function $g(\boldsymbol{x})$ such that

$$M(\boldsymbol{x}) \quad \textit{iff } \boldsymbol{x} \in \mathrm{Dom}(g).$$

Proof. If $M(\boldsymbol{x})$ is partially decidable with computable partial characteristic function $f(\boldsymbol{x})$, then from the definition we have $M(\boldsymbol{x})$ iff $\boldsymbol{x} \in \mathrm{Dom}(f)$. The converse is given by example 6.2(3) above. □

The following characterisation of partially decidable predicates shows how they are related to decidable predicates.

6.4. *Theorem*

A predicate $M(\boldsymbol{x})$ is partially decidable if and only if there is a decidable predicate $R(\boldsymbol{x}, y)$ such that

$$M(\boldsymbol{x}) \textit{ iff } \exists y R(\boldsymbol{x}, y).$$

Proof. Suppose that $R(\boldsymbol{x}, y)$ is a decidable predicate and that $M(\boldsymbol{x})$ iff $\exists y R(\boldsymbol{x}, y)$. By corollary 2-5.3 the function $g(\boldsymbol{x}) \simeq \mu y R(\boldsymbol{x}, y)$ is computable; clearly

$$M(\boldsymbol{x}) \Leftrightarrow \boldsymbol{x} \in \mathrm{Dom}(g),$$

so $M(\boldsymbol{x})$ is partially decidable by theorem 6.3.

For the converse, suppose that $M(\boldsymbol{x})$ is partially decidable, with partial decision procedure given by a program P. Define a predicate $R(\boldsymbol{x}, y)$ by

$$R(\boldsymbol{x}, y) \equiv P(\boldsymbol{x})\downarrow \text{ in } y \text{ steps}.$$

By corollary 5-1.3, $R(\boldsymbol{x}, y)$ is decidable. Moreover,

$$\begin{aligned} M(\boldsymbol{x}) &\Leftrightarrow P(\boldsymbol{x})\downarrow \\ &\Leftrightarrow \exists y R(\boldsymbol{x}, y) \end{aligned}$$

as required. □

Note. From the appendix to chapter 5 it follows that the predicate R in this characterisation may be taken to be primitive recursive (see the note 1 following corollary 5-1.4).

The characterisation given by theorem 6.4 indicates an important way to think of partially decidable predicates. It shows that partial decision procedures can always be cast in the form of an unbounded search for a

number y having some decidable property $R(\boldsymbol{x}, y)$. This search is most naturally carried out by examining successively $y = 0, 1, 2, \ldots$ to find such a y. The search halts if and when y is found such that $R(\boldsymbol{x}, y)$ holds; otherwise the search goes on for ever.

We can use theorem 6.4 to establish some further properties of partially decidable predicates, that aid us in their recognition.

6.5. *Theorem*

If $M(\boldsymbol{x}, y)$ is partially decidable, then so is the predicate $\exists y M(\boldsymbol{x}, y)$.

Proof. Take a decidable predicate $R(\boldsymbol{x}, y, z)$ such that $M(\boldsymbol{x}, y)$ iff $\exists z R(\boldsymbol{x}, y, z)$. Then we have

$$\exists y M(\boldsymbol{x}, y) \Leftrightarrow \exists y \exists z R(\boldsymbol{x}, y, z).$$

We can use the standard technique of coding the pair of numbers y, z by the single number $u = 2^y 3^z$; then the search for a pair y, z such that $R(\boldsymbol{x}, y, z)$ reduces to the search for a single number u such that $R(\boldsymbol{x}, (u)_1, (u)_2)$, i.e.

$$\exists y M(\boldsymbol{x}, y) \Leftrightarrow \exists u R(\boldsymbol{x}, (u)_1, (u)_2).$$

The predicate $S(\boldsymbol{x}, u) \equiv R(\boldsymbol{x}, (u)_1, (u)_2)$ is decidable (by substitution) and so by theorem 6.4 $\exists y M(\boldsymbol{x}, y)$ is partially decidable. □

Theorem 6.5 is described by saying that partially decidable predicates are *closed under existential quantification.* Its repeated application gives

6.6. *Corollary*

If $M(\boldsymbol{x}, \boldsymbol{y})$ is partially decidable, where $\boldsymbol{y} = (y_1, \ldots, y_m)$, then so is the predicate $\exists y_1 \ldots \exists y_m M(\boldsymbol{x}, y_1, \ldots, y_m)$.

