Chapter 1

Turing Machines

1 Basic Definitions

$N = \{0, 1, 2, \ldots\}$ is the set of natural numbers. In what follows we will feel free to say 'number', instead of 'natural number'.

Given a set A and a number $k \geq 0$, let A^k denote the set of all k-tuples of elements of A. Thus, by definition, we have that:
(i) A^0 is the singleton made out of the 0-tuple $\langle\rangle$ (also called *empty tuple* or *empty string* or *empty word* and denoted by ε), that is, $A^0 = \{\langle\rangle\}$,
(ii) $A^1 = \{\langle x \rangle \mid x \in A\}$, and
(iii) $A^k = \{\langle x_1, \ldots, x_k \rangle \mid x_1, \ldots, x_k \in A\}$ for any $k > 1$.
We identify $\langle x \rangle$ with x for each $x \in A$. Thus, $A^1 = A$. Sometimes a tuple $\langle x_1, \ldots, x_k \rangle$ is also denoted by $x_1 \ldots x_k$.
Let A^* denote the set of all finite k-tuples of elements of A, with $k \geq 0$, that is,

$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots \cup A^k \cup \ldots$

An element of A^* is also called a *string* or a *word*. The *length* of an element w of A^*, denoted $|w|$, is k iff w belongs to A^k, for any $k \geq 0$.

A set A is said to be *denumerable* (or *countable*) iff A has the cardinality of N or there exists a natural number n such that A has the same cardinality of n.

We have that A is denumerable iff A^* is denumerable. The proof is based on the dove-tailing technique [24, Chapter 1].

Definition 1. Given a set A we say that a binary relation $< \subseteq A \times A$ is a *total order* on A iff (i) $<$ is a transitive relation, and (ii) for any two elements $a, b \in A$ we have that exactly one of the three cases occurs: $\quad a < b, \quad a = b, \quad b < a.$

Let us consider a denumerable set A of symbols with a total order $<$ among its elements. Then the *canonical order*, on the set of all words in A^* is a binary relation, also denoted $<$, which is the subset of $A^* \times A^*$defined as follows. Given any two words w_1 and w_2 in A^*, we say that w_1 *canonically precedes* w_2, and we write $w_1 < w_2$, iff (i) *either* $|w_1|$ is smaller than $|w_2|$, (ii) *or* $|w_1| = |w_2|$ and $w_1 = p\,x\,q_1$ and $w_2 = p\,y\,q_2$ for some strings p, q_1, and q_2 in A^* and $x, y \in A$ such that $x < y$ (according to the order $<$ on A). We have that the order $<$ on A^* is transitive.

For instance, if $A = \{a, b\}$ and $a < b$ then the canonical order on A^* is the one which is denoted by the sequence: $\varepsilon < a < b < aa < ab < ba < bb < aaa < aab < aba < \ldots < bbb < \ldots < aaaa < \ldots$

According to the above definition, any total order is an irreflexive order. In the literature (see, for instance, [22, Chapter 1]) there is also a different notion of total order which is based on a partial order, and thus, it is a reflexive order. This second notion is defined as follows.

Definition 2. Given a set A we say that $\leq\, \subseteq A \times A$ is a *total order* on A iff (i) $\leq$ is reflexive, antisymmetric, and transitive (and thus, $\leq$ is a partial order), and (ii) for any two elements $a, b \in A$ we have that $a \leq b$ or $b \leq a$.

One can show that for any given total order $<$ on a set A according to Definition 1, there exists a total order $\leq$ according to Definition 2, such that for any two elements $a, b \in A$, we have that $a \leq b$ iff $a < b$ or $a = b$.

2 Turing Machines

Alan Turing in 1936 introduced an abstract model for computation called Turing Machine [31]. There are various variants of this model, which can be proved to be equivalent in the sense that they are all capable to compute the same set of functions from N to N.

Informally, we may say that a *Turing Machine* (TM, for short) M consists of: (i) a *finite automaton* FA, also called *control*, (ii) a one-way infinite tape divided into *cells* $\{c_i \mid i \in N,\ i > 0\}$, and (iii) a tape head which is on a cell at a time, called the *scanned cell*. Each cell contains exactly one of the symbols of the *tape alphabet* Γ. The states of the FA are also called *internal states* (or, simply, *states*), of the Turing Machine M.

We assume a left-to-right direction on the tape by stipulating that for $i > 0$ the cell c_i is immediately to the left of the cell c_{i+1}.

A Turing Machine M behaves is as follows. It starts on a tape containing in its cells $c_1\, c_2 \ldots c_n$ a sequence of n input symbols from the *input alphabet* Σ, while all other cells contain the symbol B (called *blank*) belonging to Γ. We assume that: $\Sigma \subseteq \Gamma - \{B\}$. The Turing Machine M starts with its tape head on the leftmost cell, that is, c_1, and the FA in its initial state q_0.

A *move* (or a *transition*) of the TM is given by a quintuple:

$$q_i,\, X_h \,\rightarrow\, q_j,\, X_k,\, m$$

where: (i) q_i is the current state of the FA, (ii) X_h is the symbol on the scanned cell (that is, the symbol which is read by the tape head), (iii) q_j is the new state of the FA, (iv) X_k is the symbol which, after the move, replaces X_h on the scanned cell (X_k is also called the printed symbol), and (v) m is either L or R and denotes that the tape head, after the move, will be on the cell either to the left or to the right, respectively, of the cell scanned before the move.

In Definition 6 we will give an alternative definition of a move of a Turing Machine. The context will tell the reader which of the two definitions is intended.

No two quintuples have the same first two components, and we refer to this property by saying that the TM is *deterministic*.

It is assumed that X_k is *not* B. From this assumption it follows that if the tape head reads a blank symbol then (i) all symbols to its left are not blank, and (ii) all symbols to its right (and the read symbol) are blank.

Definition 3. A *Turing Machine* (or a *TM*, for short) is a septuple $\langle Q, \Sigma, \Gamma, \delta, q_0, B, F\rangle$, where:
- Q is the set of *states*,
- Σ is the *input alphabet*,
- Γ is the *tape alphabet*,
- δ is a partial function from $Q \times \Gamma$ to $Q \times (\Gamma - \{B\}) \times \{L, R\}$, called the *transition function*, which defines the set of quintuples of the Turing Machine,
- q_0 is the initial state,
- B is the *blank symbol*, and
- F is the set of the *final states*.

We assume that Q and Γ are disjoint, and $\Sigma \subseteq \Gamma - \{B\}$.

Let us consider a Turing Machine whose tape is made out of the cells: $c_1 c_2 \ldots c_{h-1} c_h \ldots c_k$, where c_k with $1 \leq k$, is the rightmost cell with a non-blank symbol, and c_h with $1 \leq h$, is the cell scanned by the head. We have that: $1 \leq h \leq k{+}1$.

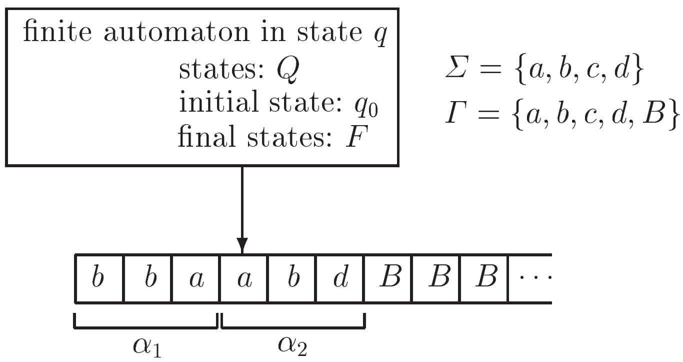


Fig. 1. A Turing Machine in the configuration $\alpha_1\ q\ \alpha_2$, that is, $b\,b\,a\,q\,a\,b\,d$.

Definition 4. A *configuration* of a TM whose tape head is reading the cell c_h for some $h \geq 1$, is the triple $\alpha_1\, q\, \alpha_2$, where:
- α_1 is the (possibly empty) sequence of $(\Gamma - \{B\})^{h-1}$ contained in the cells $c_1 c_2 \ldots c_{h-1}$,
- q is the current state of the TM, and
- α_2 is the non-empty sequence of Γ^{k-h+1} contained in the cells $c_h \ldots c_k$, if the tape head is reading a non-blank symbol (that is, $h \leq k$). Otherwise, α_2 is the sequence of one B only, if the tape head is reading a blank symbol B (that is, $h = k{+}1$).

For any given configuration we assume that the tape head is reading the leftmost symbol of α_2.

If the word which is initially written on the n leftmost cells of the tape a TM M is $w = a_1 a_2 \ldots a_n$, and all other cells contain B, then the initial configuration of M is $q_0 w$, that is, it is a configuration where α_1 is the empty sequence ε, the state of M is the initial state q_0, and $\alpha_2 = w$.

Definition 5. Given a Turing Machine whose configuration is $\alpha_1\, q\, \alpha_2$, we say that its *tape configuration* is the string $\alpha_1\, \alpha_2$ in Γ^*.

Sometimes, by abuse of language, we will simply say 'configuration', instead of 'tape configuration'. The context will tell the reader whether the intended meaning of the word 'configuration' is that of Definition 4 or that of Definition 5.

Definition 6. A *move* (or a *transition*) of a TM M is a pair of configurations of M of the form:

(i) $X_1 \ldots X_{i-2} X_{i-1}\, q\, X_i X_{i+1} \ldots X_n \quad \rightarrow_M \quad X_1 \ldots X_{i-2}\, p\, X_{i-1}\, Y\, X_{i+1} \ldots X_n$

if $\delta(q, X_i) = \langle p, Y, L\rangle$ for some $i > 1$ (that is, the tape head is *not* on the leftmost cell of the tape) or

(ii) $X_1 \ldots X_{i-2} X_{i-1}\, q\, X_i X_{i+1} \ldots X_n \quad \rightarrow_M \quad X_1 \ldots X_{i-2} X_{i-1}\, Y\, p\, X_{i+1} \ldots X_n$

if $\delta(q, X_i) = \langle p, Y, R\rangle$ for some $i \geq 1$ (that is, the tape head is reading any cell of the tape).

A move of the TM will also be called a *computation step* (or a *step*, for short). Notice that initially the tape head of a TM can only move to the right.

As usual, the reflexive and transitive closure of the relation $\rightarrow_M$ is denoted by $\rightarrow_M^*$.

We say that after a move a TM M *enters a final state* iff after that move M is in a final state.

We say that a TM *stops* (or *halts* or *gets stuck*) *in state* q iff no quintuple has q as first component. We say that a TM M *stops* (or *halts* or *gets stuck*) iff M is in a state q and M stops in state q.

We also say that a TM M *answers yes* iff M is in a *final* state q and M stops in q.

We also say that a TM M *answers no* iff after each move M made starting from the initial configuration, M has never entered a final state, and M stops in the present state.

Definition 7. Given a Turing Machine M, the set

$\{w \mid w \in \Sigma^* \text{ and } q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2 \text{ for some } q \in F,\ \alpha_1 \in \Gamma^*, \text{ and } \alpha_2 \in \Gamma^+\}$

which is a subset of Σ^*, is said to be the *language accepted* (or *recognized*) by M and it denoted by $L(M)$. Every word in $L(M)$ is said to be a word *accepted* (or *recognized*) by M.

The notion of language or word acceptance of the above definition is said to be *acceptance by final state*.

Thus, we have that a language L is accepted by a Turing Machine iff there exists a Turing Machine M such that for all words $w \in L$, M accepts w, that is, starting from the initial configuration $q_0 w$, the Turing Machine M after a finite number of moves, enters a final state. Notice that nothing is said about the behaviour of M on words which are not in L. The machine M either may get stuck without ever entering a final state or it may run forever without ever entering a final state.

We leave it as an exercise to the reader to show that if a language L is accepted by a Turing Machine M then there exists a Turing Machine $M1$ such that for every word in L, $M1$ enters a final state and stops.

Thus, for a word $w \in L(M)$ we may assume that M halts whenever M enters a final state, that is, if $q_0 w \rightarrow^*_M \alpha_1\, q\, \alpha_2$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, and $\alpha_2 \in \Gamma^+$ then no configuration γ exists such that $\alpha_1\, q\, \alpha_2 \rightarrow_M \gamma$.

Remark 1. Without loss of generality, we can assume that the transition function δ is a total function by adding a suitable *sink state* to the set Q of states. We stipulate that: (i) the sink state is *not* final, and (ii) every transition from the sink state takes the TM back to the sink state, whatever is the tape symbol printed in the scanned cell. □

Remark 2. We could allow the possibility of printing the blank symbol B. This possibility does not increase the power of a Turing Machine because the action of printing B, can be simulated by printing, instead, an extra symbol, say Δ, and then dealing with Δ as a given TM which can print B, deals with B. □

Definition 8. Given two Turing Machines M_1 and M_2 we say that they are *equivalent* iff $L(M_1) = L(M_2)$.

A Turing Machine M can be viewed as a devise for computing a partial function from N^k to N in the sense of the following definition.

Definition 9. We say that a Turing Machine M with $\Sigma = \{0, 1\}$ computes a partial function f from N^k to N iff for every $n_1, \ldots, n_k \in N$ we have that:
(i) $q_0 0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0 \rightarrow^*_M \alpha_1\, q\, 0\, 1^{f(n_1,\ldots,n_k)}\, 0^t$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, $t \geq 1$, and
(ii) starting from the initial configuration $q_0 0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0 q_0 0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0$, the Turing Machine M goes through a sequence of configurations in which $\alpha_1\, q\, 0\, 1^n\, 0^t$ is the only configuration which includes a final state.
In this case we say that: (i) from the input sequence $0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0$ the Turing Machine M computes the output sequence $0\, 1^{f(n_1,\ldots,n_k)}\, 0$, and (ii) the function f is *Turing computable*.

Remark 3. In the above Definition 9 we may also stipulate that if the TM M can print blanks, then t should be 1. Obviously, this extra condition does not modify the notion of a Turing computable function.

Example 1. The successor function $\lambda n.n{+}1$ is a Turing computable function. Indeed, the TM with the following transition function δ from the input sequence 01^n0 computes the output sequence $01^{n+1}0$:

$\delta(q_0, 0) = (q_1, 0, R)$;
$\delta(q_1, 1) = (q_1, 1, R)$; $\delta(q_1, 0) = (q_1, 1, R)$; $\delta(q_1, B) = (q_2, 0, L)$;
$\delta(q_2, 1) = (q_2, 1, L)$; $\delta(q_2, 0) = (q_3, 0, R)$;
$\delta(q_3, 1) = (q_F, 1, L)$

where q_F is the only final state. This transition function δ is depicted in Figure 2 where a quintuple:

$q_i,\ X_h \rightarrow_M q_j, X_k, m$

is represented as the following subgraph with the nodes q_i and q_j and the edge from q_i to q_j labelled by '$X_h\ \ (X_k, m)$':

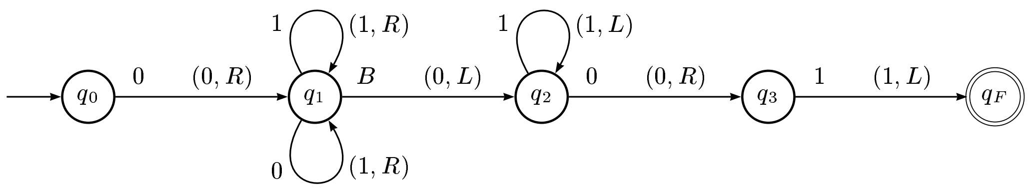


Fig. 2. A Turing Machine for computing the function $\lambda n.\, n{+}1$.

$$q_i \xrightarrow{X_h \qquad (X_k, m)} q_j$$

□

A Turing Machine M can also be viewed as a devise for computing a partial function on strings in the sense of the following definition.

Definition 10. Given the two alphabets Σ_1 and Σ_2 both subsets of Γ, we say that a Turing Machine M computes a partial function f from Σ_1^* to Σ_2^* iff for every $w \in \Sigma_1^*$ we have that:

(i) $q_0 w \;\rightarrow_M^*\; \alpha_1\, q\, b\, s$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, $b \in \Gamma$, and s in Σ_2^* with $s = f(w)$, and

(ii) starting from the initial configuration $q_0 w$, the Turing Machine M goes through a sequence of configurations in which $\alpha_1\, q\, b\, s$ is the only configuration which includes a final state.

The function f is said to be *Turing computable*.

In this definition we could have equivalently stipulated that $q_0 w \;\rightarrow_M^*\; \alpha_1\, q\, s$, instead of $q_0 w \;\rightarrow_M^*\; \alpha_1\, q\, b\, s$. Definition 10 has the advantage of making some Turing Machine constructions a bit simpler (see the following Example 2).

Notice that in what follows, we will feel free to use different conventions for the encoding of the input and output strings.

We leave it to the reader to generalize Definition 10 so that a Turing Machine can be viewed as a devise for computing partial functions from $(\Sigma_1^*)^k$ to Σ_2^* for any $k\,(\geq 0)$. In particular, the k input strings can be written on the tape by using a separating, distinguished character not in $\Sigma_1 \cup \Sigma_2$.

Example 2. Let us consider the function f from $\{a\}^*$ to $\{0, 1\}$ such that $f(w) = 0$ if the number of a's in w is even (that is, 0 or 2 or 4 or ...) and $f(w) = 1$ if the number of a's in w is odd (that is, 1 or 3 or ...). This function is Turing computable by a Turing Machine M such that: $\Sigma = \{a\}$, $\Gamma = \{a, 0, 1, B\}$, and the transition function δ is given by the graph of Figure 3.

Let us see how the Turing Machine M of Example 2 works if its input w is the empty string ε. Here is the corresponding sequence of tape configurations where ▲ denotes the position of the tape head:

1:

B	B	B	B	...
▲				

2:

0	B	B	B	...
	▲			

3:

0	0	B	B	...
▲				

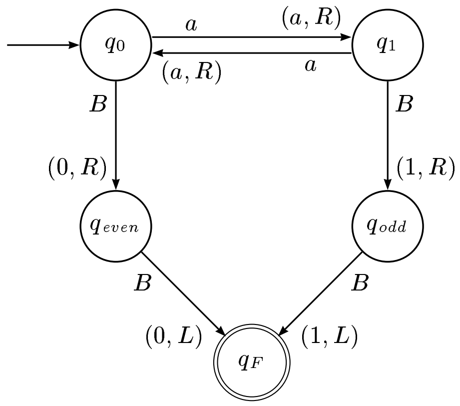
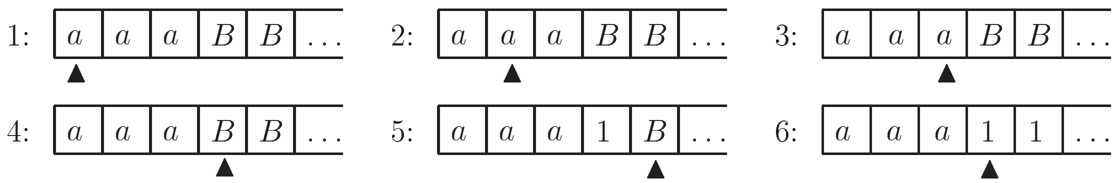


Fig. 3. A Turing Machine for checking the length of a string of a's.

In configuration 3 the 0 in the cell to the right of the scanned cell shows that the number of a's in w is even. Here is the sequence of tape configurations if the input w of the Turing Machine M of Example 2 is aaa:



In configuration 6 the 1 in the cell to the right of the scanned cell shows that the number of a's in w is odd.

We leave it as an exercise to the reader to show that in the Turing Machine of Figure 3, instead of 'B $(0, R)$', the label of the transition from state q_0 to state q_{even} can be any one of the form 'B (x, R)' for any $x \in \Gamma$. □

In what follows we will introduce the set of the partial recursive functions and in Section 17 we will state two theorems which identify the set of the partial recursive functions with the set of the Turing computable functions, both in the case of functions from N^k to N and in the case of functions from $(\Sigma_1^*)^k$ to Σ_2^*, for any $k \geq 0$.

