From:

N. J. Cutland

"Computability

An Introduction to

recursive function theory"

Cambridge Univ. Press

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. 1*a*. 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. 1*a*.

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 $I_k = \mathrm{J}(m, n, q)$ 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 $I_k = \mathrm{J}(m, n, q)$, $r_m = r_n$ and $q > s$,
(iii) if $I_k = \mathrm{J}(m, n, q)$, $r_m \neq r_n$ 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.

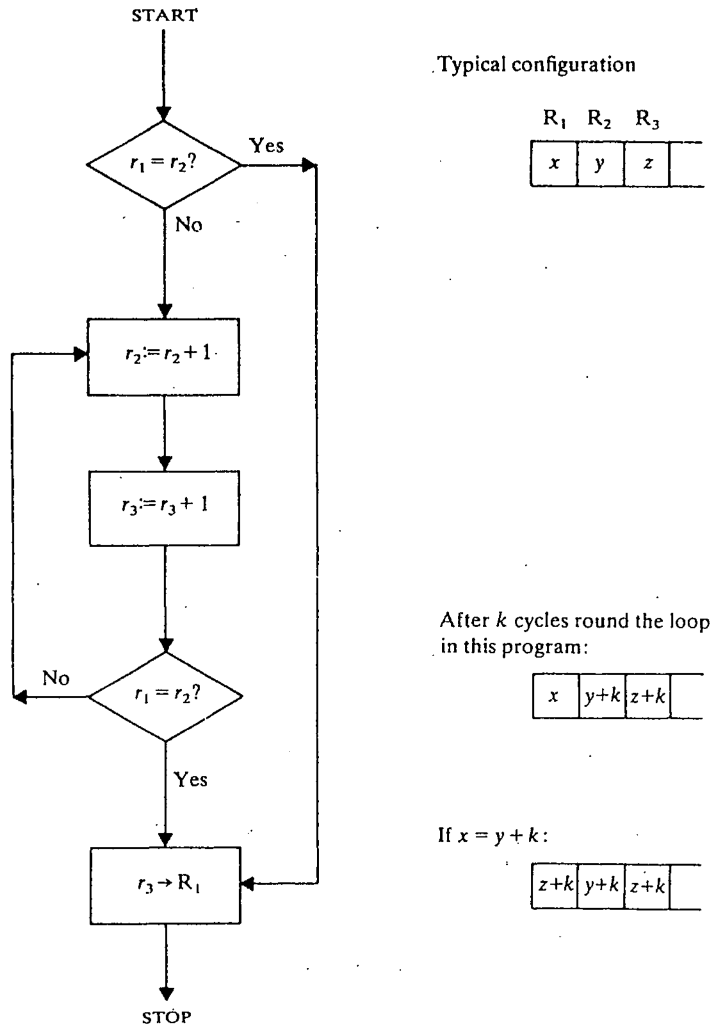
2.2. *Exercise*
Carry out the computation under the program of example 2.1 with initial configuration 8, 4, 2, 0, 0, . . .

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.

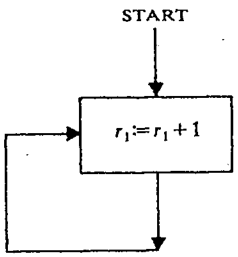
Fig. 1*b*. Flow diagram for the program of example 2.1.



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	. . .

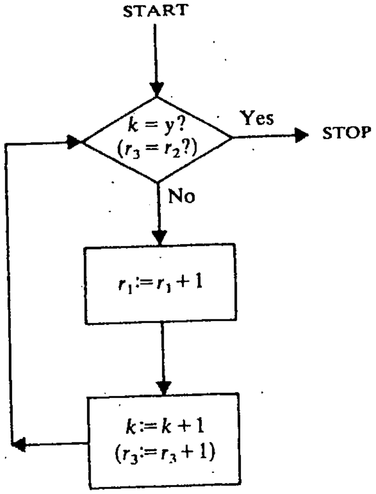
The program will be designed to stop when $k = y$, leaving $x + y$ in R_1 as required.

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	...