Let us now consider some applications of the above results.

6.7. *Examples*

1. The following predicates are partially decidable.
   (*a*) $x \in E_y^{(n)}$ (n fixed). (The Printing problem: cf. theorem 1.6.)
   (*b*) $W_x \neq \varnothing$ (Cf. exercise 1.8(1*g*).)

   Proofs

   (*a*) $x \in E_y^{(n)} \Leftrightarrow \exists z_1 \ldots \exists z_n \exists t (P_y(z_1, \ldots, z_n)\downarrow x \text{ in } t \text{ steps})$. The predicate in the brackets on the right is decidable; apply corollary 6.6.

   (*b*) $W_x \neq \varnothing \Leftrightarrow \exists y \exists t (P_x(y)\downarrow \text{ in } t \text{ steps})$; again the predicate in brackets is decidable, so corollary 6.6 applies.

2. Provability in the predicate calculus is partially decidable (this is for those who have read § 5).

Proof. We proceed informally; in the predicate calculus a *proof* is defined as a finite object (usually a sequence of statements) in such a way that the predicate

$\Pr(d, \sigma) \equiv$ 'd is a proof of the statement σ'

is decidable. Then we have

σ is provable $\Leftrightarrow \exists d \Pr(d, \sigma)$,

hence 'σ is provable' is partially decidable.

6.8. *Diophantine predicates* (cf. § 3)

Suppose that $p(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is a polynomial with integer coefficients. Then the predicate $M(\boldsymbol{x})$ given by

$$M(\boldsymbol{x}) = \exists y_1 \ldots \exists y_m (p(\boldsymbol{x}, y_1, \ldots, y_m) = 0)$$

is called a *diophantine* predicate, because of its obvious connection with diophantine equations. (The quantifiers $\exists y_1, \ldots, \exists y_m$ are taken as ranging over $\mathbb{N}$.)

Example The predicate 'x is a perfect square' is diophantine, since it is equivalent to $\exists y(x - y^2 = 0)$.

From corollary 6.6 we have immediately

6.9. *Theorem*

Diophantine predicates are partially decidable.

Proof. The predicate $p(\boldsymbol{x}, \boldsymbol{y}) = 0$ is decidable; apply corollary 6.6. □

Clearly, diophantine predicates are partially decidable predicates that can be cast in a relatively simple form, and for a long time it was not known whether any undecidable diophantine predicates existed. This question is closely connected with Hilbert's tenth problem (§ 3), as we shall see. It was a most remarkable achievement, therefore, when Matiyasevich proved in 1970:

6.10. *Theorem*

Every partially decidable predicate is diophantine.

The proof of this result by Matiyasevich rested heavily on earlier work of Davis, Robinson and Putnam, and is far too long to present here. Full

proofs are given in Davis [1973], Bell & Machover [1977] and Manin [1977]. The major part of the proof consists in showing that diophantine predicates are closed under bounded universal quantification; i.e. if $M(\boldsymbol{x}, y)$ is diophantine then so is the predicate $\forall z < y M(\boldsymbol{x}, z)$. (It is an easy exercise to show that partially decidable predicates are closed under bounded universal quantification; see exercise 6.14(5) below.)

We can see how a negative solution to Hilbert's tenth problem is easily derived from Matiyasevich's theorem. First note that if the problem posed by Hilbert is decidable, then so is the problem of deciding for a general polynomial equation $p(x_1, \ldots, x_n) = 0$ (with integer coefficients) whether it has a solution in the natural numbers: this is because any natural number is expressible as the sum of four squares, so we simply look for integer solutions to

$$p(s_1^2 + t_1^2 + u_1^2 + v_1^2, \ldots, s_n^2 + t_n^2 + u_n^2 + v_n^2) = 0.$$

Now take a polynomial $p(x, y_1, \ldots, y_m)$ such that

$$x \in W_x \Leftrightarrow \exists y_1 \ldots \exists y_m (p(x, y_1, \ldots, y_m) = 0)$$

(this is possible by Matiyasevich's theorem). Then a decision procedure for Hilbert's problem would give the following decision procedure for '$x \in W_x$': to test whether $a \in W_a$ see whether the polynomial $q(y_1, \ldots, y_m) = p(a, y_1, \ldots, y_m)$ has a solution in $\mathbb{N}$. So '$x \in W_x$' has been reduced to Hilbert's problem; hence the latter is undecidable.