Remark 4. We could have allowed three moves of the tape head: (i) L (*left*), (ii) R (*right*), and (iii) S (*stationary*), instead of the moves L and R only. This does not determines an increase of the power of the Turing Machine because the concatenation of some consecutive *stationary* moves followed by a *non-stationary* move, is equivalent to a *left* or a *right* move. Indeed, in a *stationary* move, after printing the new symbol on the tape, we know the new state of the Turing Machine and the new symbol which is read (which is the one that the Turing Machine has just printed) and, thus, we also know the configuration of the Turing Machine after its next move. □

Remark 5. In the literature [7] there is also a formalization of the Turing Machine via 4-tuples, instead of 5-tuples, in the sense that at each move of the Turing Machine: (i) either it writes of a symbol, (ii) or it moves one square to the left, (iii) or it moves one square to the right. In this formalization the Turing Machine can print the blank symbol. Any 4-tuple consists of: (i) the old state, (ii) the read symbol, (iii) the printed symbol or the move (to the left or to the right), and (iv) the new state. □

3 Techniques for Constructing Turing Machines

In this section we will present some techniques for constructing Turing Machines. The basic ideas will be presented through various examples.

3.1 Use of States to Store Information

The states of a TM can be used for storing information. We will illustrate this technique by looking at th following example. Let us assume that we want to construct a TM to check whether or not an input string belongs to the language $\{01^n \mid n \geq 0\} \cup \{10^n \mid n \geq 0\}$. If this is the case then the TM should enter a final state and an 1 should be in the cell to the right of the scanned cell.

Initially, in the tape we have the given string in $\{0,1\}^*$, and the cells $c_1\, c_2 \, \ldots \, c_{n+1}$ hold a string of length $n{+}1$. If that string is of the form 01^n or 10^n for some $n \geq 0$ then and only then the check is positive.

Here are the sets defining the desired TM.

$Q = \{p_0, p_1\} \times \{0, 1, a\}$,
$\Sigma = \{0, 1\}$,
$\Gamma = \{0, 1, B\}$,
$q_0 = [p_0, a]$, and
$F = \{[p_1, a]\}$.

The transition function δ is defined as depicted in Figure 4. The second component of the state is for storing the information about the first symbol of the input string which is either o or 1.

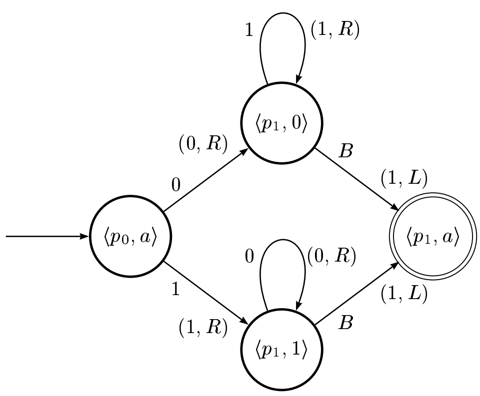


Fig. 4. A Turing Machine for testing whether or not a given string is an element of $\{01^n \mid n \geq 0\} \cup \{10^n \mid n \geq 0\}$.

3.2 Use of Multiple Tracks

If we have more than one track on the tape of a Turing Machine, it is sometimes easier to construct a Turing Machine which solves a given problem. Thus, it is useful in practice to consider Turing Machines with tapes which have multiple tracks. Notice, however, that the assumption that the tape has k tracks, for $k \geq 1$, is equivalent to the assumption that we have new symbols which are k-tuples of old symbols. The tape head looks at a k-tuple at a time. The new blank symbol is the tuple of k old blank symbols.

To have multiple tracks is useful when one has to perform various destructive operations on the same input string as shown by the following example.

Let us assume that we want to construct a Turing Machine for testing whether or not a given number is divisible by 3 and 5 (that is, by 15). We may consider three tracks, like, for instance, the following ones:

$$\begin{array}{ll} \text{track 1:} & \$\ 1\ 0\ 1\ 0\ 1\ 1\,1\,0\ \$\ B\ B\ldots \\ \text{track 2:} & B\ 1\ 0\ 1\ 0\ 1\ 0\,1\,1\ B\ B\ B\ldots \\ \text{track 3:} & B\ B\ B\ B\ B\ B\ B\,1\,1\ B\ B\ B\ldots \end{array}$$

The given number is written in binary on the first track between \$ symbols. The input symbols are: $\begin{vmatrix} \$ \\ B \\ B \end{vmatrix}, \begin{vmatrix} 0 \\ B \\ B \end{vmatrix}$, and $\begin{vmatrix} 1 \\ B \\ B \end{vmatrix}$ (in the three-track tape above, each symbol is represented in a vertical way by its three components).

The tape symbols are like the input symbols, but they may have either 0 or 1 or B in the second and in the third track.

One can test the divisibility by 3 and 5 of the given number: (i) by writing 3 in binary on the third track, (ii) by copying the input number on the second track, and (iii) repeatedly subtract 3 from the second track until 0 is found. Then, one should repeat this process by writing 5 in binary on the third track and copying the input number (which is kept unchanged on the first track) on the second track, and repeatedly subtract 5 from the second track until 0 is found. In this case and only in this case, the given number is divisible by 3 and 5.

For instance, copying the input on the second track is performed by quintuples of the form:

$$q,\ \begin{vmatrix} 1 \\ y \\ z \end{vmatrix} \ \rightarrow\ q,\ \begin{vmatrix} 1 \\ 1 \\ z \end{vmatrix},\ R \quad \text{for any } y, z \in \{B, 0, 1\}$$

$$q,\ \begin{vmatrix} 0 \\ y \\ z \end{vmatrix} \ \rightarrow\ q,\ \begin{vmatrix} 0 \\ 0 \\ z \end{vmatrix},\ R \quad \text{for any } y, z \in \{B, 0, 1\}$$

We leave to the reader as an exercise to complete the description of the desired TM.

3.3 Test of String Equality

In this section we present an example where we construct a Turing machine which computes the function f from $\{0,1\}^* \times \{0,1\}^*$ to $\{0,1\}$ such that

$f(x, y) = \text{if } x = y \text{ then } 1 \text{ else } 0.$ (†)

This TM has a tape with two tracks and thus, we assume that the symbols of the input alphabet and the tape alphabet are pairs. Here is the definition of the desired TM M.

$Q = \{p_1, p_{10}, p_{11}, p_{20}, p_{21}, p_3, p_4, p_5, p_6\}$,

$\Sigma = \left\{ \begin{vmatrix} 0 \\ B \end{vmatrix}, \begin{vmatrix} 1 \\ B \end{vmatrix}, \begin{vmatrix} \# \\ B \end{vmatrix} \right\}$,

$\Gamma = \left\{ \begin{vmatrix} 0 \\ B \end{vmatrix}, \begin{vmatrix} 0 \\ \checkmark \end{vmatrix}, \begin{vmatrix} 1 \\ B \end{vmatrix}, \begin{vmatrix} 1 \\ \checkmark \end{vmatrix}, \begin{vmatrix} \# \\ B \end{vmatrix}, \begin{vmatrix} B \\ B \end{vmatrix} \right\}$,

the initial state is p_1,

the blank symbol is $\begin{vmatrix} B \\ B \end{vmatrix}$, and

$F = \{p_6\}$.

The symbols 0 and 1 of the specification (†) are represented within the Turing Machine M by $\begin{vmatrix} 0 \\ B \end{vmatrix}$ and $\begin{vmatrix} 1 \\ B \end{vmatrix}$, respectively. As usual, the output symbol will be written to the right of the cell scanned by the tape head when the final state is entered. Thus, the input strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$ are represented on the leftmost position of the tape as follows:

track 1: $x_1\ x_2 \ldots x_n\ \#\ y_1\ y_2 \ldots y_m\ B\ B \ldots$

track 2: $B\ B\ B\ \ B\ B\ B\ B\ B\ \ B\ B\ B \ldots$

and initially, the tape head looks at the leftmost cell where $\begin{vmatrix} x_1 \\ B \end{vmatrix}$ is placed.

The transition function δ is defined as follows. First of all, the TM M checks whether or not $n = 0$. If $n = 0$ then M checks that also $m = 0$ and, if this is the case, it returns $\begin{vmatrix} 1 \\ B \end{vmatrix}$ to the right of the cell scanned by the head when the final state is entered (see quintuples 1 and 2 below).

If $n > 0$ and $x_1 = 0$ (the case when $x_1 = 1$ is analogous) then M checks it off by writing $\begin{vmatrix} 0 \\ \checkmark \end{vmatrix}$, moves to the right, passes $\begin{vmatrix} \# \\ B \end{vmatrix}$, and looks for the leftmost unchecked symbol of y which should be 0. If M finds it, M checks it off by writing $\begin{vmatrix} 0 \\ \checkmark \end{vmatrix}$ and moves to the left, passing over $\begin{vmatrix} \# \\ B \end{vmatrix}$ and looks for the leftmost unchecked symbol of x.

Then the Turing Machine M starts the process again until the leftmost unchecked symbol is $\begin{vmatrix} \# \\ B \end{vmatrix}$. At that point it makes sure that all symbols to the right of $\begin{vmatrix} \# \\ B \end{vmatrix}$ are checked (that is, they have on the lower track $\checkmark$) and it terminates successfully (that is, it writes $\begin{vmatrix} 1 \\ B \end{vmatrix}$, moves left, and stops). In all other cases the termination of the TM is unsuccessful (that is, it writes $\begin{vmatrix} 0 \\ B \end{vmatrix}$, moves left, and stop).

We have the following quintuples:

1. $p_1, \left|\begin{matrix}\#\\ B\end{matrix}\right| \to p_5, \left|\begin{matrix}\#\\ B\end{matrix}\right|, R$ 2. $p_5, \left|\begin{matrix}B\\ B\end{matrix}\right| \to p_6, \left|\begin{matrix}1\\ B\end{matrix}\right|, L$

The following two quintuples look for $\left|\begin{matrix}B\\ B\end{matrix}\right|$ immediately to the right of the checked symbols:

3. $p_5, \left|\begin{matrix}0\\ \checkmark\end{matrix}\right| \to p_5, \left|\begin{matrix}0\\ \checkmark\end{matrix}\right|, R$ 4. $p_5, \left|\begin{matrix}1\\ \checkmark\end{matrix}\right| \to p_5, \left|\begin{matrix}1\\ \checkmark\end{matrix}\right|, R$

Notice that in state p_5 if some symbols of y are not checked then $x \neq y$ because, if this the case, the strings x and y have different length.

The following quintuples are needed for checking off symbols on the strings x and y. If the symbol of x to be checked is 0 we use the quintuples 0.1–0.5, if it is 1 we use the quintuples 1.1–1.5.

0.1 $p_1, \left|\begin{matrix}0\\ B\end{matrix}\right| \to p_{10}, \left|\begin{matrix}0\\ \checkmark\end{matrix}\right|, R$ Checking off the leftmost unchecked symbol of x. Let it be 0.

0.2 $p_{10}, \left|\begin{matrix}s\\ B\end{matrix}\right| \to p_{10}, \left|\begin{matrix}s\\ B\end{matrix}\right|, R$ Moving over to y. (2 quintuples: for $s = 0, 1$)

0.3 $p_{10}, \left|\begin{matrix}\#\\ B\end{matrix}\right| \to p_{20}, \left|\begin{matrix}\#\\ B\end{matrix}\right|, R$ Passing # and going over to y.

0.4 $p_{20}, \left|\begin{matrix}s\\ \checkmark\end{matrix}\right| \to p_{20}, \left|\begin{matrix}s\\ \checkmark\end{matrix}\right|, R$ Looking for the leftmost unchecked symbol of y. (2 quintuples: for $s = 0, 1$)

0.5 $p_{20}, \left|\begin{matrix}0\\ B\end{matrix}\right| \to p_3, \left|\begin{matrix}0\\ \checkmark\end{matrix}\right|, L$ Now we go back to string x.

1.1 $p_1, \left|\begin{matrix}1\\ B\end{matrix}\right| \to p_{11}, \left|\begin{matrix}1\\ \checkmark\end{matrix}\right|, R$ Checking off the leftmost unchecked symbol of x. Let it be 1.

1.2 $p_{11}, \left|\begin{matrix}s\\ B\end{matrix}\right| \to p_{11}, \left|\begin{matrix}s\\ B\end{matrix}\right|, R$ Moving over to y. (2 quintuples: for $s = 0, 1$)

1.3 $p_{11}, \left|\begin{matrix}\#\\ B\end{matrix}\right| \to p_{21}, \left|\begin{matrix}\#\\ B\end{matrix}\right|, R$ Passing # and going over to y.

1.4 $p_{21}, \left|\begin{matrix}s\\ \checkmark\end{matrix}\right| \to p_{21}, \left|\begin{matrix}s\\ \checkmark\end{matrix}\right|, R$ Looking for the leftmost unchecked symbol of y. (2 quintuples: for $s = 0, 1$)

1.5 $p_{21}, \left|\begin{matrix}1\\ B\end{matrix}\right| \to p_3, \left|\begin{matrix}1\\ \checkmark\end{matrix}\right|, L$ Now we go back to string x.

The following quintuples allow us to go back to the string x and to reach the leftmost unchecked symbol of x, if any. Then we check off that symbol of x.

5. $p_3, \left|\begin{matrix}s\\ \checkmark\end{matrix}\right| \to p_3, \left|\begin{matrix}s\\ \checkmark\end{matrix}\right|, L$ Moving over to x. (2 quintuples: for $s = 0, 1$)

6. $p_3, \left|\begin{matrix}\#\\ B\end{matrix}\right| \to p_4, \left|\begin{matrix}\#\\ B\end{matrix}\right|, L$ Passing # and going over to x.

7. $p_4, \left|\begin{matrix} s \\ B \end{matrix}\right| \rightarrow p_4, \left|\begin{matrix} s \\ B \end{matrix}\right|, L$ Looking for the leftmost unchecked symbol of x.
(2 quintuples: for $s = 0, 1$)

8. $p_4, \left|\begin{matrix} s \\ \checkmark \end{matrix}\right| \rightarrow p_1, \left|\begin{matrix} s \\ \checkmark \end{matrix}\right|, R$ Ready to check off one more symbol of x.
(2 quintuples: for $s = 0, 1$)

All undefined transitions should be made to go to the final state p_6 and the symbol to the right of the scanned cell should be $\left|\begin{matrix} 0 \\ B \end{matrix}\right|$, denoting that $x \neq y$.

For instance, for $x = y = 0010$, we have the following sequence of tape configurations with two tracks. The initial configuration is:

$$\begin{matrix} 0 & 0 & 1 & 0 & \# & 0 & 0 & 1 & 0 & B & B \ldots \\ B & B & B & B & B & B & B & B & B & B & B \ldots \end{matrix}$$

then we get:

$$\begin{matrix} 0 & 0 & 1 & 0 & \# & 0 & 0 & 1 & 0 & B & B \ldots \\ \checkmark & B & B & B & B & B & B & B & B & B & B \ldots \end{matrix}$$

After some moves to the right, having checked the equality of the first symbols of the two sequences, we have:

$$\begin{matrix} 0 & 0 & 1 & 0 & \# & 0 & 0 & 1 & 0 & B & B \ldots \\ \checkmark & B & B & B & B & \checkmark & B & B & B & B & B \ldots \end{matrix}$$

and after some moves to the left, looking for the second symbols to check, we have:

$$\begin{matrix} 0 & 0 & 1 & 0 & \# & 0 & 0 & 1 & 0 & B & B \ldots \\ \checkmark & \checkmark & B & B & B & \checkmark & B & B & B & B & B \ldots \end{matrix}$$

and so on, until at the end, we have:

$$\begin{matrix} 0 & 0 & 1 & 0 & \# & 0 & 0 & 1 & 0 & 1 & B \ldots \\ \checkmark & \checkmark & \checkmark & \checkmark & B & \checkmark & \checkmark & \checkmark & \checkmark & B & B \ldots \\ & & & & & & & & \blacktriangle & & \end{matrix}$$

and the tape head is at the cell marked with $\blacktriangle$.

3.4 Shift of Strings

A TM may shift the content of its tape of $k\,(> 0)$ cells to the right (or to the left, if there are cells available). Let us consider the case of a right shift of 1 cell. This shift can be obtained by: (i) going to the leftmost cell c_1, (ii) storing in a state of the control the symbol in c_1, (iii) printing in c_1 a symbol to denote that the cell is empty, and while moving to the right, (iv) storing in a state of the control the symbol read in the scanned cell, and (v) printing in the scanned cell the symbol which was to the left of it and was stored in a state of the control during the previous move.

When a TM shifts the content of its tape, we say that it has applied the *shifting-over* technique.

3.5 Use of Subroutines

We restrict our attention to subroutines without recursion and without parameters. Later on in Section 8 we shall see how to overcome this limitation by explaining in an informal way how a Turing Machine can simulate a Random Access Machine. If a TM, say S, is a subroutine for another TM, say M, we first require that the set of states of S is disjoint from the set of states of M. When a call to the subroutine S is made, the initial state of S is entered and when the subroutine S has finished, M enters one of its states and proceeds.

We present the idea by way of the following example. Let Σ be $\{0, 1\}$. Let us consider the Turing Machine *Sum* which takes the initial tape configuration:

$\ldots \$ \lceil n \rceil \; \# \; \lceil m \rceil \; \$ \ldots$

where $\lceil n \rceil$ and $\lceil m \rceil$ are strings of bits (each one in a separate cell) denoting the two non-negative numbers n and m written in binary (using enough bits to allow the process to continue as specified below). The machine *Sum* derives the final tape configuration:

$\ldots \$ \; \Delta \; \Delta \ldots \Delta \; \lceil n{+}m \rceil \; \$ \ldots$

where Δ's are printable blanks. We also assume that *Sum* never goes outside the tape section between \$'s. The TM *Mult* for multiplying two natural numbers can be constructed by making use of the subroutine *Sum* as follows.

(1) *Mult* starts from the tape configuration:

$\ldots \lceil n \rceil \; \# \; \lceil m \rceil \; \$ \; \lceil 0 \rceil \; \# \; \lceil 0 \rceil \; \$ \ldots$

(2) *Mult* gives control to *Sum* and eventually we get:

$\ldots \lceil n \rceil \; \# \; \lceil m \rceil \; \$ \; \Delta \; \Delta \ldots \Delta \; \lceil s \rceil \; \$ \ldots$

where s is the sum of the numbers between \$'s. The first time $s = 0$.

(3) *Sum* returns the control to *Mult*, and
if $n = 0$ *then* *Mult* produces the result s
else begin *Mult* derives $\lceil n{-}1 \rceil$ from $\lceil n \rceil$ and copies $\lceil m \rceil$
so that eventually we get the tape configuration:

$\ldots \lceil n{-}1 \rceil \; \# \; \lceil m \rceil \; \$ \; \lceil m \rceil \; \# \; \lceil s \rceil \; \$ \ldots$

and then *Mult* goes back to (2)
end

We leave it as an exercise to the reader to complete the details of this construction of the TM *Mult*.

4 Extensions of Turing Machines Which Do Not Increase their Power

4.1 Two-way Infinite Tape

We may assume a different definition of TM where the tape is infinite in both directions. We have that a Turing Machine M_2 whose tape is infinite in both directions, can be simulated by a Turing Machine M_1 whose tape is one-way infinite as implicitly specified by Definition 3 and Definition 7.

Let $M_2 = \langle Q_2, \Sigma_2, \Gamma_2, \delta_2, q_2, B, F_2\rangle$ be the given TM with the two-way infinite tape. The following TM $M_1 = \langle Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, B, F_1\rangle$ with the one-way infinite tape with two tracks simulates M_2 as we now indicate (see also Figure 5).

M_2 :

$\dots$	a_{-3}	a_{-2}	a_{-1}	a_0	a_1	a_2	a_3	$\dots$

M_1 :

a_0	a_1	a_2	a_3	$\dots$	U: up
\$	a_{-1}	a_{-2}	a_{-3}	$\dots$	D: down

Fig. 5. Simulating a two-way infinite tape by a one-way infinite tape.