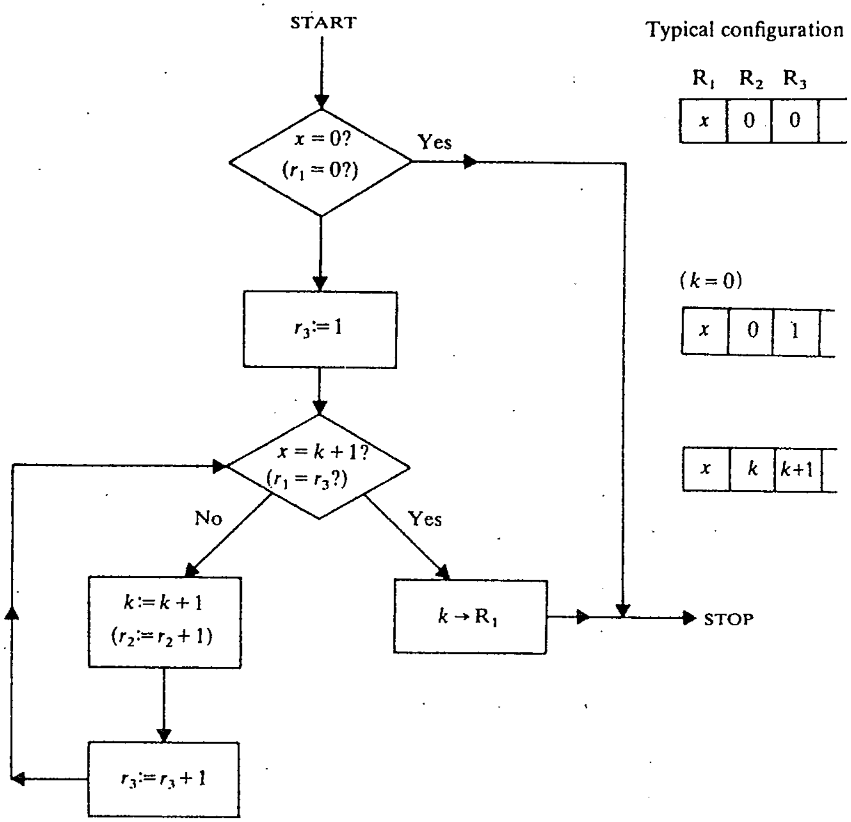
Check whether $x = k + 1$; if so, the required result is k; otherwise increase both counters by 1, and check again.

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

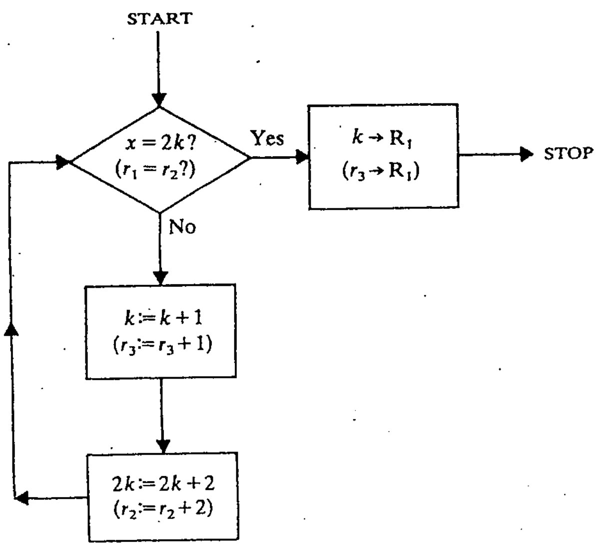
A procedure for computing $f(x)$ is as follows. Run two counters, containing k and $2k$ for $k=0,1,2,3,\ldots$; for successive values of k, check whether $x=2k$; if so, the answer is k; otherwise increase k by one, and repeat. If x is odd, this procedure will clearly continue for ever.

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* $c_M(\mathbf{x})$ (setting $\mathbf{x}=(x_1, \ldots, x_n)$) is given by

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

4.1 *Definition*

The predicate $M(\mathbf{x})$ is *decidable* if the function c_M is computable; $M(\mathbf{x})$ is *undecidable* if $M(\mathbf{x})$ 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*) $\mathbf{0}$: 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.