We shall mention another (surprising) consequence of Matiyasevich's theorem in the next chapter.

We conclude this chapter with two important results, linking partially decidable predicates with decidable predicates (theorem 6.11) and computable functions (theorem 6.13).

6.11. *Theorem*

A predicate $M(\boldsymbol{x})$ is decidable if and only if both $M(\boldsymbol{x})$ and 'not $M(\boldsymbol{x})$' are partially decidable.

Proof. If $M(\boldsymbol{x})$ is decidable, so is 'not $M(\boldsymbol{x})$', so both are partially decidable.

Conversely, suppose that partial decision procedures for $M(\boldsymbol{x})$ and 'not $M(\boldsymbol{x})$' are given by programs F, G. Then

$$F(\boldsymbol{x})\downarrow \Leftrightarrow M(\boldsymbol{x}) \text{ holds}$$

and

$$G(\boldsymbol{x})\downarrow \Leftrightarrow \text{'not } M(\boldsymbol{x})\text{' holds.}$$

Moreover, for any $\boldsymbol{x}$, *either* $F(\boldsymbol{x})\downarrow$ *or* $G(\boldsymbol{x})\downarrow$ but not *both*. Thus the following is an algorithm for deciding $M(\boldsymbol{x})$. Given $\boldsymbol{x}$, run the

computations $F(x)$ and $G(x)$ simultaneously (or carry out alternately one step in each computation), and go on until one of them stops. If it is $F(x)$ that stops, then conclude that $M(x)$ holds; if it is $G(x)$ that stops, then $M(x)$ does not hold. □

This theorem gives an alternative proof that the predicate '$x \notin W_x$' is not partially decidable. Similarly we have

6.12. *Corollary*

The predicate '$P_x(y)\uparrow$' (the Divergence problem: *equivalently*, '$y \notin W_x$', *or* '$\phi_x(y)$ *is undefined*') *is not partially decidable.*

Proof. If this problem were partially decidable, then by theorem 6.11 and example 6.2(1) the Halting problem $P_x(y)\downarrow$ would be decidable. □

The final result of this chapter gives a useful way to show that a function is computable.

6.13. *Theorem*

Let $f(x)$ be a partial function. Then f is computable if and only if the predicate

$$'f(x) \simeq y'$$

is partially decidable.

Proof. If f is computable by a program P, then we have

$$f(x) \simeq y \Leftrightarrow \exists t(P(x)\downarrow y \text{ in } t \text{ steps}).$$

The predicate on the right is partially decidable by theorem 6.4 and corollary 5-1.3.

Conversely, suppose that the predicate '$f(x) \simeq y$' is partially decidable. Let $R(x, y, t)$ be a decidable predicate such that $f(x) \simeq y \Leftrightarrow \exists t R(x, y, t)$. Then we have the following algorithm for computing $f(x)$.

Search for a pair of numbers y, t such that $R(x, y, t)$ holds; if and when such a pair is found, then $f(x) \simeq y$.

Hence f is computable. (A formal proof of the computability of f could be given by the standard technique of coding a pair y, t by the single number $z = 2^y 3^t$. See exercise 6.14(8) below.) □

Further properties of partially decidable predicates are given in the exercises below (see in particular exercises 6.14(4, 5, 9)).

In the next chapter we will be studying unary partially decidable predicates in greater detail, in the guise of *recursively enumerable sets*. We shall see in particular why partially decidable predicates are often described as recursively enumerable predicates.