For the Turing Machine M_1 we stipulate that:

$Q_1 = \{\langle q, U\rangle, \langle q, D\rangle \mid q \in Q_2\} \cup \{q_1\}$,

$\Sigma_1 = \{\begin{vmatrix} a \\ B \end{vmatrix} \mid a \in \Sigma_2\}$,

$\Gamma_1 = \{\begin{vmatrix} a \\ b \end{vmatrix} \mid a, b \in \Gamma_2\} \cup \{\begin{vmatrix} a \\ \$ \end{vmatrix} \mid a \in \Gamma_2\}$ where \$ is not in Γ_2,

the initial state is q_1, and

$F_1 = \{\langle q, U\rangle, \langle q, D\rangle \mid q \in F_2\}$.

U stands for 'up' and D for 'down', corresponding to the upper and lower track, respectively.

Now, we assume that if the input string of the machine M_2 is placed on the cells, say $0, 1, \dots, n$ with symbols $a_0, a_1, \dots, a_n$, respectively (see Figure 5) and all remaining cells have blanks, then in the machine M_1 the input string is placed in leftmost cells with content $\begin{vmatrix} a_0 \\ B \end{vmatrix}, \begin{vmatrix} a_1 \\ B \end{vmatrix}, \dots, \begin{vmatrix} a_n \\ B \end{vmatrix}$, respectively. We also assume that the tape head of M_2 is at the cell 0 and that of $M_1$is at its leftmost cell, as usual. The moves of the machine M_1 are defined in terms of the moves of M_2 according to the following rules.

1. Initial moves of M_1:

for each $a \in \Sigma_2 \cup \{B\}$,

$q_1, \begin{vmatrix} a \\ B \end{vmatrix} \to \langle q, U\rangle, \begin{vmatrix} X \\ \$ \end{vmatrix}, R$ if in M_2 we have: $q_2, a \to q, X, R$

$q_1, \begin{vmatrix} a \\ B \end{vmatrix} \to \langle q, D\rangle, \begin{vmatrix} X \\ \$ \end{vmatrix}, R$ if in M_2 we have: $q_2, a \to q, X, L$

M_1 prints \$ on the leftmost cell of the lower track and continues working as M_2.

2.1 M_1 simulates the moves of M_2 on the upper track:

for each $\begin{vmatrix} a \\ Y \end{vmatrix} \in \Gamma_1$ with $Y \neq \$$,

$$\langle q, U\rangle, \begin{vmatrix} a \\ Y \end{vmatrix} \rightarrow \langle p, U\rangle, \begin{vmatrix} b \\ Y \end{vmatrix}, m \quad \text{if in } M_2 \text{ we have: } q, a \rightarrow p, b, m$$

2.2 M_1 simulates the moves of M_2 on the lower track:

for each $\begin{vmatrix} Y \\ a \end{vmatrix} \in \Gamma_1$,

$$\langle q, D\rangle, \begin{vmatrix} Y \\ a \end{vmatrix} \rightarrow \langle p, D\rangle, \begin{vmatrix} Y \\ b \end{vmatrix}, R \quad \text{if in } M_2 \text{ we have: } q, a \rightarrow p, b, L$$

$$\langle q, D\rangle, \begin{vmatrix} Y \\ a \end{vmatrix} \rightarrow \langle p, D\rangle, \begin{vmatrix} Y \\ b \end{vmatrix}, L \quad \text{if in } M_2 \text{ we have: } q, a \rightarrow p, b, R$$

3. Possible change of track:

$$\langle q, U\rangle, \begin{vmatrix} a \\ \$ \end{vmatrix} \rightarrow \langle p, D\rangle, \begin{vmatrix} b \\ \$ \end{vmatrix}, R \quad \text{if in } M_2 \text{ we have: } q, a \rightarrow p, b, L$$

(from the upper track to the lower track)

$$\langle q, U\rangle, \begin{vmatrix} a \\ \$ \end{vmatrix} \rightarrow \langle p, U\rangle, \begin{vmatrix} b \\ \$ \end{vmatrix}, R \quad \text{if in } M_2 \text{ we have: } q, a \rightarrow p, b, R$$

(remaining in the upper track)

$$\langle q, D\rangle, \begin{vmatrix} a \\ \$ \end{vmatrix} \rightarrow \langle p, D\rangle, \begin{vmatrix} b \\ \$ \end{vmatrix}, R \quad \text{if in } M_2 \text{ we have: } q, a \rightarrow p, b, L$$

(remaining in the lower track)

$$\langle q, D\rangle, \begin{vmatrix} a \\ \$ \end{vmatrix} \rightarrow \langle p, U\rangle, \begin{vmatrix} b \\ \$ \end{vmatrix}, R \quad \text{if in } M_2 \text{ we have: } q, a \rightarrow p, b, R$$

(from the lower track to the upper track)

We leave it to the reader to convince himself that M_1 is equivalent to M_2 (see Definition 8).

4.2 Multitape Turing Machines

A *multitape* TM has k tapes with $k > 1$, and each tape has its own head. A move depends on the k-tuple of symbols read by the heads. In a single move k new symbols are printed (they may be all different), one on each tape, and the heads are moved, left or right, independently. We assume that the input is written on a given tape, say tape 1, and the tapes 2, 3, ..., and k are all filled with blanks.

Let us informally explain how a multitape Turing Machine can be simulated by a single tape Turing Machine.

A k-tape TM M_k is simulated by a $2k$-track TM M, each tape being represented by two tracks: on the upper track there is a copy of the corresponding tape and on the lower track there is either a blank symbol or the 'head marker' ▲ to denote the position of the tape head (see Figure 6 where we did not show the blank symbols). Initially the tape head of M is on the position corresponding to the leftmost marker ▲. The internal states of M hold the information of both (i) the internal states of M_k and (ii) the count of the head markers to the left of the head of M itself.

A move m of M_k is simulated by the TM M as follows. M moves its head from the leftmost to the rightmost head marker and then back to the leftmost head marker. While the move to the right proceeds, M recalls in its internal states the number of head markers passed, the symbols scanned by each head marker. When all head markers have been passed, M

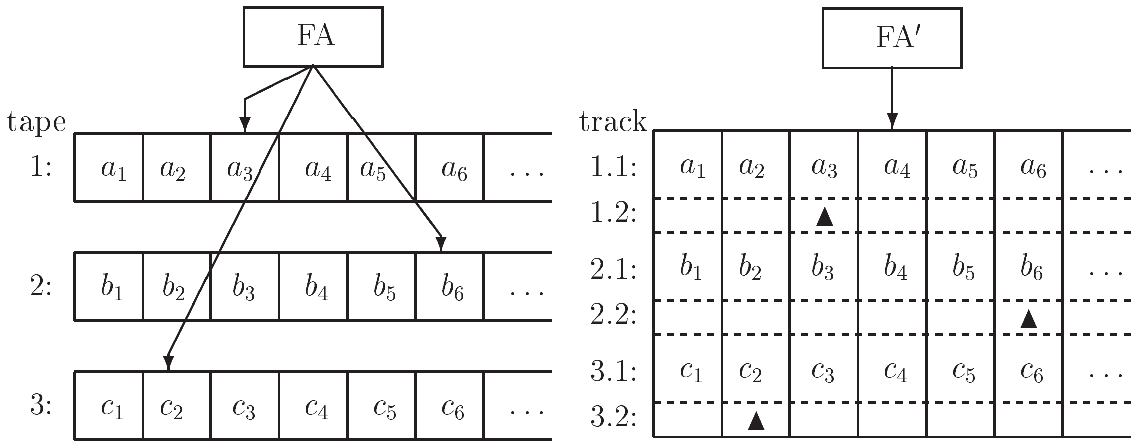


Fig. 6. Simulating a 3-tape Turing Machine by a 6-track Turing Machine.

has the necessary information for simulating the move m of M_k. Thus, M changes its internal state according to the move m to be simulated, and makes a leftwards movement until the leftmost head marker is reached. During that movement M updates its tape configuration (that is, updating both the symbols scanned by the head markers and the positions of the head markers) according to the move m. As done in the movement to the right, in its movement to the left M counts in its internal states the number of the head markers visited and then M will know when to stop its movement to the left. Having simulated the printing and the head movements, at the end M changes its internal state to simulate the change of the internal state of M_k.

It should be noticed that when moving back to the left, M may be forced to make some one-cell movements to the right for simulating one-cell movements to the right of some of the heads of M_k.

In the simulation we have described, M requires many moves for simulating a single move of M_k. Indeed, in this simulation (unlike the simulation of the two-way infinite tape TM by the one-way infinite tape TM) there is a slowdown, because at each move which is simulated the distance of the leftmost head from the rightmost head may increase by 2 cells. Thus, the head of the simulating machine M may have to travel $2r$ cells when simulating the r-th move of M_k.

The slowdown is quadratic because: (i) if we disregard the distance to travel, the number of transitions to be made by M for simulating the moves of M_k is constant, being k fixed, and (ii) the sum of the distances after r moves, apart from some constants which we do not take into consideration, is $2+4+6+\ldots+2r$, that is, $O(r^2)$.

4.3 Multihead Turing Machines

Multihead Turing machines have many heads which move independently of each other, and the transition of the machine depends on the internal state and on the symbols which are scanned. A quintuple of a Turing Machine with r heads looks as follows:

$q, \langle x_1, \ldots x_r \rangle \;\rightarrow\; p, \langle y_1, \ldots y_r \rangle, \langle m_1, \ldots m_r \rangle$
where for $i = 1, \ldots, r$, x_i and y_i are tape symbols, and m_i is an element of $\{L, R\}$.

The simulation of this kind of Turing Machines by a TM with one head only is similar to the one of the multitape TM. The only difference is that now the simulating one-head Turing Machine need $k+1$ tracks, k tracks for the k heads and one track for the tape.

4.4 Off-line and On-line Turing Machines

In this section we introduce two more kinds of Turing Machines which are considered in the literature.

The first one, called the *off-line Turing Machine*, is considered when studying Complexity Theory and, in particular, when analysing the amount of space required by Turing Machine computations.

An off-line Turing Machine is similar to a two-tape Turing Machine with the limitation that there is a tape which contains the input, called the *input tape*, between the special symbols, say ¢ and \$, and it can be read, but not modified. Moreover, it is not allowed to use this tape outside the cells where the input is written (see Figure 7).

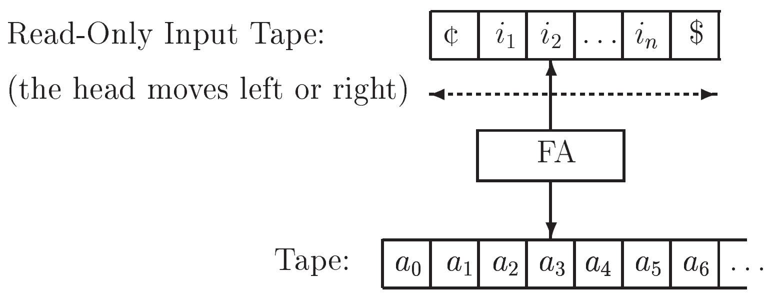


Fig. 7. An off-line Turing (the lower tape may equivalently be two-way or one-way infinite).

An off-line Turing Machine is equivalent to a standard Turing Machine (that is, the one of Definition 3) because: (i) an off-line Turing Machine is a special case of a two-tape Turing Machine, which is equivalent to a standard Turing Machine, and (ii) for any given Turing Machine M and input w there exists an off-line Turing Machine M_{off} with input w which can simulate M on the input w. Indeed, M_{off} first copies its input which is on its input tape, onto the other tape, and then M_{off} behaves as M does.

The second kind of Turing Machine we consider in this section is the *on-line Turing Machine*. This kind of Turing Machine is used for studying the so called *on-line algorithms* [11] and their complexity.

The on-line Turing Machine is like the off-line Turing Machine with the assumption that the head on the input tape initially scans the symbol ¢ and then it can remain stationary or move to the right. It cannot move to the left. When the head on the input tape moves to the right and by that move it scans \$, then the finite control of the Turing Machine enters a state which is *either* a final state (and this is the case when the on-line

Turing Machine accepts its input) *or* a non-final state (and this is the case when the on-line Turing Machine does not accept its input). Other equivalent conventions on the final reading move are described in the literature.

4.5 Bidimensional and Multidimensional Turing Machines

In this section we look at a Turing Machine with a bidimensional tape. The cases with multidimensional tapes are similar. The head of the machine may move left, right, up, and down, and before moving it prints a symbol on the scanned cell. At each moment there exists a rectangle outside which all cells are filled with blanks. (see Figure 8)

B	r	e	s	B	B	B
d	s	a	u	B	k	B
B	B	t	n	B	B	B
B	B	B	h	B	B	B

Fig. 8. The non-blank portion of the tape of a two-dimensional Turing Machine.

A bidimensional TM M_2 can be simulated by a standard TM M_1 with two tapes as follows. The need for the second tape is simply for explanatory reasons, and it related to the fact that for the simulation we need to count the number of the special symbols $*$ and \$ (see below). The basic idea is to simulate *line-by-line*, *top-down* a bidimensional configuration by a linear configuration. For instance, a possible line-by-line, top-down encoding of the tape configuration of Figure 8 is:

$\$\, B\, r\, e\, s\, B\, B\, B * d\, s\, a\, u\, B\, k\, B * B\, B\, t\, n\, B\, B\, B * B\, B\, B\, h\, B\, B\, B\, \$$

where the symbol $*$ is a new symbol, not in the tape alphabet of M_2. This line-by-line representation is the usual one for television images. $*$ denotes the end of a line. \$ denotes, so to speak, the upper left corner and the lower right corners of the rectangle. We may assume that the input to the machine M_2 is given a sequence of symbols in one dimension only, say the horizontal dimension. Horizontal moves of M_2 inside the current rectangle are simulated in the obvious way by the TM M_1. Vertical moves inside the current rectangle can be simulated by counting the distance from the nearest left $*$ so that it is possible to place the head on the proper position in the upper or lower line.

If a move goes outside the current rectangle and it is vertical and upwards then M_1 replaces \$ to the left of the current configuration by $\$\, B\, B\, \ldots\, B\, *$, where the number of B's are counted in the second tape by finding first the length of a line. This operation corresponds to the insertion of a new line.

The case in which the move is downwards is similar, but the machine M_1 should operate at the right end of the configuration.

If the move of M_2 which goes outside the rectangle is to the right then M_1 should first add an extra blank at the end of each line (and this can be done by shifting) and then it simulates the move of M_2. Analogously for the case when the move is to the left.

This simulation of the bidimensional TM can be extended to the case when there are many heads or many multidimensional tapes by combining together the various techniques presented in the previous sections.

4.6 Deterministic and Nondeterministic Turing Machines

Now we show that nondeterministic Turing Machines are equivalent to deterministic Turing Machines (that is, the ones of Definition 3).

A nondeterministic TM M is like a standard TM (that is the one of Definition 3), but the condition that the transition function δ be a partial function is released in the sense that there can be more that one quintuple with the same first two components. Thus, $\rightarrow_M$ is a binary relation, not a function. The notion of the language accepted by a nondeterministic Turing Machine M, denoted by $L(M)$, is like the one for a deterministic TM. Thus, we have:

$$L(M) = \{w \mid w \in \Sigma^* \text{ and } q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2 \text{ for some } q \in F,\ \alpha_1 \in \Gamma^*, \text{ and } \alpha_2 \in \Gamma^+\}$$

Often one refers to the nondeterminism of a Turing Machine as being *angelic*, in the sense that $w \in L(M)$ iff *there exists* a sequence of quintuples in the transition relation δ such that $q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, and $\alpha_2 \in \Gamma^+$.

The deterministic Turing Machine M which simulates a given nondeterministic Turing Machine N, uses three tapes and it works as follows. (The inverse simulation is obvious because the deterministic TM is a particular case of the nondeterministic TM.)

Let us assume that the transition relation of the nondeterministic Turing Machine N has at most $b\,(\geq 2)$ quintuples with the same first two components. We also assume that the quintuples of the transition relation of N are ordered, so that it makes sense to choose the n-th quintuple, for $n = 1, \ldots, b$, among all those with the same first two components. If n is larger than the number of quintuples with the given first two components, then we say that we made an *improper choice* of the quintuple. On the first tape M keeps a copy of the input of N. On the second tape M generates in the canonical order (see Section 1) the strings in $\{1, \ldots, b\}^*$, where $1 < \ldots < b$, that is, $\varepsilon, 1, \ldots, b, 11, \ldots, 1b, 21, \ldots, 2b, \ldots, b1, \ldots, bb, 111, \ldots, 11b, \ldots,$, one at a time (ε is the empty sequence). After any move only one string appears on the second tape. It is clear that after a finite number of moves, any given string s of k numbers, each in $\{1, \ldots, b\}$, will appear on the second tape. It is also clear that M can compute any next string of the canonical order from the previous one it finds written on the tape.

M simulates N by first copying on the third tape the input (which is on its first tape) and then reading from-left-to-right the sequence of numbers, each one in $\{1, \ldots, b\}$, written on the second tape as a sequence of instructions for choosing the quintuple among those available at each move. The empty sequence means that no move should be made. Suppose that the sequence is: 2 3 1 3 4. It means that on the first move, for the given scanned symbol and the given state, M should choose the 2nd quintuple among the quintuples which apply for the first move, then for the 2nd move M should choose the 3rd quintuple among the quintuples which apply for the second move, for the 3rd move M should choose the 1st quintuple among the quintuples which apply for the third move, and so on.

If the sequence on the second tape forces an improper choice then M proceeds as follows.

(1) M generates on the second tape the sequence which in the canonical order follows the sequence which is currently written on the tape.

(2) Then M copies again the input from the first tape onto the third tape and makes its moves as specified by the sequence, by reading the numbers of the sequence from-left-to-right. If the sequence has been completely exhausted and M did not enter a final state then M proceeds by first generating the next sequence on the second tape and proceeding as indicated in this Point (2).

If after any move the machine M ever enters a final state then M stops and accepts the input word.

Now, it is obvious that for any input word w, if M accepts w then there exists a sequence of choices among the various alternative for the application of the transition relation δ such that N accepts w. Since by the canonical generation, M generates in a finite number of moves any given sequence of numbers, M accepts w iff N accepts w.

In order for M to simulate N (in the sense that $L(N) = L(M)$) it is not necessary that M generates the sequences of numbers precisely in the canonical order. Other orders are possible as we now indicate.

Let us first notice that a set of strings which is closed under the prefix operation naturally defines a tree. In particular, we may consider a sequence of numbers as a node on a tree, call it T. For instance, in Figure 9, the sequence $2\,2\,4\,4\,3$ corresponds to the node n. Now, in order for M to simulate N it is enough that M generates the sequences of numbers so that the generation of the corresponding nodes of the tree T satisfies the following property:

Property (A): for any already generated node n of T, after a finite number of moves, M generates *all* nodes which are the leaves of a finite subtree whose root is n itself.

In order to start the process of node generation, we assume that the root node of the tree T (that is, the empty sequence ε) has already been generated by M before it starts its simulation of the nondeterministic Turing Machine N.

We leave to the reader the proof that indeed M simulates N iff it generates the sequence of numbers in any order whose corresponding node generation complies with Property (A) above.

Remark 6. The property that nondeterministic Turing Machines are equivalent to deterministic Turing Machines should be compared with the analogous property holding for Finite Automata. The expert reader may also recall that this is *not* the case for Pushdown Automata, that is, nondeterministic Pushdown Automata [13] accept a class of languages which is strictly larger than the class accepted by deterministic Pushdown Automata. For Linear Bounded Automata [13] the problem of whether or not the nondeterministic ones accept a class of languages which is strictly larger than the class accepted by deterministic Linear Bounded Automata is an open problem.

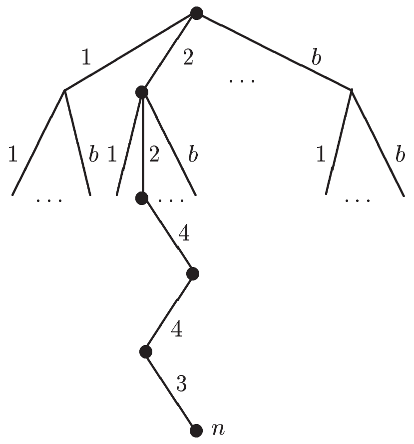


Fig. 9. The tree of choice-sequences. The node n corresponds to the sequence: 2 2 4 4 3.

5 Restrictions to Turing Machines Which Do Not Diminish Their Power

In this section we study some restrictions to the Turing Machines which do *not* diminish their power. We study these restrictions because they will help us to relate the Turing computable functions and the functions which are computable according to other models of computation, which we will introduce in Section 9.

5.1 One Tape = One Tape + Read-Only Input Tape = Two Stacks

Definition 11. A *deterministic two-stack machine* (or *two-stack machine*, for short) is an off-line deterministic TM with its read-only input tape and, instead of the work tape, it has two tapes which are used with the following restrictions (which make the tapes to behave like stacks):
(i) they are one-way infinite to the right, and
(ii) when the head moves left a printable blank is printed.

A deterministic two-stack machine is depicted in Figure 10.

Theorem 1. A deterministic TM can be simulated by a deterministic two-stack machine.

Proof. The tape configuration of Figure 11 is simulated by the two stacks with top elements a_n and a_{n+1}, respectively.
When the stack α_1 is empty the head is on the leftmost position. When the stack α_2 is empty the head is on the leftmost blank. The blank symbol never needs to be placed on the stack. □

Theorem 2. A deterministic two-stack machine $M2I$ with its read-only input tape can be simulated by a deterministic two-stack machine $M2$ without a read-only input tape.

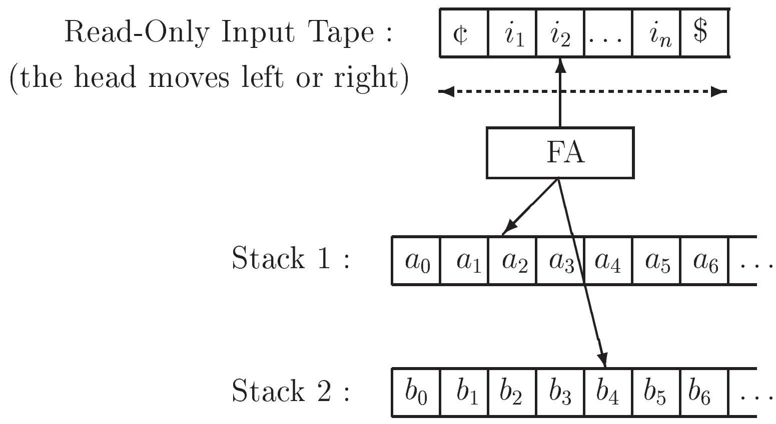


Fig. 10. A deterministic two-stack machine.

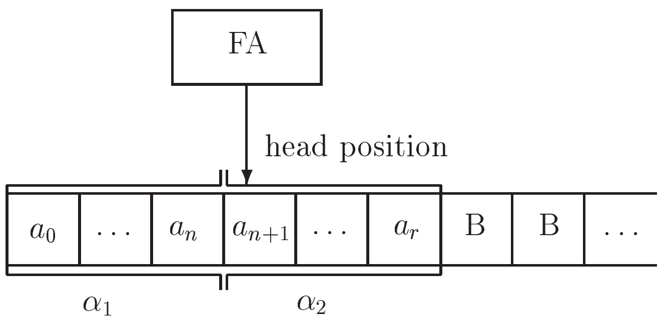


Fig. 11. Two stacks α_1 and α_2 which simulate a Turing Machine tape.

Proof. A two-stack deterministic machine $M2I$ with a read-only input tape can be simulated by an off-line Turing Machine M with a read-only input tape and one standard tape. The machine M is equivalent to a Turing Machine $M1$ with one standard tape only, and finally, by Theorem 1, $M1$ can be simulated by a deterministic two-stack machine $M2$.

An alternative proof is as follows. Consider, as above, the off-line Turing Machine M with input tape I and standard tape T, which is equivalent to $M2I$. Then the Turing Machine M can be simulated by a Turing Machine M_{tt} with two-track tape T_{tt}. Indeed, M_{tt} simulates M by considering both:
(i) the input symbol read by M (marked by the head position $X1$) on the leftmost, read-only part of its tape T_{tt} between the cells $\left|\begin{matrix}\text{¢}\\ B\end{matrix}\right|$ and $\left|\begin{matrix}\$\\ B\end{matrix}\right|$, and
(ii) the tape symbol read by M (marked by the head position $X2$) on the part of its tape T_{tt} to the right of the cell $\left|\begin{matrix}\$\\ B\end{matrix}\right|$ (see Figure 12). Finally, the Turing Machine M_{tt} with tape T_{tt} can be simulated by a two-stack machine. □

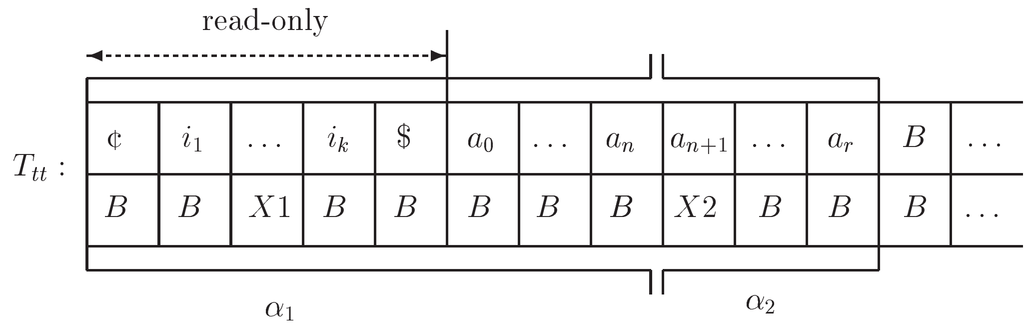


Fig. 12. Two stacks which simulate a tape and a read-only input tape. $X1$ and $X2$ show the head position on the input tape and the standard tape, respectively.

5.2 Two Stacks = Four Counters

Let us introduce the notion of a counter machine with n counters, which is depicted in Figure 13.

Definition 12. A *counter machine* with n counters, with $n > 0$, is a deterministic finite automaton with a read-only input tape and n stacks. Each stack which is said to be a *counter*, has two symbols: Z_0 and Δ (printable blank) only. The following conditions are satisfied: (i) Z_0 is initially written on the bottom of the stack, (ii) on the bottom cell only Z_0 can be written, (iii) on the other cells of the stack only Δ can be written, and (iv) one can test whether or not a head is reading Z_0.

Thus, each counter of a counter machine is a device for storing an integer, say k, and the available operations are: (i) addition of 1, (ii) subtraction by 1, and (iii) test for 0.

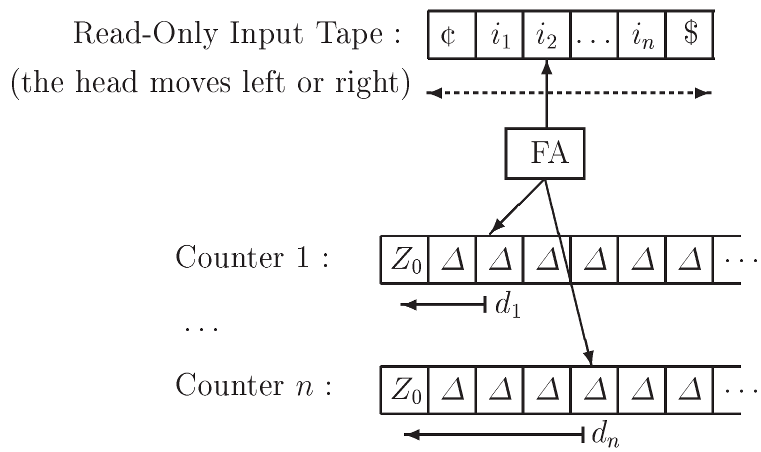


Fig. 13. A Counter Machine with n counters. As usual, the finite automaton FA is deterministic.

Actually, stacks of the form described in the above definition are called *iterated counters* (see, for instance, [11]), rather than counters. In [11] the name counter is reserved to stacks where one can perform (i) addition of 1 and (ii) subtraction by 1 only.

Theorem 3. A Turing Machine M can be simulated by a counter machine with four counters.

Proof. We first show how to simulate a stack by two counters. Suppose that the stack has $k-1$ tape symbols. Thus, each stack configuration can be encoded into a number written in base k. For instance, the tape symbols are a, b, and c, can encoded by the numbers 1, 2, and 3, respectively. Then the stack configuration $accb$ with top symbol b, can be encoded by the integer 1332 in base 4:

$$1\times 4^3 + 3\times 4^2 + 3\times 4^1 + \underset{\uparrow}{2}\times 4^0$$

which is 126 (in base 10). The symbol $\uparrow$ indicates the top of the stack which is pointing to the element 2.

Notice that we should use the base k and avoid the use of the digit 0 because otherwise we will identify two different stack configurations which differ by leading 0's only. For instance, if the symbols a, b, and c are encoded by 0, 1, and 2, respectively, and we use base 3, then the tape configurations cb and acb are encoded by the number 7 (in base 10). Indeed, cb is encoded by $2\times 3^1 + 1\times 3^0$ and acb by $0\times 3^2 + 2\times 3^1 + 1\times 3^0$.

Thus, (i) the operation '*push x*' on the stack is simulated by the multiplication by k and the addition of $H(x)$, where $H(x)$ is the number encoding the tape symbol x, (ii) the operation '*pop*' is simulated by the integer division by k (discarding the remainder), and (iii) the operation '*top*' is simulated by copying the number encoding the current configuration, say j, into the second tape while computing the top symbol as the value of j modulo k.

The above operations can be performed as follows:

(i) addition of n on a counter can be performed by moving its head n cells to the right,

(ii) multiplication by k of a number on the counter $C1$ with the result on the counter $C2$, can be performed by going left one cell on $C1$ while going right (starting from the leftmost cell) k cells on $C2$ until $C1$ reaches the bottom cell,
(iii) in an analogous way, division by k can be done by moving left k cells on a counter while the other goes right one cell (care should be taken, because if it is impossible to go left by k cells, then the head of the other counter should not move to the right), and
(iv) while going left on one counter, say $C1$, by 1 cell, we go right on the other counter $C2$ by 1 cell and at the same time in the states of the finite automaton we move by 1 state in a circle of k states to recall j modulo k. At the end, when the copying operation is finished, the state reached will tell us the value of j modulo k which can be stored in $C1$. □

5.3 Four Counters = Two Counters

Theorem 4. A Turing Machine M can be simulated by a counter machine with two counters.

Proof. By Theorem 3 it is enough to show how four counters can be simulated by two counters, call them $D1$ and $D2$. We encode the four values of the counters, say p, q, r, s by the number $2^p\ 3^q\ 5^r\ 7^s$ to be placed on the counter $D1$. Obviously, two different 4-tuples are encoded by two different numbers. Any other choice of four prime numbers, instead of 2, 3, 5, and 7, is valid.

We need to show: (a) how to increment a counter, (b) how to decrement a counter, and (c) how to test whether or not a counter holds 0. Let $C(1) = 2$, $C(2) = 3$, $C(3) = 5$, and $C(4) = 7$. Let us assume that the given counter configuration p, q, r, s is encoded in the counter $D1$.
Point (a). To increment the i-th counter by 1 we multiply the counter $D1$ by $C(i)$. This is done by going one cell to the left in $D1$ while going $C(i)$ cells to the right on $D2$, starting from the leftmost cell. The result will be on the counter $D2$, and if we want the result on $D1$ we have then to copy $D2$ onto $D1$.
Point (b). To decrement the i-th counter by 1 we perform an integer-division of the counter $D1$ by $C(i)$. This is done analogously to what is done at Point (a).
Point (c). To determine whether or not p or q or r or s is 0 we first compute $2^p\ 3^q\ 5^r\ 7^s$ modulo 2, as specified in the proof of Theorem 3 (recall that by that operation the value of $2^p\ 3^q\ 5^r\ 7^s$ is not destroyed) and it is 0 iff $p \neq 0$ (because $2^p\ 3^q\ 5^r\ 7^s$ is even iff $p \neq 0$). If $p = 0$ then the finite control enters a suitable state, say α_2 (see Figure 14).
We may also compute $2^p\ 3^q\ 5^r\ 7^s$ modulo 3, modulo 5, and modulo 7 to check whether or not $q = 0$, or $r = 0$, or $s = 0$, respectively. Clearly this computation can be performed by the finite control of a counter machine. If none of the numbers p, q, r, and s is 0 then the finite control of the counter machine enters the state ε (see Figure 14). In that figure the states α_1, β_1, γ_1, δ_1, and the grey states are the states visited while computing the value of $2^p\ 3^q\ 5^r\ 7^s$ modulo 2, 3, 5, and 7. This computation can easily be extended to the case when the test should return a different answer according to which of the 16 combinations of the four values of p, q, r, and s (each of these values can be either 0 or different from 0). In this case the finite control should enter one of 16 possible ‘answer states’. In Figure 14 we have depicted some of the states of the finite control for testing the values of p, q, r,

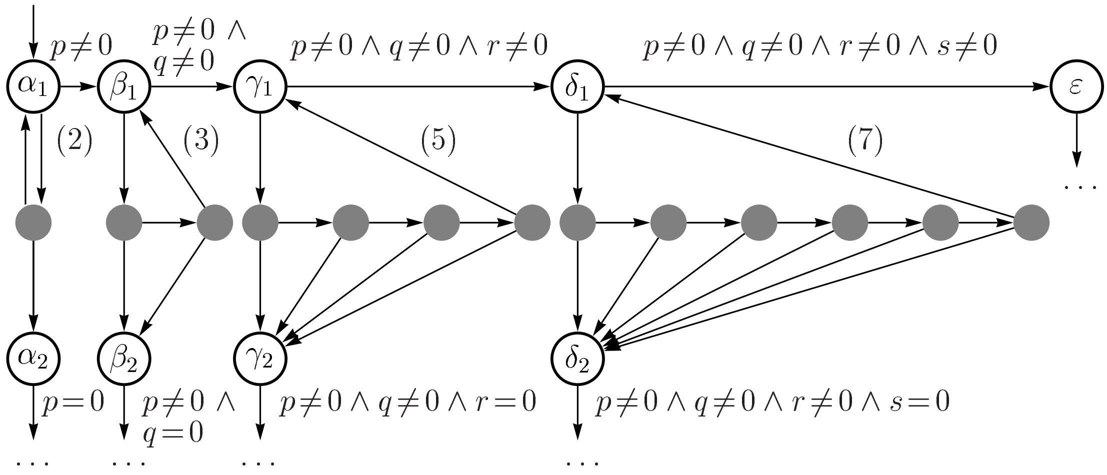


Fig. 14. The simulation of four counters by two counters. (i) In α_2: $p = 0$. (ii) In β_2: $p \neq 0 \wedge q = 0$. (iii) In γ_2: $p \neq 0 \wedge q \neq 0 \wedge r = 0$. (iv) In δ_2: $p \neq 0 \wedge q \neq 0 \wedge r \neq 0 \wedge s = 0$. (iv) In ε: $p \neq 0 \wedge q \neq 0 \wedge r \neq 0 \wedge s \neq 0$.

and s: they correspond to the tree of Figure 15. The 16 answer states would be the leaves of the tree of Figure 15 when the tree is completed by constructing: (i) a subtree rooted in γ_2 as the one rooted in δ_1, (ii) a subtree rooted in β_2 as the one rooted in γ_1, and (iii) a subtree rooted in α_2 as the one rooted in β_1.

The position of the reading head on the input tape of the two-counter machine is the same as the position of the four-counter machine.

Finally, we have to show how the two-counter machine can choose the move corresponding to the move of the four-counter machine. Any move of the four-counter machine depends on: (i) its internal state, and (ii) the position of the heads of the four counters, which can be one of the following sixteen configurations: (1) all four heads pointing to Z_0, (2-5) three heads pointing to Z_0 and one pointing to Δ, (6-11) two heads pointing to Z_0 and two pointing to Δ, (12-15) one head pointing to Z_0 and three pointing to Δ, (16) all four heads pointing to Δ.

The information about the internal state of the four-counter machine and the position of the heads of the four counters (which can be determined as we have seen at Point (c) above) can be stored in the internal states of the two-counter machine. In order to store the above information, the internal states of the two-counter machine may be very large. Indeed, in general, they should replicate the states of the four-counter machine for each of the sixteen configurations of the positions of the heads of the four counters.

Then, the two-counter machine can make the move corresponding to the one of the four-counter machine by suitably modifying the content of its two counters and changing its internal state. □

5.4 Turing Machines with Three or Two States Only

We state without proof the following result.

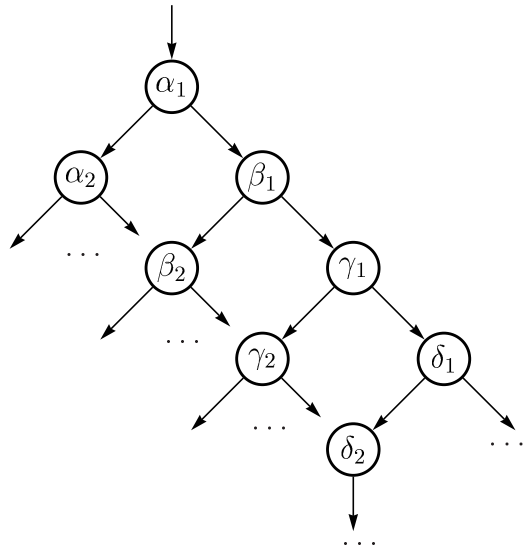


Fig. 15. A decision tree for testing the values of the four counters.

Theorem 5. [Shannon 56][29]. Given an alphabet Σ and a language $R \subseteq \Sigma^*$ such that $R = L(M)$ for some Turing Machine M then there exists a Turing Machine $M3$ and a tape alphabet $\Sigma 3 \supseteq \Sigma$ such that $R = L(M3)$ and $M3$ has three states.

If one assumes that the acceptance of a word in Σ^* is by halting (and not by entering a final state, as we have stipulated above), then we can reduce the number of states from 3 to 2. (Actually, this is what Shannon originally proved.)

5.5 Turing Machines with Three or Two Tape Symbols Only

Theorem 6. Given a language $R \subseteq \{0, 1\}^*$ such that $R = L(M)$ for some Turing Machine M then there exists a Turing Machine $M3$ which uses the tape alphabet $\{0, 1, B\}$, such that $R = L(M3)$.

Proof. Every tape symbol for the machine M will be encoded by a sequence of 0's and 1's of length k, where k is known when M is given. Let us assume that $M3$ is reading the leftmost cell of the k cells which encode the symbol read by M. $M3$ simulates the move of M by first reading k symbols to the right. If M reads a blank symbol then $M3$ should first write the code for the blank symbol on its tape (using k symbols to the right of the current position). The machine $M3$ begins the simulation of the machine M by first computing the encoding $en(w)$ of the word w which is the input of M. That encoding is computed starting from the leftmost symbol of w. During that computation the symbols of w to the right of the symbol, say s, which will be encoded, are shifted to the right by $k-1$ cells to create k free cells for the code of the symbol s itself.

In contrast to what is done for the input the encoding of the blank symbols is not done initially, because there are infinite blanks. The encoding of one blank symbol is done on demand, one blank at a time when it is required. $\square$

Corollary 1. Given a language $R \subseteq \Sigma^*$ for any finite input alphabet Σ such that $R = L(M)$ for some Turing Machine M then there exists an off-line Turing Machine $M3$ with the input tape and a standard tape whose alphabet is $\{0, 1, B\}$ such that $R = L(M3)$.