6.14. *Exercises*

1. Show that the following predicates are partially decidable:
   (*a*) '$E_x^{(n)} \neq \varnothing$' (n fixed),
   (*b*) '$\phi_x(y)$ is a perfect square',
   (*c*) 'n is a Fermat number'. (We say that n is a *Fermat number* if there are numbers $x, y, z > 0$ such that $x^n + y^n = z^n$.)
   (*d*) 'There is a run of exactly x consecutive 7s in the decimal expansion of π'.
2. (For those knowing some group theory) Show that the word problem for any finitely presented group is partially decidable.
3. A finite set S of 3×3 matrices is said to be *mortal* if there is a finite product of members of S that equals the zero matrix. Show that the predicate 'S is mortal' is partially decidable. (It has been shown that this problem is *not* decidable; see Paterson [1970].)
4. Suppose that $M(x)$ and $N(x)$ are partially decidable; prove that the predicates '$M(x)$ and $N(x)$', '$M(x)$ or $N(x)$' are partially decidable. Show that the predicate 'not $M(x)$' is not necessarily partially decidable.
5. Suppose that $M(x, y)$ is partially decidable. Show that
   (*a*) '$\exists y < z\, M(x, y)$' is partially decidable,
   (*b*) '$\forall y < z\, M(x, y)$' is partially decidable.
   (*Hint.* If $f(x, y)$ is the partial characteristic function of M, consider the function $\prod_{y<z} f(x, y)$.)
   (*c*) '$\forall y M(x, y)$' is not necessarily partially decidable.
6. Show that the following predicates are diophantine.
   (*a*) 'x is even',
   (*b*) 'x divides y'.
7. (This exercise shows how the technique of reducibility (§ 1) may be used to show that a predicate is not partially decidable.)
   (a) Suppose that $M(x)$ is a predicate and k a total computable function such that $x \in W_x$ iff $M(k(x))$ does not hold. Prove that $M(x)$ is not partially decidable.
   (*b*) Prove that 'ϕ_x is not total' is not partially decidable.
   (*Hint.* Consider the function k in the proof of theorem 1.6.)
   (*c*) By considering the function

$$f(x, y) \simeq \begin{cases} 1 & \text{if } P_x(x) \text{ does not converge in } y \text{ or} \\ & \text{fewer steps,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

show that 'ϕ_x is total' is not partially decidable. (*Hint.* Use the s–m–n theorem and (a).)

8. Give a formal proof of the second half of Theorem 6.13; i.e. if '$f(\boldsymbol{x}) \simeq y$' is partially decidable, then f is computable.
9. Suppose that $M(x_1, \ldots, x_n)$ is partially decidable and $g_1, \ldots, g_n$ are computable partial functions. Show that the predicate $N(\boldsymbol{y})$ given by

   $N(\boldsymbol{y}) \equiv M(g_1(\boldsymbol{y}), \ldots, g_n(\boldsymbol{y}))$

   is partially decidable. (We take this to mean that $N(\boldsymbol{y})$ does not hold if any one of $g_1(\boldsymbol{y}), \ldots, g_n(\boldsymbol{y})$ is undefined.)

7
Recursive and recursively enumerable sets

The sets mentioned in the title to this chapter are subsets of $\mathbb{N}$ corresponding to decidable and partially decidable predicates. We discuss recursive sets briefly in § 1. The major part of this chapter is devoted to the study of recursively enumerable sets, beginning in § 2; many of the basic properties of these sets are derived directly from the results about partially decidable predicates in the previous chapter. The central new result in § 2 is the characterisation of recursively enumerable sets that gives them their name: they are sets that can be enumerated by a recursive (or computable) function.

In §§ 3 and 4 we introduce *creative* sets and *simple* sets: these are special kinds of recursively enumerable sets that are in marked contrast to each other; they give a hint of the great variety existing within this class of sets.

1. Recursive sets

There is a close connection between unary predicates of natural numbers and subsets of $\mathbb{N}$: corresponding to any predicate $M(x)$ we have the set $\{x : M(x) \text{ holds}\}$, called the *extent* of M (which could, of course, be $\varnothing$); while to a set $A \subseteq \mathbb{N}$ there corresponds the predicate '$x \in A$'.[1] The name *recursive* is given to sets corresponding in this way to predicates that are *decidable*.

1.1. *Definition*

Let A be a subset of $\mathbb{N}$. The *characteristic function* of A is the function c_A given by

$$c_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{if } x \notin A. \end{cases}$$

[1] As mentioned in a footnote to § 3 of the Prologue, predicates are often identified with their extent: that view would not be inconsistent with our exposition.