Theorem 7. [Wang 57] Given a language $R \subseteq \Sigma^*$ for any finite input alphabet Σ such that $R = L(M)$ for some Turing Machine M, there exists an off-line Turing Machine $M2$ with: (i) an read-only input tape whose alphabet is Σ, and (ii) a standard tape whose alphabet is $\{0, 1\}$, where 0 plays the role of blank (thus, the blank symbol B does not exist in the tape alphabet), such that $R = L(M2)$ and $M2$ cannot write a 0 over a 1 (that is, $M2$ cannot erase symbols: $M2$ has a non-printable blank symbol 0).

Proof. The simulation of M by $M2$ proceeds by rewriting along the tape the configurations of M. We need special symbols (written as strings of 0's and 1's) to encode the beginning and the end of each configuration, the position of the head, and some suitable markers to guide the process of copying, with the necessary changes, the old configuration into the new one. □

6 Turing Machines as Language Generators

Let us consider a fixed alphabet Σ. A Turing Machine can be viewed as a mechanism for generating languages over Σ^* as follows. We consider a two-tape TM M. The first tape is one-way infinite and it is used by the TM M as usual. The second tape is an *output tape*, that is, it is a write-only tape whose head moves to the right only. On this second tape the machine M prints, one after the other, separated by a special symbol, say #, the words of the language it generates.

The language $L \subseteq \Sigma^*$ generated by a TM M will be denoted by $G(M)$. If $G(M) \subseteq \Sigma^*$ is infinite then M does not halt.

Notice that we do not require that on the output tape the words of $G(M)$ are generated either (i) only once, or (ii) in any fixed order. In particular, for any given word $w \in G(M)$ of length $|w|$ (≥ 0) and of the form: $x_1 x_2 \ldots x_{|w|}$, there exists an index $i \in N$ such that the output tape cells $c_i, c_{i+1}, \ldots, c_{|w|-1}$ contain the symbols $x_1, x_2, \ldots, x_{|w|}$, respectively.

Theorem 8. For any given Turing Machine M and language R, $R = G(M)$ iff there exists a Turing Machine $M1$ such that $R = L(M1)$, that is, a language is generated by a Turing Machine M iff there exists a Turing Machine $M1$ which accepts it.

Proof. $(\Rightarrow)$ Let us assume that R is generated by the machine M. The machine $M1$ which accepts R can be constructed as follows. Given an input word $w \in \Sigma^*$, $M1$ works as M and when M writes a word w_1# on the output tape, $M1$ compares w_1 with w. If $w_1 = w$ then $M1$ accepts w, otherwise $M1$ continues the simulation of M. Obviously, $M1$ accepts w iff w is generated by M.

$(\Leftarrow)$ Given a Turing Machine $M1$ we have to construct a Turing Machine M which generates exactly the same language accepted by $M1$. Before showing this construction we need to define the notion of the i-th word in Σ^*, for any given finite alphabet Σ. (Actually this notion can be extended to the case where Σ is denumerable.) We can use the canonical order of the strings in Σ^*. In particular, if $\Sigma = \{0, 1\}$ and we assume that

$0 < 1$, then the i-th word of Σ^* is the i-th word occurring in the listing of the *canonical* order of the words of Σ^*, that is: $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 111, \ldots$. Thus, the i-th word of Σ^* is the i-th word in the listing Σ^0, Σ^1, Σ^2, ..., having listed the elements within each set Σ^i (for $i \geq 0$) in the usual *alphabetic* order from left to right. Obviously, one can devise a Turing Machine for computing the i-th word for any given $i \geq 0$. The machine M has three tapes plus an output tape. On the first tape it generates all natural numbers in increasing order starting from 0. For each such number, say n, M produces on the second tape a pair of numbers, say $\langle i, j \rangle$, using a dove-tailing bijection from $N \times N$ to N. Then given i, the Turing Machine M computes on the third tape the i-th word, say w_i, of Σ^*. At the end of this computation the third tape contains the word w_i. Then M behaves on its third tape as $M1$ does on its own tape but only for j steps (that is, j applications of the transition function), and if $M1$ accepts w_i using *no more than* j steps then M writes on its output tape w_i. □

Theorem 9. A language R is generated by a Turing Machine M, that is, $R = G(M)$, and each word in R is generated *once only*, iff R is accepted by some Turing Machine $M1$, that is, $R = L(M1)$.

Proof. The proof is like the one of Theorem 8, where, instead of 'no more than j', we say 'exactly j', because if a language is accepted by a TM there exists a deterministic Turing Machine which accepts it, and if a deterministic TM $M1$ accepts a word, say w, then $M1$ accepts w in exactly one way, that is, after a number of steps which depends on w only. □

Theorem 10. A language R is generated by a Turing Machine M, that is, $R = G(M)$, and it is generated in the *canonical order* iff there exists a Turing Machine $M1$ which accepts R, that is, $R = L(M1)$, such that for all $w \in \Sigma^*$, $M1$ stops after a finite number of moves (possibly depending on w).

Proof. ($\Rightarrow$) The machine $M1$ to be constructed, simulates M until: either (i) M halts without generating w, in which case $M1$ rejects w, or (ii) w is generated by M, in which case $M1$ accepts w, or else (iii) a word longer than w is generated by M (and this means that w will not be generated in the future by M, being the generation done by M in canonical order), in which case $M1$ rejects w.

($\Leftarrow$) Given the language R accepted by the machine $M1$, that is, $R = L(M1)$, we construct the machine M which generates R in the canonical order, as follows. If R is finite M looks at the elements of R and rearrange them in the canonical order and output them. If R is infinite, the language itself is given via the Turing Machine $M1$, and the machine M to be constructed, works as follows. M computes each word w in Σ^* in the canonical order and for each of them, M (i) simulates $M1$ and (ii) generates the word w of R iff $M1$ accepts w. Since for every input word w in Σ^*, $M1$ accepts or rejects w after a finite number of moves, we have that for any given word w in R, M generates w after a finite number of moves. It is immediate to see that the machine M indeed generates the language R in the canonical order. □

7 Turing Computable Functions and Type 0 Languages

Theorem 11. For any language $R \subseteq \Sigma^*$ if R is generated by the type 0 grammar $G = \langle T, V, P, S \rangle$, where T is the set of terminal symbols, V is the set of non-terminal symbols, P is the finite set of productions, and S is the axiom, then there exists a TM M such that $L(M) = R$.

Proof. Given R we construct a nondeterministic TM M with two tapes. On the first tape M has the word w to be accepted iff $w \in R$. On the second tape M starts from the tape configuration which is given by the axiom S only, and then (i) in a nondeterministic way M chooses a production of the grammar G, say $\alpha \to \beta$, and a position of the string, say σ, on the second tape, and (ii) M checks whether or not the string α is a substring of σ and if this is the case then M transforms σ so to realize the production $\alpha \to \beta$. During this transformation the Turing Machine M applies the shifting-over technique either to the right if $|\alpha| < |\beta|$, or to the left if $|\alpha| > |\beta|$. Then M checks whether or not the string produced on the second tape is equal to w which is kept on the first tape. Now, recalling that the notion of nondeterminism we have, is the angelic one, it is easy to see that $w \in R$ iff $w \in L(M)$. □

Theorem 12. For any language $R \subseteq \Sigma^*$ such that there exists a TM M such that $L(M) = R$ then there exists a type 0 grammar $G = \langle \Sigma, V, P, A_1 \rangle$, where Σ is the set of terminal symbols, V is the set of non-terminal symbols, P is the finite set of productions, and A_1 is the axiom, such that R is the language generated by G.

Proof. Given the TM M and a word $w \in \Sigma^*$, we construct a type 0 grammar G which first makes two copies of w and then simulates the behaviour of M on one copy. If M accepts w then G generates w, otherwise G does not generate w. The detailed construction of G is as follows.
Let $M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$. The productions of G are the following ones, where the symbols of the form of pairs $[-,-]$ are elements of the set V of the non-terminal symbols:
1. $A_1 \to q_0\, A_2$

These productions nondeterministically generate two copies of w:

2. $A_2 \to [a,a]\, A_2$ for each $a \in \Sigma$

These productions generate all tape cells necessary for simulating the computation of the Turing Machine M:

3. $A_2 \to [\varepsilon, B]\, A_2 \mid [\varepsilon, B]$

These productions simulate the moves to the right:

4. $q\,[a,X] \to [a,Y]\,p$
   for each $a \in S \cup \{\varepsilon\}$,
   for each $p, q \in Q$, and
   for each $X, Y \in \Gamma$ such that $\delta(q,X) = (p,Y,R)$

These productions simulate the moves to the left:

5. $[b,Z]\,q\,[a,X] \to p\,[b,Z]\,[a,Y]$
   for each $a \in \Sigma \cup \{\varepsilon\}$,
   for each $p, q \in Q$, and
   for each $X, Y, Z \in \Gamma$ such that $\delta(q,X) = (p,Y,L)$

When a final state q is reached, these productions propagate the state q to the left and to the right, and generate the word w, making q to disappear when all the terminal symbols of w have being generated:

$$\left.\begin{array}{l}6.\ \ [a,X]\,q \to q\,a\,q \\ \quad\ q\,[a,X] \to q\,a\,q \\ \quad\ q \to \varepsilon\end{array}\right\}\ \text{for each } a \in \Sigma \cup \{\varepsilon\},\ q \in F, \text{ and } X \in \Gamma$$

We will not formally prove that these productions simulate the behaviour of M, that is, for any $w \in \Sigma^*$, w is generated by the grammar G iff $w \in L(M)$.

The following observations should be sufficient:
(i) the first components of the nonterminals $[-,-]$ are never touched by the productions so that the given word w is kept unchanged,
(ii) never a non-terminal $[-,-]$ is made to be a terminal if a final state q is not encountered first,
(iii) if the acceptance of a word w requires at most a finite number of tape cells, say k, then the initial configuration of the tape for the word $w = a_1a_2 \ldots a_n$ is simulated by:

$$A_1 \ \to^* \ q_0[a_1,a_1][a_2,a_2]\ldots[a_n,a_n][\varepsilon,B][\varepsilon,B]\ldots[\varepsilon,B]$$

where k is the number of non-terminals on the right hand side of the production. □

8 Multitape Turing Machines and Random Access Memory Machines

A RAM (*Random Access Machine*) is an abstract model of computation which consists of an infinite number of memory cells (which are indexed by the numbers $0, 1, \ldots$), each of which can hold an integer, and a finite number of arithmetic registers, capable of storing and manipulating integers according to some given set of elementary instructions such as: *Store*, *Load*, *Add*, *Multiply*, *Jump*, *Conditional Jump* (that is, jump if the content of some given register is 0), and *Stop*.

Definition 13. A function f from N to N is said to be *RAM computable* iff starting from n in memory cell 0 we get the value of $f(n)$ in memory cell 0 when executing the instruction *Stop*.

As usual, integers may encode instructions. The instructions, once interpreted, modify the content of the memory cells and/or the registers.

It is not difficult to convince ourselves that a RAM can simulate any given Turing Machine or digital computer.

Now we show that, provided that the elementary instructions can be simulated by a TM (and this is left to the reader as an exercise, by using the techniques for the construction of Turing Machines that we have presented in previous sections), a RAM R can be simulated by a TM M. (In what follows, for simplicity, we will not distinguish between numbers and their encoding representations on the four tapes in use.) We assume that R has k registers.

Let us consider the following TM M with four tapes:

memory tape (n cells):	$\#\,0 * v_0\,\#\,1 * v_1\,\#\ldots\#\,i * v_i\,\#\ldots\#\,n * v_n\,\#\,B\,B\ldots$
register tape (k registers):	$\#\,0 * v_0\,\#\,1 * v_1\,\#\ldots\#\,k * v_k\,\#\,B\,B\ldots$
program counter:	$\#\,i\,\#$ (i is a memory address)
memory address register:	$\#\,j\,\#$ (j is a memory address)

If the program counter holds the value i then M looks for the i-th memory cell in the memory tape. If M does not find the string $\# \, i * v_i \, \#$ (because in the scan from the left of the memory tape it finds a B before $\# \, i * v_i \, \#$), then M stops.

Otherwise, M having found the string $\# \, i * v_i \, \#$, takes v_i and interprets it as the code of an instruction, say 'ADD j, 2' (that is, add to the register 2 the value in the memory cell whose address is j). Then M increases by 1 the program counter, looks on the memory tape for the cell whose address is j, and looks on the register tape for the register 2.

If M finds the memory cell with address j then it performs the elementary instruction of adding the number v_j to the register 2. If M does not find the memory cell with address j we may assume that the j-th cell contains 0, thus nothing should be done to the register 2. (In this case we could also assume that an error occurs and both M and R stop.) After simulating the instruction 'ADD j, 2', the machine M continues the simulation by performing the next instruction because the program counter has been updated.

As a consequence of the fact that Turing Machines can simulate RAM's and vice versa we have that the set of Turing computable functions is the same as the set of RAM computable functions.

9 Other Models of Computation

In this section we consider various other models of computation. They are all equivalent to Turing Machines as we will indicate below.

9.1 Post Machines

A *Post Machine* over the alphabet $\Sigma = \{a, b\}$ with the auxiliary symbol #, can be viewed as a flowchart which operates on a variable x. This variable denotes a queue, represented as a list with the usual operations: *head*, *tail*, *cons*, *append* (denoted by the infix @), and the constant *nil*.

The operations on the queue x are:
- $inqueue(x, e)$, which given x and an element $e \in \{a, b, \#\}$, produces the new queue represented by the list $x@[e]$, and
- $dequeue(x, e)$, which given x, returns an error if x is the empty queue (represented by the list nil), and otherwise, it makes the new value of the variable x to be $tail(x)$, and the new value of the variable e to be $head(x)$.

The *inqueue* and *dequeue* operations are used in a restricted way, because the flowchart specifying a Post Machine, has the following statements only:

- one *start* statement (indegree 0, outdegree 1),

- 0 or more *accept* or *reject* statements (indegree 1, outdegree 0),

- some *assignment* statements (indegree 1, outdegree 1) each of which is of the form: $inqueue(x, a)$, or $inqueue(x, b)$, or $inqueue(x, \#)$,

- some *test* statements (indegree 1, outdegree 4) each of which returns a different exit point according to the cases: (i) $x = nil$, (ii) $head(x) = a$, (iii) $head(x) = b$, and (iv) $head(x) = \#$. A test statement on the variable x leaves x unchanged if $x = nil$, otherwise it transforms x into $tail(x)$.

The language $L \subseteq \Sigma^*$ accepted (or rejected) by a Post Machine is the set of words w such that if x is equal to w before the execution of the corresponding flowchart, then at the end of the execution, the Post Machine performs an *accept* statement (or a *reject* statement, respectively).

Notice that for some initial value of x a Post Machine may run forever, without performing any *accept* or *reject* statement.

By recalling the results of Section 5 where a one-way infinite tape of a Turing Machine can be simulated by two stacks, we can establish an equivalence between Post Machines and Turing Machines which preserves the accepted language, by showing that the queue x can be represented as the concatenation of two lists, say x_L and x_R, so that $x = x_L @ x_R$, which in turn, can be represented by two stacks, say s_L and s_R (see Figure 16).

Indeed, for each element $e \in \{a, b, \#\}$, $inqueue(x, e) = push(s_R, e)$, and $dequeue(x, e) = \langle pop(s_L), top(s_L)\rangle$. We have that the symbol # plays the role of the *blank* symbol on the tape of a Turing Machine.

Since the representation of x in terms of s_L and s_R is *not* unique, it may be the case that x is not empty while s_L is empty. Thus, in order to perform a *dequeue* operation, it may be necessary to transfer the elements of the stack s_R to the stack s_L, and this can be done by popping the elements of s_R and pushing them into s_L. (Notice that the queue order is preserved.)

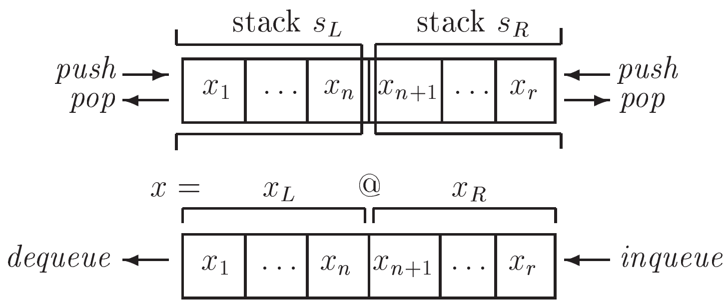


Fig. 16. A queue simulated by two stacks.

Now let us stipulate that a function f from N to N is computable by a Post Machine iff starting from $a^n b a^{f(n)}$ in the queue it eventually performs an *accept* statement.

As a consequence of the equivalence between Turing Machines and Post Machines which preserves the accepted language, we have that a function f from N to N is Turing computable iff f is computable by a Post Machine.

9.2 Post Systems

Given a set V of symbols, Post considered rewriting rules of the following kind to generate new strings of V^* from old strings of V^*:

$$g_0\, S_1\, g_1\, S_2\, g_2 \,\ldots S_m\, g_m \;\rightarrow\; h_0\, S_{j_1}\, h_1\, S_{j_2}\, h_2 \ldots\, S_{j_n}\, h_n \qquad (\dagger)$$

where:

- $g_0, g_1, \ldots, g_m, h_0, h_1, \ldots, h_n$ are (possibly empty) strings of V^* (they are called the *fixed strings*),
- $S_1, S_2, \ldots, S_m$ are distinct symbols not in V, called *variables*, ranging over strings in V^*,
- each variable in the list $\langle S_{j_1}, S_{j_2}, \ldots, S_{j_n}\rangle$ is taken from the set $\{S_1, S_2, \ldots, S_m\}$ and need not be distinct from the other variables of the list $\langle S_{j_1}, S_{j_2}, \ldots, S_{j_n}\rangle$.

A rule of the form (†) can be used for defining a binary relation $\rightarrow$ in $V^* \times V^*$ as follows.

Given s in V^* we match s with the whole left hand side of (†) and we get a substitution ϑ of the variables $S_1, S_2, \ldots, S_m$, each S_i, with $1 \leq i \leq m$, being mapped into a string of V^*. We then derive the rewriting:

$s \rightarrow (h_0\, S_{j_1}\, h_1\, S_{j_2}\, h_2 \ldots S_{j_n} h_n)\vartheta$.

For instance, given $V = \{r, s\}$ and the rule: $r\, S_1\, s\, S_2 \;\rightarrow\; S_2\, r\, S_2\, r$, we get the following rewritings of strings in V^*:

$r\,s\,r \;\rightarrow\; r\,r\,r\,r$ where $\vartheta = \{S_1/\varepsilon,\ S_2/r\}$,
$r\,s\,r\,s \;\rightarrow\; r\,r$ where $\vartheta = \{S_1/s\,r,\ S_2/\varepsilon\}$,
$r\,s\,r\,s \;\rightarrow\; r\,s\,r\,r\,s\,r$ where $\vartheta = \{S_1/\varepsilon,\ S_2/r\,s\}$, and
$s\,s\,r$ cannot be rewritten because it does not match $r\, S_1\, s\, S_2$.

We may also define, as usual, the reflexive and transitive closure of $\rightarrow$, denoted by $\rightarrow^*$.

Now we give the formal definition of a Post System. A Post System G consists of:
- a finite alphabet Σ,
- a finite set T of auxiliary symbols,
- a finite subset A of Σ^*, called the set of *axioms*, and
- a finite set of *productions* of the form (†), where the fixed strings are from the set V^*, where $V = \Sigma \cup T$.

The splitting of the alphabet V used for the fixed strings, into the sets Σ and T, will be clarified in an example below. Indeed, we will see that the auxiliary symbols in T may be used as separators between substrings of Σ^*. They play the role of separators between the adjacent cells of a Turing Machine tape, while the symbols in Σ are used for the contents of the cells. We say that a word w in Σ^* is generated by the Post System G iff there exists an axiom a_1 in the set A of axioms in G such that $a_1 \rightarrow^* w$, using the productions of G.

A Post System for producing the strings in Σ^* with $\Sigma = \{a, b\}$, which are palindromes, is as follows. $T = \emptyset$. The axioms are: ε, a, b. The productions are: $S_1 \rightarrow a\, S_1\, a$, and $S_1 \rightarrow b\, S_1\, b$.

A Post System for producing the valid addition equalities, that is, strings of the form: $1^m + 1^n = 1^{m+n}$, is as follows.

$\Sigma = \{1, +, =\}$. $T = \emptyset$. The axiom is $1 + 1 = 11$. The productions are:

$S_1 + S_2 = S_3 \;\rightarrow\; 1\, S_1 + S_2 = 1\, S_3$

$S_1 + S_2 = S_3 \;\rightarrow\; S_1 + 1\, S_2 = 1\, S_3$

For instance, we get: $1 + 1 = 11 \;\rightarrow\; 11 + 1 = 111 \;\rightarrow\; 11 + 11 = 1111$.

A Post System for producing binary representations of positive natural numbers, given in unary notation, is as follows.

$\Sigma = \{1, 0, =\}$. $T = \{¢, \$, \#\}$. The axioms are the unary representations of the positive natural numbers, that is, A is $\{1, 11, 111, \ldots\}$. The productions are the following ones.

The following production p1 is for making two copies of the input number S_1.

p1. $S_1 \rightarrow S_1 ¢ \$ S_1 \#$

The following production p2 is for dividing by 2 the number between \$ and #, and placing the result between ¢ and \$.

p2. $S_1 ¢ S_3 \$ 1 1 S_4 \# S_2 \rightarrow S_1 ¢ S_3 1 \$ S_4 \# S_2$

The following productions p3 and p4 are for placing 0 or 1 in the desired binary representation, depending on the fact that the given number is even (that is, no 1 is left between \$ and #) or odd (that is, one 1 is left between \$ and #). The substring between \$ and # is then initialized using the quotient S_3 of the previous division, for performing a subsequent division by 2.

p3. $S_1 ¢ S_3 \$ \# S_2 \rightarrow S_1 ¢ \$ S_3 \# 0 S_2$

p4. $S_1 ¢ S_3 \$ 1 \# S_2 \rightarrow S_1 ¢ \$ S_3 \# 1 S_2$

The following production p5 is for writing the result of the computation as desired.

p5. $S_1 ¢ \$ \# S_2 \rightarrow S_1 = S_2$.

For instance, we have:

111111	→ (by p1)	111111¢\$111111#	→ (by p2)	111111¢1\$1111#
	→ (by p2)	111111¢11\$11#	→ (by p2)	111111¢111\$#
	→ (by p3)	111111¢\$111#0	→ (by p2)	111111¢1\$1#0
	→ (by p4)	111111¢\$1#10	→ (by p4)	111111¢\$#110
	→ (by p5)	111111 = 110.		

From the above sequence of rewritings, we see that the auxiliary symbols ¢, \$, and # of our Post System are used as delimiters of four substrings: (i) the substring to the left of ¢, (ii) the substring to the right of ¢ and to the left of \$, (iii) the substring to the right of \$ and to the left of #, and (iv) the substring to the right of #.

We may encode the input n and the output $f(n)$ of a Turing computable function f from N to N, as the two sequences: 1^n and $1^{f(n)}$, separated by the symbol &. Using this encoding, we may say that the function f from N to N is *Post computable* iff there exists a Post System with alphabet $\Sigma = \{1, \&\}$ such that from the axiom & we can derive the string: $1^n \& 1^{f(n)}$ for any $n \in N$. We have that every Turing computable function from N to N is Post computable.

Vice versa, by encoding sequences of symbols of a given finite alphabet Σ into natural numbers, we have that every Post computable function is a Turing computable function.

Theorem 13. [Post Normal Form Theorem] There is not loss of computational power if one restricts the productions of a Post System to be of the form:

$g\, S \rightarrow S\, h$

where the fixed strings g and h are in $(\Sigma \cup T)^*$. Thus, one variable symbol S is sufficient.

Post Systems whose productions are of the form:

$$S_1\, g\, S_2 \rightarrow S_1\, h\, S_2 \qquad (\dagger\dagger)$$

provide an alternative presentation of Chomsky's formal grammars. Indeed, the set of terminal and non-terminal symbols of a given grammar is the set Σ of the corresponding Post System, and a given grammar production: $g \rightarrow h$ is simulated by the Post production (††).

9.3 Unlimited Register Machines

An *Unlimited Register Machine*, URM for short, can be viewed as a collection of an infinite number of registers, say $R_1, \ldots, R_n, \ldots$, each holding a natural number. The content of the register R_i will be denoted by r_i. A URM is associated with a finite sequence of instructions $P = \langle I_1, \ldots, I_p \rangle$, with $p \geq 1$, each of which is of the form:
(i) $Z(n)$, which makes r_n to be 0, or
(ii) $S(n)$, which increases r_n by 1, or
(iii) $Assign(m, n)$, which makes r_m to become equal to r_n (that is, $r_m := r_n$), or
(iv) $Jump(m, n, j)$, which does not change the content of the registers, and if $r_m = r_n$ then the next instruction to be executed is I_j, with $1 \leq j \leq p$, otherwise it is the instruction which follows the same instruction $Jump(m, n, j)$ in the given sequence P.

The sequence of instructions is also called program. A URM works by executing instructions, as usual, according to the given program P. The first instruction to be executed is I_1.

It is possible to define, in the usual way, the computation of a URM for a given program and an initial content of the registers. The computation of a URM stops iff there is no next instruction to be executed.

A URM M computes a function f from N^k to N in the sense that: $f(a_1, \ldots, a_k)$ is defined and its value is b iff starting from the values $a_1, \ldots, a_k$ in the registers $R_1, \ldots, R_k$, respectively, and the value 0 in all other registers, the computation of M eventually stops and we get b as the content of the register R_1.

One can show that the set of functions computed by URM's does not change if we do not allow *Assign* instructions. An important result about URM's is that the set of functions from from N^k to N computed by URM's, is the set of the Turing computable functions from N^k to N.

9.4 Markov Systems

A *Markov System* can be viewed as a particular kind of type 0 grammar. The idea of a Markov System is to make deterministic the nondeterministic process of replacing a substring by another substring in a type 0 grammar.

A Markov System is a quintuple $\langle \Sigma, V_N, R, R_1, R_2 \rangle$, where Σ is the set of terminal symbols, and V_N is the set of non-terminal symbols. Let V be the set $V_N \cup \Sigma$. R is the finite *sequence* $r_1, \ldots, r_n$ of rules, that is, ordered pairs of words in $V \times V$, made out of the two disjoint sequences R_1 and R_2.

We define a rewriting relation $\rightarrow$ on $V^* \times V^*$ as follows: $u \rightarrow v$ iff given the string u, we get the string v by rewriting (as usually done with grammar productions) a substring of u according to the following two constraints:
(i) we have to use the *leftmost* rule in R (that is, the rule with smallest subscript) among those which can be applied, and
(ii) we have to apply that rule at the *leftmost* occurrence (if it can be applied in more than one substring of u).

We also define a *restricted transitive closure* of $\rightarrow$, denoted by $\rightarrow^{\circledast}$, as the usual transitive closure with the condition that the last rewriting step, and that step only, is performed by using a rule in R_2. A Markov System computes a function f from Σ^* to Σ^* in the sense that:

$f(u) = v$ iff $u \rightarrow^{\circledast} v$.

One can show that the set of functions from Σ^* to Σ^* computed by the Markov Systems is the set of the Turing computable functions Σ^* to Σ^*.

10 Church Thesis

In Sections 7 and 8 we have shown that there exists a bijection: (i) between Turing Machines and Type 0 grammars and (ii) between Turing Machines and Random Access Machines which preserves the computed functions. Actually, as we already mentioned, there exists such a bijection also between Turing Machines and each of the models of computation we have introduced in the previous section, that is: (i) Post Machines [19], (ii) Post Systems [5, page 61], (iii) Unlimited Register Machines, (iv) Markov Systems (see [5, page 65] and [18, page 263]), so that the set of computable functions from N to N is preserved.

Also Partial Recursive Functions which we will introduce in the following chapter, and other models of computations such as: (i) Lambda Calculus [4] and (ii) Combinatory Logic [4], define the same set of computable functions from N to N.

The fact that all these models of computations which have been introduced in the literature for capturing the informal notion of 'effective computability', rely on quite different intuitive ideas, and yet there exists a bijection between any two of them, is a strong evidence of the validity of the following statement, called *Church Thesis* or *Turing Thesis*:

Church Thesis. *Every function which, in intuitive terms, is 'effectively computable' is also Turing computable.*

Notice that it is not possible to formally prove Church thesis (and this is why it is called a thesis, rather than a theorem) because the notion of being 'effectively computable' is not given in a formal way.

Chapter 2

Partial Recursive Functions

11 Preliminary Definitions

Let D be a (not necessarily proper) subset of N^k for some $k \geq 0$.
A *partial function* f of arity k (that is, of k arguments) from N^k to N is a subset of $D \times N$, with $D \subseteq N^k$, such that for all x in D there exists a unique m in N such that $\langle x, m \rangle \in f$. For every $x \in D$ we say that f is *convergent* in x, or f is *defined* in x, or $f(x)$ is *convergent,* or $f(x)$ is *defined.*

The *domain* of a partial function f which is a subset of $D \times N$, such that for all x in D there exists a unique m in N such that $\langle x, m \rangle \in f$, is D. We also say that D is the subset of N^k where f is *defined.*

We say that a partial function f is *divergent* in x, or f is *undefined* in $x,$ or $f(x) =$ undefined, iff x does not belong to the domain of f.

A partial function of arity k is said to be *total* iff its domain is N^k.

The *codomain* of a partial function from N^k to N is N. The *range* of a partial function f from N^k to N is the set $\{m \mid \langle x, m \rangle \in f\}$. The element m of the range of f such that $\langle x, m \rangle \in f$, is also called the *value of f in x*. Instead of $\langle x, m \rangle \in f$, we also write $m = f(x)$.

Each partial function of arity 0, which is total, is identified with an elements of N. In formal terms, for each $m \in N$, we identify the function $\{\langle \langle \rangle, m \rangle\}$ with m. The domain of each partial function f of arity 0 from N^0 to N which is a total function, is N^0, that is, the singleton $\{\langle \rangle\}$.

There exists a unique partial function of arity 0 from N^0 to N which is *not* total. It is the function which is undefined in the argument $\langle \rangle$. Unless otherwise specified, when we will say ‘function’ we will mean ‘partial function’.

We write the application of the function f of arity k to the argument x as $f(x)$ or fx. If x is the k-tuple $\langle x_1, \ldots, x_k \rangle$ we also write $f(x)$ as $f(x_1, \ldots, x_k)$ or $f x_1 \ldots x_k$ or $f(z, x_k)$, where z is the $(k-1)$-tuple $\langle x_1, \ldots, x_{k-1} \rangle$.

We sometimes use extra parentheses, and we write an expression e as (e).

Below we will define the set PRF_k of the so called *partial recursive functions* of arity k, for any $k \geq 0$. It is a subset of the set of all partial functions of arity k.

The set PRF of all partial recursive functions is:

$\bigcup_{k \geq 0} \mathrm{PRF}_k$

By a total partial recursive function we mean a function in PRF which is total. Often in the literature a total partial recursive function is also called a total recursive function, or simply, a recursive function.

There exists a bijection, call it τ, between N^* and N (see below). Thus, there is a bijection between the set PRF of all partial recursive functions of zero or more arguments and the set PRF_1 of all partial recursive functions of one argument (that is, from N to N).

We will see below that: (i) this bijection τ and its inverse τ^{-1} are total partial recursive functions (actually, they are primitive recursive functions (see Definition 14)), and (ii) the class of the partial recursive functions is closed under composition.

In what follows we will consider also a class C of properties of partial recursive functions. In order to study such class C of properties, without loss of generality, we will restrict ourselves to partial recursive functions of arity 1, because for that class C we have that every partial recursive function $g(n)$ from N to N has a property P in C iff the partial recursive function, say $\widetilde{g}(x) = g(\tau(x))$ from N^k to N, with $k \geq 0$ and $\tau(x) = n$, has property P.

For denoting functions we will often use the so called *lambda notation*. This means that a function which given the argument n, returns the value of the expression e, is written as the *lambda term* $\lambda n.e$. In order to make it explicit that the argument n may occur in the expression e, we will also write $\lambda n.e[n]$, instead of $\lambda n.e$. We will allow ourselves to say the function e, instead of the function $\lambda n.e$, when the argument n is understood from the context.

Let us now introduce some elementary notions concerning the lambda notation which will be useful below.

Given a set K of constants with arity, with k ranging over K, and a set V of variables, with v ranging over V, a *lambda term* e is either

a constant k or

a variable v or

an abstraction $\lambda v.e$ or

an application $(e_1 e_2)$, also denoted $e_1(e_2)$ or simply $e_1 e_2$.

For every lambda term p and e, we define the notion of '*p occurs in e*' (or '*p is a subterm of e*' or '*e contains p*') as follows:
(i) p occurs in p, (ii) if p occurs in e_1 or e_2 then p occurs in $(e_1 e_2)$, and (iii) if p occurs in e or p is the variable v then p occurs in $\lambda v.e$.

An occurrence of a variable v in a term e is said to be *bound* iff it occurs in a subterm of e of the form $\lambda v.e_1$. An occurrence of a variable v in a term e is said to be *free* iff it is not bound. For instance, the leftmost occurrence of v in $v\,(\lambda v.(vx))$ is free, while the other two occurrences are bound.

Given a lambda term e the set of its *free variables*, denoted $FV(e)$ can be inductively defined as follows:

(constant)	$FV(k) = \emptyset$,
(variable)	$FV(v) = \{v\}$,
(abstraction)	$FV(\lambda v.e) = FV(e) - \{v\}$,
(application)	$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$.

The result of the application of the function $\lambda n.e[n]$ to the argument m is $e[m/n]$, where by $e[m/n]$ we mean the expression e where all *free* occurrences of n in e have been replaced by m. Thus, for instance, (i) $(\lambda n.n+1)5 = 5+1$, and (ii) $(\lambda n.(n+((\lambda n.n+1)3)))\,5 = (5+((\lambda n.n+1)3)) = 5+(3+1) = 9$.

Notice that the name of a bound variable is insignificant, that is, $\lambda x.e$ is the same as $\lambda y.e[y/x]$. Thus, for instance, $(\lambda n.(n+((\lambda n.n+1)3))) = (\lambda n.(n+((\lambda x.x+1)3)))$.

Functions of zero arguments returning the value of the expression e are written as $\lambda().e$ or simply e. For $r \geq 2$ we will write $\lambda x_1 \ldots x_r.e$ or $\lambda x_1, \ldots, x_r.e$ as an abbreviation for the lambda term $\lambda x_1.(\ldots(\lambda x_r.e)\ldots)$.

12 Primitive Recursive Functions

Definition 14. The set of the *primitive recursive functions* is the smallest set, call it PR, of functions, which includes:

(i) the *zero* functions z_k of k arguments, that is, $\lambda x_1 \ldots x_k.0$, for all $k \geq 0$,

(ii) the *successor* function s of one argument, that is, $\lambda x.x+1$, and

(iii) the i-th *projection* functions π_{ki} of k arguments, that is, $\lambda x_1 \ldots x_k.x_i$, for all $k \geq 1$ and all i such that $1 \leq i \leq k$, and it is closed under *composition* and *primitive recursion*, that is,

(iv) (*composition*) if the function h of $k \geq 1$ arguments is in PR and the k functions $g_1, \ldots, g_k$, each of them being of $m\,(\geq 0)$ arguments, are in PR, then also the following function f of m arguments

$$f(x_1, \ldots, x_m) = h(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$$

is in PR, and

(v) (*primitive recursion*) if a function h of $k+1$ arguments and a function g of $k-1$ arguments are in PR for some $k \geq 1$, then also the function f of k arguments such that:

$$f(0, x_2, \ldots, x_k) = g(x_2, \ldots, x_k)$$
$$f(x_1+1, x_2, \ldots, x_k) = h(x_1, x_2, \ldots, x_k, f(x_1, x_2, \ldots, x_k))$$

is in PR.

Obviously, we have that each function in PR is a function from N^k to N for some $k \geq 0$. The *identity function* $\lambda x.x$ from N to N is primitive recursive, because it is the projection function π_{11}.

Lemma 1. All partial functions of arity k, for $k \geq 0$, which are constant functions, are primitive recursive, that is, for all $n \geq 0$ and $k \geq 0$, $\lambda x_1 \ldots x_k.n$ is primitive recursive.

Proof. By induction on the number k of arguments.
(Basis) For $k = 0$ we have that: (i) $\lambda().0$ is primitive recursive by Definition 14 (i), and (ii) for all $n \geq 0$, $(\lambda().n+1)() = s((\lambda().n)())$, where for any 0-ary function f we denote by $f()$ the application of f to the empty tuple of arguments. Thus, by Definition 14 (iv) we have that for all $n \geq 0$ if the function $\lambda().n$ is primitive recursive then also the function $\lambda().n+1$ is primitive recursive.
(Step) Let us consider the constant function $f_m = \lambda x_1 \ldots x_m.n$. Then the function $f_{m+1} = \lambda x_1 \ldots x_m x_{m+1}.n$ is defined as follows:

$$f_{m+1}(0, x_2, \ldots, x_{m+1}) = f_m(x_2, \ldots, x_{m+1})$$
$$f_{m+1}(x_1+1, x_2, \ldots, x_{m+1}) = \pi_{m+2\,m+2}(x, x_2, \ldots, x_{m+1}, f_{m+1}(x, x_2, \ldots, x_{m+1})). \qquad \square$$

Lemma 2. Primitive recursive functions are closed under: (i) addition of dummy arguments, (ii) substitution of constants for variables, (iii) identification of arguments, and (iv) permutation of arguments.

Proof. The reader may convince himself by looking at the following examples.
(i) Addition of dummy arguments. For instance, $\lambda xy.f(x,y)$ is obtained from $\lambda x.g(x)$ where for all x and y, $f(x,y) = g(x)$, by composing primitive functions because $f(x,y) = g(\pi_{21}(x,y))$.
(ii) Substitution of constants for variables. For instance, $\lambda x.g(x)$ is obtained from $\lambda xy.f(x,y)$ where for all x, $g(x) = f(x,3)$, by composing primitive functions because $g(x) = f(\pi_{11}(x), (\lambda x.3)(x))$.
(iii) Identification of arguments. For instance, $\lambda x.g(x)$ is obtained from $\lambda xy.f(x,y)$ where for all x, $g(x) = f(x,x)$, by composing primitive functions because $g(x) = f(\pi_{11}(x), \pi_{11}(x))$.
(iv) Permutation of arguments. For instance, $\lambda xy.g(x,y)$ is obtained from $\lambda xy.f(x,y)$ where for all x and y, $g(x,y) = f(y,x)$, by composing primitive functions because $g(x,y) = f(\pi_{22}(x,y), \pi_{21}(x,y))$. $\square$

When showing that a function is primitive recursive we will allow ourselves to make use of Lemma 3 and to adopt abbreviated notations.

For instance, we will show that the addition function is primitive recursive by writing the following equations:

$sum(0,y) = y$
$sum(x{+}1,y) = sum(x,y) + 1$ (by primitive recursion)

instead of

$sum(0,y) = (\lambda x.x)y$
$sum(x+1,y) = sp(x,y,sum(x,y))$ (by primitive recursion)

where

$sp(x,y,z) = (\lambda x.x{+}1)(\pi_{33}(x,y,z))$ (by composition)

If we apply Lemma 2 the Composition Schema (Definition 14.iv) may be generalized by allowing some of the g_i's to be functions of some, but *not all* the variables $x_1,\ldots,x_m$. Analogously, the Primitive Recursion Schema (Definition 14.v) may be generalized by allowing the function h to have as arguments only some of the $k{+}1$ arguments $x_1,x_2,\ldots,x_k$, $f(x_1,x_2,\ldots,x_k)$.

The functions defined by equations like the ones of Case (v) of Definition 14, which are called *recursive equations*, are assumed to be evaluated in the *call-by-value* mode, also called *inside-out and left-to-right* mode. For more details the reader may look at [25].

This means that the evaluation of an expression proceeds by:

(i) considering the *leftmost* subexpression, say u, among all *innermost* subexpressions which match a left-hand-side of a recursive equation, and

(ii) replacing u by $(R\vartheta)$, if u is equal to $(L\vartheta)$ for some recursive equation $L = R$ and substitution ϑ.

Actually, since all primitive recursive functions are total, the mode of evaluation of the recursive equations does not matter. However, it does matter in the case of the partial recursive functions (see Section 13) because of the presence of the minimalization operator which may lead to a non-terminating computation.

It can be shown that: (i) the functions defined by composition and primitive recursion are uniquely determined by their recursive equations (this result is stated in the so called *Recursion Theorem* which, for instance, can be found in [8]), and (ii) for all x in the domain of f we can compute by using the call-by-value mode of evaluation the unique element m of N such that $f(x) = m$.

It follows from Definition 14 that the set of primitive recursive functions is denumerable. Indeed, for the set PR we have the following domain equation [24, Chapter 4]:

$$\mathrm{PR} = N \ + \ \{1\} \ + \ N^{*} \times N + \mathrm{PR} \times \mathrm{PR}^{*} + \mathrm{PR} \times \mathrm{PR}$$

Each of the five summands of the right hand side of this equation corresponds to a case of Definition 14. It can be shown by induction that every primitive recursive function is total.
Almost all 'algorithmic functions' of ordinary mathematics are primitive recursive. Here is the proof for some of them. The proof is given by exhibiting the recursive equations of those functions. We leave to the reader to fill in the details and to show how these recursive equations can be replaced by suitable applications of the composition and primitive recursion schemata as indicated above for the function *sum*. We have:

1. (addition)

$sum(0, y) = y \qquad sum(x+1, y) = sum(x, y) + 1$ (by primitive recursion)

We will also write $sum(x, y)$ as $x+y$. Notice that we have overloaded the symbol $+$, because $x+1$ also denotes the unary successor function $\lambda x.s(x)$.

2. (predecessor: $x \neg 1 = \mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ x-1$)

$0 \neg 1 = 0 \qquad (x+1) \neg 1 = x$ (by primitive recursion)

It is a primitive recursive function because:
$0 \neg 1 = (\lambda().0)()$ and $(x+1) \neg 1 = \pi_{21}(x, x \neg 1)$.

3. (proper subtraction: $x \neg y = \mathit{if}\ y \geq x\ \mathit{then}\ 0\ \mathit{else}\ x-y$)

$x \neg 0 = x \qquad x \neg (y+1) = (x \neg y) \neg 1$ (by primitive recursion)

As often done in the literature we have overloaded the symbol $\neg$ because we have used it for proper subtraction and predecessor. If $x \geq y$ we will feel free to write $x - y$, instead of $x \neg y$. Indeed, if $x \geq y$ then $x \neg y = x - y$, where $-$ is the usual subtraction operation between natural numbers we learnt at school.

4. (multiplication)

$0 \times y = 0 \qquad (x+1) \times y = (x \times y) + y$ (by primitive recursion)

We will also write $x \times y$ as xy.

5. (exponentiation)

$x^{0} = 1 \qquad x^{y+1} = x^{y} x$ (by primitive recursion)

6. (factorial)

$0! = 1 \qquad (x+1)! = (x+1) + (x!)$ (by primitive recursion)

7. (sign: $sg(x) = \mathit{if}\ x=0\ \mathit{then}\ 0\ \mathit{else}\ 1$)

$sg(0) = 0 \qquad sg(x{+}1) = 1$ (by primitive recursion)

8. (negated sign: $nsg(x) = \mathit{if}\ x=0\ \mathit{then}\ 1\ \mathit{else}\ 0$)

$nsg(0) = 1 \qquad nsg(x{+}1) = 0$ (by primitive recursion)

9. (minimum between two natural numbers: min) (by composition)

$min(x,y) = x \neg (x \neg y)$

10. (maximum between two natural numbers: max) (by composition)

$max(x,y) = x + (y \neg x)$

11. (absolute difference: $|x-y| = max(x,y) \neg min(x,y)$)

$|x-y| = (x \neg y) + (y \neg x)$

We have that: $|x-y| = |y-x|$.

The following properties relate the functions sg, nsg, and the *if-then-else* construct. For any $a, b \in N$, we have that:

$$\begin{aligned} &\mathit{if}\ x{=}0\ \mathit{then}\ a\ \mathit{else}\ b\ =\ a\ nsg(x){+}b\ sg(x) \\ &\mathit{if}\ x{=}y\ \mathit{then}\ a\ \mathit{else}\ b\ =\ a\ nsg(|x-y|){+}b\ sg(|x-y|) \end{aligned}$$

In particular, for any a in N,

$$\begin{aligned} &\mathit{if}\ x{=}y\ \mathit{then}\ 0\ \mathit{else}\ a\ =\ \mathit{if}\ x{\neq}y\ \mathit{then}\ a\ \mathit{else}\ 0\ =\ a\ sg(|x-y|) \\ &\mathit{if}\ x{=}y\ \mathit{then}\ a\ \mathit{else}\ 0\ =\ \mathit{if}\ x{\neq}y\ \mathit{then}\ 0\ \mathit{else}\ a\ =\ a\ nsg(|x-y|) \end{aligned}$$

12. ($rem(x,y)$ = integer remainder when x is divided by y)

We assume that for any $x \in N$, $rem(x,0) = x$. Thus, rem is a total function from N^2 to N. We have that: for any $y \in N$,

$$\begin{aligned} rem(0,y) &= 0 \\ rem(x+1,y) &= \mathit{if}\ (rem(x,y){+}1) \neq y\ \mathit{then}\ rem(x,y){+}1\ \mathit{else}\ 0 \end{aligned}$$

Thus,

$$\begin{aligned} rem(0,y) &= 0 \\ rem(x+1,y) &= (rem(x,y){+}1)\ sg(|(rem(x,y){+}1) - y|) \end{aligned}$$

This last equation can be rewritten as follows:

$$\begin{aligned} &rem(x{+}1,y) = g(rem(x,y),y) \qquad \text{where} \\ &g(y,z) = (y{+}1)\ sg(|(y{+}1) - z|) \end{aligned}$$

and the function g is primitive recursive because it can be obtained by composition.

Notice that $rem(x,y) < y$ iff $y > 0$. In particular, $rem(x,y) < y$ does *not* hold if $y = 0$. Indeed, for any $x \in N$, we have that $rem(x,0) = x$ and $x \not< 0$.

13. ($quot(x,y)$ = integer quotient when x is divided by y)

We assume that for any $x \in N$, $quot(x,0) = x$. Thus, $quot$ is a total function from N^2 to N. We have that:

$$\begin{aligned} quot(0,y) &= 0 \\ quot(x+1,y) &= \mathit{if}\ (rem(x,y){+}1) \neq y\ \mathit{then}\ quot(x,y)\ \mathit{else}\ quot(x,y){+}1 \\ &= quot(x,y)\ +\ (\mathit{if}\ (rem(x,y){+}1) \neq y\ \mathit{then}\ 0\ \mathit{else}\ 1) \end{aligned}$$

Notice that, analogously to what happens in standard arithmetics, for all $x, y \in N$, we have that:

$x = y \times quot(x, y) + rem(x, y)$.

14. (divisibility: $div(x, y) = if\ x$ is divided by $y\ then\ 1\ else\ 0$)

We assume that x is divided by 0 iff $x = 0$, that is, $div(x, 0) = 1$ iff $x = 0$. Thus, div is a total function from N^2 to N. We have that:

$div(x, y) = nsg(rem(x, y))$

Here are some examples of applications of the functions rem, $quot$, and div.

(i) For the division $0/4$ we have: $rem(0, 4) = 0$ and $quot(0, 4) = 0$.
(ii) For the division $4/0$ we have: $rem(4, 0) = 4$ and $quot(4, 0) = 0$.
(iii) $div(0, x) = nsg(rem(0, x)) = 1$ for any $x \geq 0$.
(iv) $div(x, 0) = nsg(rem(x, 0)) = 0$ for $x \neq 0$.

A *predicate* P on N^k for $k \geq 0$, is a subset of N^k. If $x \in P$ we also write $P(x)$ and we say that $P(x)$ holds or $P(x)$ is true. Given a predicate P on N^k, its characteristic function, denoted f_P, is a function of arity k from N^k to N, defined as follows:

$f_P(x) = if\ x \in P\ then\ 1\ else\ 0$, for any $x \in N^k$.

Definition 15. A predicate is said to be *primitive recursive* iff its characteristic function is primitive recursive.

For instance, the predicates $x < y$, $eq(x, y)$ (also written as $x = y$), and $x > y$, for $x, y \in N$ are primitive recursive. Indeed, $f_{x<y} = sg(y \neg x)$, $f_{eq(x,y)} = nsg(|x - y|)$, and $f_{x>y} = sg(x \neg y)$.

Lemma 2 can easily be extended to primitive recursive predicates. We leave that extension to the reader.

Lemma 3. Primitive recursive predicates are closed under negation, (finite) disjunction, and (finite) conjunction.

Proof. (i) $f_{not\,P}(x) = 1 \neg f_P$, (ii) $f_{P\,or\,Q}(x) = sg(f_P(x) + f_Q(x))$, and
(iii) $f_{P\,and\,Q}(x) = f_P(x) \times f_Q(x)$. □

We have that the predicate $x \leq y$ is primitive recursive, because $x \leq y$ iff ($x < y$ or $eq(x, y)$).

Given a predicate P on N^{k+1} for $k \geq 0$, the *bounded existential quantification* of P is the predicate Q on N^{k+1} such that $Q(x, n)$ iff $\exists i \in [0, n] P(x, i)$.

Here and below, for any pair a and b of natural numbers, with $a \leq b$, $[a, b]$ denotes the subset $\{a, a+1, \ldots, b\}$ of N.

Analogously, for the *bounded universal quantification* by replacing $\forall$ for $\exists$.

Lemma 4. Primitive recursive predicates are closed under bounded existential quantification and bounded universal quantification.

Proof. If $f_P(x,n)$ is the characteristic function of the predicate $P(x,n)$ then the characteristic function $f_Q(x,n)$ of the predicate $Q(x,n) = \exists i \in [0,n].\, P(x,i)$ is defined by primitive recursion as follows:

$$f_Q(x,0) = f_P(x,0) \qquad f_Q(x,n{+}1) = f_Q(x,n) + f_P(x,n{+}1).$$

For the bounded universal quantification, it is enough to recall that Lemma 3 and the fact that $\forall i \in [0,n].\, P(x,i)$ is the same as $not(\exists i \in [0,n].\ not(P(x,i)))$. □

Lemma 5. (*Definition by cases*) Let us consider the set $\{\langle g_i(x), P_i(x)\rangle \mid 1 \leq i \leq r$ and $1 < r\}$, where for $1 \leq i \leq r$, $g_i(x)$ is a primitive recursive function from N^k to N and $P_i(x)$ is a primitive recursive predicate denoting a subset of N^k. Let us assume that for each x in N^k one and only one of the predicates $P_i(x)$, for $1 \leq i \leq r$, holds. Then the function f which satisfies the following equations:

$$\begin{aligned} f(x) &= g_1(x) \quad && \textit{if} \ \ P_1(x) \\ &= g_2(x) && \textit{if} \ \ P_2(x) \\ &\ \ \ldots \\ &= g_r(x) && \textit{if} \ \ P_r(x) \end{aligned}$$

is primitive recursive.

Proof. The case where $r = 1$ is obvious, because the definition of $f(x)$ corresponds to the equation $f(x) = g_1(x)$ *if* true, that is, $f(x) = g_1(x)$. For $r > 1$ we have the following. Let $c_i(x)$ be the characteristic function of $P_i(x)$ for $1 \leq i \leq r$. Then

$$f(x) = (\ldots(g_1(x)c_1(x) + g_2(x)c_2(x)) + \ldots + g_r(x)c_r(x)).$$ □

Notice that the case where $r = 2$ corresponds to the equation:

$$f(x) = \textit{if } P(x) \textit{ then } g_1(x) \textit{ else } g_2(x)$$

which we also write as:

$$\begin{aligned} f(x) &= g_1(x) \quad && \textit{if } P(x) \\ &= g_2(x) && \textit{otherwise.} \end{aligned}$$

Remark 7. In a definition by cases of a primitive recursive function, the evaluation of the predicates $P_1(x)$, $P_2(x)$, ..., and $P_r(x)$ is done sequentially from left to right. Since the corresponding characteristic functions are total, the order of evaluation of these predicates is *not* important. However, if we allow predicates with characteristic functions which are partial functions, as we will do in the sequel, this order of evaluation is important, because otherwise, the definition by cases may fail to specify a partial function. □

Now we define a new operator, called the *bounded minimalization operator*, and denoted *min*. It allows us to define a function, say f, of arity $k{+}1$, with $k \geq 0$, starting from a given predicate Q, subset of N^{k+1}, as follows. For any $x \in N^k$ and $n \in N$

$$\begin{aligned} f(x,n) &= \text{the minimum } y \in [0,n] \text{ such that } Q(x,y) \text{ holds} \\ &\qquad\qquad \textit{if there exists } y \in [0,n] \text{ such that } Q(x,y) \text{ holds} \\ &= 0 \quad \textit{otherwise} \end{aligned}$$

The function $f(x,n)$ is also denoted by $min\ y \in [0,n].Q(x,y)$.

In the *otherwise* clause we could have chosen for $f(x,n)$ a value different from 0 without modifying the theory of the partial recursive functions. The choice of 0 is convenient when formulating Lemma 9 and, in particular, when establishing Lemma 9 Point (iv).

Lemma 6. If the predicate $Q(x,y)$ subset of N^{k+1}, for $k \geq 0$, is primitive recursive then the function $f(x,n) = min\ y \in [0,n].Q(x,y)$ is primitive recursive.

Proof. Let us consider the function

$$\begin{aligned} g(x,n,x) &= m && \textit{if } \exists i \in [0,n].Q(x,i) \\ &= n{+}1 && \textit{if not}(\exists i \in [0,n].Q(x,i)) \textit{ and } Q(x,n{+}1) \\ &= 0 && \textit{if not}(\exists i \in [0,n{+}1].Q(x,i)) \end{aligned}$$

The function g is primitive recursive because it is defined by cases starting from primitive recursive functions and primitive recursive predicates. We will show that $f(x,n)$ is primitive recursive by showing that it is equal to the following function h defined by primitive recursion, as follows:

$$\begin{aligned} h(x,n) &= 0 \\ h(x,n{+}1) &= g(x,n,h(x,n)) \end{aligned}$$

(Recall that g is primitive recursive). By unfolding the definition of g, we have that the function h is also defined by the equations:

$$\begin{aligned} h(x,0) &= 0 \\ h(x,n{+}1) &= h(x,n) && \textit{if } \exists i \in [0,n].Q(x,i) \\ &= n{+}1 && \textit{if not}(\exists i \in [0,n].Q(x,i)) \textit{ and } Q(x,n{+}1) \\ &= 0 && \textit{if not}(\exists i \in [0,n{+}1].Q(x,i)) \end{aligned}$$

Now, recalling the definition of the bounded minimalization operator *min*, we also have that:

$$\begin{aligned} f(x,0) &= \ min\, y \in [0,0].\, Q(x,y) = \textit{if }\ Q(x,0) \textit{ then } 0 \textit{ else } 0 \\ f(x,n{+}1) &= \ \textit{if } \exists i \in [0,n].\, Q(x,i) \textit{ then } f(x,n) \textit{ else} \\ &\qquad \textit{if } Q(x,n{+}1) \textit{ then } n{+}1 \textit{ else } 0 \end{aligned}$$

Thus, we conclude that $h(x,n) = f(x,n)$ for all $x \in N^k$, and $n \in N$, because they are defined by the same recursion equations. □

By using Lemma 6 we can show that many other mathematical functions are primitive recursive. For instance, we have that the *integersquareroot*(x), defined as the largest number whose square does not exceed x, is a primitive recursive function. Indeed,

$$integersquareroot(x) = min\ y \in [0,x].\, (y \leq x^2) \ \text{ and } \ ((y{+}1)^2 > x).$$

Lemma 7. Let $h(n)$ be a primitive recursive function from N to N and $f(x,m)$ a primitive recursive function from N^{k+1} to N of the form: $min\ y \in [0,m].\, Q(x,y)$, where x is a k-tuple of numbers and m is a number. Then the function

$$g(x,n) = min\ y \in [0,h(n)].\, Q(x,y)$$

is primitive recursive.

Proof. The function $g(x,n)$ is defined by composition as follows:

$$g(x,n) = f(x,h(n)) = f(\pi_{k+1\,1}(x,n), \ldots, \pi_{k+1\,k}(x,n),\ h(\pi_{k+1\,k+1}(x,n))).$$ □

Lemma 8. Let $f(x,n)$ be a primitive recursive function from N^{k+1} to N where x is a k-tuple of numbers and m is a number. Then the *bounded sum of* $f(x,n)$ *below* r, denoted by $\Sigma_{n<r} f(x,n)$, defined as follows:

$\Sigma_{n<r} f(x,n) = \textit{if } r{=}0 \textit{ then } 0 \textit{ else } f(x,0) + f(x,1) + \ldots + f(x,r{-}1))$, and

the *bounded product of* $f(x,n)$ *below* r, denoted by $\Pi_{n<r} f(x,n)$, defined as follows:

$\Pi_{n<r} f(x,n) = \textit{if } r=0 \textit{ then } 1 \textit{ else } f(x,0) \times f(x,1) \times \ldots \times f(x,r-1))$,

are primitive recursive. The same holds if in the summations and in the products we replace $<$ by $\leq$. Often the qualification *'below* r' is dropped and we will simply say '*bounded sum of* $f(x,n)$' or '*bounded product of* $f(x,n)$'.

Proof. We have the following primitive recursive equations:

$\Sigma_{n<0} f(x,n) = 0 \quad \Sigma_{n<(r+1)} f(x,n) = \Sigma_{n<r} f(x,n) + f(x,r)$, and

$\Pi_{n<0} f(x,n) = 1 \quad \Pi_{n<(r+1)} f(x,n) = \Pi_{n<r} f(x,n) \times f(x,r)$.

For instance, if we write $\Sigma_{n<r} f(x,n)$ as $S(x,r)$, we have that:

$S(x,0) = 0$

$S(x,r+1) = S(x,r) + f(x,r)$.

$S(x,r)$ is a primitive recursive function, because $S(x,r) + f(x,r) = sum1(x,r,S(x,r))$ and $sum1(x,r,m)$ is a primitive recursive function. Indeed, we have that:

$sum1(x,r,m) = sum(\pi_{33}(x,r,m), f(\pi_{31}(x,r,m), \pi_{32}(x,r,m)))$.

When replacing $<$ by $\leq$ we have:

$\Sigma_{n\leq r} f(x,n) = \Sigma_{n<(r+1)} f(x,n)$, and $\Pi_{n\leq r} f(x,n) = \Pi_{n<(r+1)} f(x,n)$.

Thus, we have that for all $r \geq 0$, the functions $\lambda x.\Sigma_{n\leq r} f(x,n)$ and $\lambda x.\Pi_{n\leq r} f(x,n)$ are primitive recursive by Lemma 7. □

Lemma 9. The following functions are primitive recursive:

(i) $D(n)$ = the number of divisors of n, according to the following definition of divisor:

y is a divisor of x iff $div(x,y) = 1$ and $y \leq x$.

Thus, we have: $D(0) = 1$, $D(1) = 1$, $D(2) = 2$, $D(3) = 2$, $D(4) = 3$, $D(5) = 2, \ldots$

(ii) $Pr(n) = \textit{if } (n\geq 2 \text{ and } n \text{ is prime}) \textit{ then } 1 \textit{ else } 0$

(iii) $p(n)$ (also written as p_n) = the n-th prime number, with $n\geq 0$, defined as follows:

$p_0 = 0$, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, $p_4 = 7$, $p_5 = 11, \ldots$

(iv) $e(x,n) = \textit{if } n=0 \textit{ or } x=0 \textit{ then } 0 \textit{ else}$ the exponent of p_n in the prime factorization of x.

Thus, for instance, $e(x,1)$, $e(x,2)$, and $e(x,3)$ are the exponents of 2, 3, and 5, respectively, in the prime factorization of x.

(v) The inverse functions of the so called *pair function* $C(x,y) = 2^x(2y+1) \neg 1$. Notice that $C(x,y)$ is a bijection from N^2 to N.

Proof. (i) $D(x) = D1(x,x)$ where the function $D1$ is defined as follows:

$D1(x,0) = div(x,0)$ and $D1(x,n+1) = D1(x,n) + div(x,n+1)$.

(ii) $Pr(x) = \textit{if } D(x) = 2 \textit{ then } 1 \textit{ else } 0$.

$D(x)=2$ means that $x>1$ and the only divisors of x are 1 and x. Thus,

$Pr(x) = nsg(|D(x)-2|)$.

(iii) $p_0 = 0$ and $p_{n+1} = min\, z \in [0, p_n!+1].\, (z > p_n \text{ and } Pr(z))$.

This is a definition by primitive recursion, because: (a) $(z > p_n \text{ and } Pr(z))$ is a primitive recursive predicate, (b) we may apply Lemma 7 for $h(n) = p_n!+1$, and (c) the remainders of the n divisions of $p_n!+1$ by $p_0, p_1, \ldots$, and p_n, respectively, are all different from 0, and thus, in the interval $[p_n+1,\ p_n!+1]$ there is at least one prime.

(iv) $e(x,n) = min\, z \in [0,x].\, (div(x, p_n^{z+1}) = 0)$.

(v) Let n be $C(x, y)$. Thus, $n+1$ is $2^x(2y+1)$. We have that: $x = e(n+1, 1)$, because $2y+1$ is odd, and $y = quot(quot(n+1, 2^x)\neg 1, 2)$. $\square$

An element $\langle x_1, x_2, \ldots, x_k\rangle$ of N^k, for $k \geq 0$, can be encoded by the following element of N:

$\beta(x_1, x_2, \ldots, x_k) = if\ \ k=0\ then\ 1\ else\ (p_1^{x_1+1} \times p_2^{x_2+1} \times \ldots \times p_k^{x_k+1})$.

From the value of $\beta(x_1, x_2, \ldots, x_k)$, call it B, we can get the values of k and those of the x_i's for $i = 1, \ldots, k$, as follows:

$k = if\ \ B=1\ then\ 0\ else\ min\, z \in [0, B].\, (e(B, z+1) = 0)$, and

for $i = 1, \ldots .k$, $x_i = e(B, i)\neg 1$.

The encoding β from N^* to N is *not* a bijection. Indeed, for instance, no element x exists in N^* such that $\beta(x) = 3$. The same holds for $\beta(x) = 0$.

The following encoding τ of k-tuples is a bijection from N^* to N. Given an element $\langle x_1, x_2, \ldots, x_k\rangle$ of N^k, for $k \geq 0$, let τ be the following function with arity k:

$$\tau(x_1, x_2, \ldots, x_k) = if\ \ k=0\ then\ 0\ else$$
$$2^{x_1} + 2^{x_1+x_2+1} + 2^{x_1+x_2+x_3+2} + \ldots + 2^{x_1+x_2+\ldots+x_k+k\neg 1}.$$

If $k = 0$ then $\tau(x_1, x_2, \ldots, x_k) = \tau(\langle\rangle) = 0$. If $k > 0$ then $\tau(x_1, x_2, \ldots, x_k) > 0$. We have that τ is primitive recursive.

Now, given the value of $\tau(x_1, x_2, \ldots, x_k)$ we can compute τ^{-1}, that is, we can derive the values of $x_1, x_2, \ldots,$ and x_k, by using primitive recursive functions only as we now indicate.

We have that $\tau^{-1}(0) = \langle\rangle$.

Given any $x \geq 0$, we have that x has a unique expression of the form:

$x = \alpha(0, x)2^0 + \ldots + \alpha(i, x)2^i + \ldots$ with $\alpha(i, x) \in \{0, 1\}$ for any $i \geq 0$.

Thus, the values of $\alpha(i, x)$ for $i \geq 0$, give us the familiar binary expansion of the number x. Given any $x > 0$, it has a unique expression of the form:

$$x = 2^{b(1,x)} + 2^{b(2,x)} + \ldots + 2^{b(M(x),x)}$$

with $0 \leq b(1, x) \leq b(2, x) \leq \ldots \leq b(M(x), x)$ and $1 \leq M(x)$, where $M(x)$ is the number of the powers of 2 occurring in the binary expansion of x. The number x has also a unique expression of the form:

$$x = 2^{a(1,x)} + 2^{a(1,x)+a(2,x)+1} + 2^{a(1,x)+a(2,x)+a(3,x)+2} + \ldots$$
$$+\, 2^{a(1,x)+a(2,x)+\ldots+a(M(x),x)+k\neg 1}$$

where $1 \leq M(x)$ and for $j = 1, \ldots, M(x)$, we have that $a(j, x) \geq 0$.

For example, if $x = 2^5 + 2^8 + 2^{15}$ we have that:

$b(1, x)=5$, $b(2, x)=8$, $b(3, x)=15$, $M(x)=3$, $a(1, x)=5$, $a(2, x)=2$, $a(3, x)=6$, $k=3$, and $\tau^{-1}(x)=\langle 5, 2, 6\rangle$.

The following expressions show that the function τ^{-1} is primitive recursive:

(i) $\alpha(0, x) = rem(x, 2)$ $\qquad$ $\alpha(i+1, x) = \alpha(i, quot(x, 2))$ for $i \geq 0$

(ii) Let us denote $b(M(x), x)$ by $E(x)$. We have that $E(x)$ is the highest exponent of 2 occurring in the binary expansion of x. We have:

$E(x) = if\ \ x=0\ then\ 0\ else\ min\ z \in [0, x].\, (2^{z+1} > x)$.

(iii) $M(x) = if\ \ x=0\ then\ 0\ else\ K(E(x), x)$

where $K(0, x) = \alpha(0, x)$ and $K(j+1, x) = K(j, x) + \alpha(j+1, x)$ for any $j \geq 0$.

$K(E(x), x)$ is the sums of the exponents of the powers of 2 occurring in the binary expansion of x. We also have that:

$M(x) = \Sigma_{j<x}\, \alpha(i, x)$.

(Recall that since for any $x \geq 0$, $x < 2^x$ we have that: for any $i \geq x$, $\alpha(i, x){=}0$).

(iv) $b(i, x) = \mathit{if}\ x{=}0\ \mathit{or}\ i{=}0\ \mathit{or}\ i{>}M(x)\ \mathit{then}\ 0\ \mathit{else}$
$\qquad\qquad \mathit{if}\ i{=}1\ \mathit{then}\ r\ \mathit{else}\ b(i{-}1, x{-}2^r)$

where $r = \mathit{min}\, z \in [0, x].\, (\alpha(z, x) > 0)$. Thus, r is the smallest power of 2 occurring in the binary expansion of x. Notice that if the condition '$x{=}0\ \mathit{or}\ i{=}0\ \mathit{or}\ i{>}M(x)$' is false then i is in the interval $[1, M(x)]$. We also have:

$b(i, x) = \mathit{min}\, z \in [0, x]\, .((\Sigma_{k\leq z}\, \alpha(k, x)) = i)$.

(v) The function $\alpha(j, x)$ can be expressed as follows:

$\alpha(j, x) = \mathit{if}\ x{=}0\ \mathit{or}\ j{=}0\ \mathit{or}\ j{>}M(x)\ \mathit{then}\ 0\ \mathit{else}$
$\qquad\qquad \mathit{if}\ j{=}1\ \mathit{then}\ b(1, x)\ \mathit{else}\ ((b(j, x) \neg b(j{-}1, x)) \neg 1)$.

Exercise 1. (i) Show that primitive recursive functions are closed under mutual recursive definitions. Without loss of generality, let us consider the following functions g and h with one argument only:

$$\begin{aligned}
g(0) &= a\\
g(n{+}1) &= r(n, g(n), h(n))\\
h(0) &= b\\
h(n{+}1) &= s(n, g(n), h(n)
\end{aligned}$$

where the functions r and s are primitive recursive. We have that the function $f(n) = 2^{g(n)}\, 3^{h(n)}$ is primitive recursive, because:

$$\begin{aligned}
f(0) &= 2^a 3^b\\
f(n{+}1) &= 2^{g(n+1)} 3^{h(n+1)} =\\
&= 2^{r(n,g(n),h(n))} 3^{s(n,g(n),h(n))} =\\
&= 2^{r(n,u,v)} 3^{s(n,u,v)} \text{ where } \langle u, v\rangle = \langle e(f(n), 1),\ e(f(n), 2)\rangle
\end{aligned}$$

where the definition of the function $\lambda x, n.\, e(x, n)$ is given in Lemma 9 Point (iv). Thus, $g(n) = e(f(n), 1)$ (because $g(n)$ is the exponent of 2 in the binary expansion of $f(n)$) and $h(n) = e(f(n), 2)$ (because $h(n)$ is the exponent of 3 in the binary expansion of $f(n)$).

(ii) Show that the Fibonacci function is primitive recursive. We have that:

$$\begin{aligned}
\mathit{fib}(0) &= 0\\
\mathit{fib}(1) &= 1\\
\mathit{fib}(n{+}2) &= \mathit{fib}(n{+}1) + \mathit{fib}(n)
\end{aligned}$$

We have that the function $f(n) = 2^{\mathit{fib}(n+1)}\, 3^{\mathit{fib}(n)}$ is primitive recursive, because:

$$\begin{aligned}
f(0) &= 2^1 3^0\\
f(n{+}1) &= 2^{\mathit{fib}(n+2)} 3^{\mathit{fib}(n+1)} =\\
&= 2^{u+v} 3^u \text{ where } \langle u, v\rangle = \langle \mathit{fib}(n{+}1),\ \mathit{fib}(n)\rangle\\
&= 2^{u+v} 3^u \text{ where } \langle u, v\rangle = \langle e(f(n), 1),\ e(f(n), 2)\rangle
\end{aligned}$$

where the definition of the function $\lambda x, n.\, e(x, n)$ is given in Lemma 9 Point (iv).

Thus, $\mathit{fib}(n) = e(f(n), 2)$. □

Remark 8. The *course-of-value recursion* can be reduced to primitive recursion (see the result by R. Péter presented in [16, page 273]), in the sense that we get an equivalent definition of the set of the primitive recursive functions if in Definition 14 we replace the second equation of (v) by the following one:

$$f(x_1{+}1, x_2, \ldots, x_k) =$$
$$h(x_1, x_2, \ldots, x_k, f(x_1, x_2, \ldots, x_k), f(x_1{-}1, x_2, \ldots, x_k), \ldots, f(0, x_2, \ldots, x_k)). \qquad \square$$

13 Partial Recursive Functions

Now we introduce the minimalization operator μ w.r.t. a class C of functions. Given a function t from N^{k+1} to N for some $k \geq 0$, in the class C, the minimalization operator μ allows us to define a new function f from N^k to N, denoted by $\lambda x.(\mu y.[t(x, y) = 0])$ such that for any $x \in N^k$,

$$\begin{array}{rll} f(x) & = \text{the minimum } y \text{ such that } t(x,y){=}0 & \textit{if} \text{ there exists } y \in N \text{ such that } t(x,y){=}0 \\ & = \text{undefined} & \textit{otherwise} \end{array}$$

In the literature there exists a different minimalization operator, call it μ^*, w.r.t. a class C of functions. Given a function t from N^{k+1} to N for some $k \geq 0$, in the class C, the minimalization operator μ^* allows us to define a new function f^* from N^k to N, denoted by $\lambda x.(\mu^* y.[t(x, y) = 0])$ such that for any $x \in N^k$,

$$\begin{array}{rll} f^*(x) & = \text{the minimum } y \text{ such that} & \text{(i) for all } d{\leq}y, t(x,d) \text{ is defined and (ii) } t(x,y){=}0, \\ & & \textit{if} \text{ there exists } y \in N \text{ such that} \\ & & \quad \text{(ii.1) for all } d \leq y,\ t(x,d) \text{ is defined and} \\ & & \quad \text{(ii.2) } t(x,y) = 0 \\ & = \text{undefined} & \textit{otherwise} \end{array}$$

Remark 9. (i) When applying the operators μ and μ^* we need to know the class C of functions from which the function t is taken.

(ii) When introducing the minimalization operators, one could consider, instead of the predicate $\lambda x, y.[t(x, y) = 0]$, the predicate $\lambda x, y.[t(x, y) = r]$ for some fixed number $r \in N$. This different choice of r does not modify the theory of the partial recursive functions. □

Now we define the set PRF of the partial recursive functions and, for any $k \geq 0$, the sets PRF_k of the partial recursive functions of arity k.

Definition 16. The set PRF of all *partial recursive functions* (*p.r.f.*'s, for short) is the smallest set of partial functions in $\bigcup_{k\geq 0}(N^k \to N)$ which includes: (i.1) the zero functions, (i.2) the successor function, and (i.3) the projection functions (as defined in Definition 14) and (ii) it is closed w.r.t. the following operations: (ii.1) composition, (ii.2) primitive recursion, and (ii.3) minimalization μ w.r.t. the subset of the partial recursive functions themselves which are *total* functions.

The set PRF_k of the partial recursive functions of arity k, for any $k \geq 0$, is the subset of PRF which includes all partial recursive functions of arity k, that is, from N^k to N. □

It follows from the above Definition 16 that the set of the partial recursive functions is denumerable. Indeed, for the set PRF we have the following domain equation [24, Chapter 4]:

$$\mathrm{PRF} = N \ + \ \{1\} \ + \ N^{*}{\times}N + \mathrm{PRF}{\times}\mathrm{PRF}^{*} + \mathrm{PRF}{\times}\mathrm{PRF} + \mathrm{TPRF}$$

where TPRF is the subset of PRF which includes all partial recursive functions which are total. Obviously, the cardinality of TPRF is not smaller than that of N (because for all $n \in N$, the constant function $\lambda x.n$ is in TPRF) and it is not larger than that of PRF.

One can show that the set PRF_k of partial recursive functions is *not* closed under μ operations w.r.t. the class PRF_{k+1} of functions. (Notice that PRF_{k+1} include also partial functions.) In particular, for $k = 1$ there exists a partial recursive function f such that $\lambda n.(\mu y.[f(n, y) = 0])$ is *not* a partial recursive function [25, page 42].

One can also give the following equivalent definition of the set PRF of the partial recursive functions.

Definition 17. The set PRF of the partial recursive functions is the smallest set of partial functions $\bigcup_{k \geq 0}(N^k \rightarrow N)$ which: (i) includes all primitive recursive functions, and (ii) it is closed w.r.t.: (ii.1) composition, (ii.2) primitive recursion, and (ii.3) minimalization μ^{*} w.r.t. the set PRF itself of the partial recursive functions. □

In the above Definition 17, instead of saying 'includes all primitive recursive functions', we could have said, as in Definition 16, 'includes the zero functions, the successor function, and the projection functions'.

Theorem 14. Each partial recursive function of one argument can be expressed as:
$\lambda n.p(\mu y.[t(n, y) = 0])$
where p and t are primitive recursive functions.

As a consequence of Theorem 14, we have an alternative proof that the set of p.r.f.'s is denumerable. Indeed, the partial recursive functions are as many as the pairs of primitive recursive functions, that is, $|N \times N|$ (which is equal to $|N|$).

Remark 10. A statement similar to the one of Theorem 14 where the function p is omitted, does *not* hold. In particular, there exists a p.r.f. f such that no total p.r.f. g exists such that f is $\lambda n.(\mu y.[g(n, y) = 0])$ [25, page 37]. □

Theorem 15. (Kleene Normal Form) The set PRF_1 of the p.r.f.'s of arity 1 can be obtained by choosing two fixed primitive recursive functions P (of arity 1) and T (of arity 3). We have that $\mathrm{PRF}_1 = \{\lambda n.P(\mu y.[T(i, n, y) = 0]) \mid i \geq 0\}$. □

The function T of Theorem 15, also called the *Kleene predicate*, can be described by saying that $T(i, n, y) = 0$ iff i is the encoding of a Turing Machine M such that M with input n computes a value in fewer than y computation steps. (Notice also that Turing Machines can be encoded into numbers.) Since the function T is primitive recursive Theorem 15 holds also for μ^{*}, instead of μ.

For all $i \geq 0$ the expression $\lambda n.P(\mu y.[T(i, n, y) = 0])$ defines the i-th partial recursive function of arity 1. Every partial recursive function of arity 1 has infinitely many indices, that is, for every p.r.f. f from N to N there exists an infinite subset I of N such that for all $i \in I$ we have that $f = \lambda n.P(\mu y.[T(i, n, y) = 0])$.

Some partial recursive functions are total. Indeed, this is the case when, with reference to Definition 16, for all $x \in N^k$ there exists a minimum value of y such that $t(x, y) = 0$.

A total p.r.f. is also called a *recursive function*. In the literature, unfortunately, partial recursive functions sometimes are called *recursive functions* tout court. The reader should be aware of this ambiguous terminology.

The set of the total partial recursive functions can be defined as in Definition 16 with the extra condition that the total partial recursive function t which is used in the minimalization operator μ, satisfies the following property:
$\forall x \in N. \exists y \in N. t(x, y) = 0$.

Definition 18. A predicate, subset of N^k, is said to be *recursive* iff its characteristic function from N^k to N is a recursive function.

There are p.r.f.'s which are always undefined, that is, p.r.f.'s whose domain is empty. One such function is, for instance, $\lambda n.(\mu y.[t(n, y) = 0])$ where $t = \lambda n, y.1$ (that is, t is the constant function of two arguments which returns 1).

There exists a total p.r.f. which is *not* a primitive recursive function. A p.r.f. which is total and is not primitive recursive is the Ackermann function from $N \times N$ to N defined as follows:

for any $m, n \in N$,

$$\begin{aligned}
ack(0, n) &= n+1 \\
ack(m+1, 0) &= ack(m, 1) \\
ack(m+1, n+1) &= ack(m, ack(m+1, n))
\end{aligned}$$

We omit the proof that the Ackermann function is a p.r.f. which is *not* a primitive recursive function. The proof that the Ackermann function is not primitive recursive is based on the fact that it grows faster than any primitive recursive function (see also [5, page 194]), that is, for all primitive recursive function p there exists n such that $ack(n, n) > p(n)$.

The total p.r.f.'s are as many as $|N|$.

Theorem 16. There exists a p.r.f. f from N to N which cannot be extended to a total p.r.f., call it g, from N to N, in the sense that for all $n \in N$ if $f(n)$ is convergent then $f(n) = g(n)$.

Proof. Assume that for each p.r.f. f_i there exists a total p.r.f., call it f_{i^*}, which extends f_i. Let us consider the function h such that for all $i \geq 0$,

$h(i) = \textit{if } f_i(i) \text{ is defined } \textit{then } f_i(i)+1 \textit{ else} \text{ undefined}$

where f_i is the i-th p.r.f. from N to N. Thus, we have that:

$h(i) = \textit{if } \mu y.[T(i, i, y) = 0] \geq 0 \textit{ then } f_i(i)+1 \textit{ else } \mu y.[1(y) = 0]$

where $\lambda y.1$ is the constant function of one argument which returns 1. From this expression and recalling that the function T is total, we have that the function h is a p.r.f. Let us assume that the function h is the p.r.f. of index r, that is, $h = f_r$. By assumption, we have that there exists the p.r.f. f_{r^*} which is total. Since (a) if $f_i(i)$ is defined then h is defined and (b) f_{r^*} extends h, we have that for all $i \geq 0$,

if $f_i(i)$ is defined *then* ($f_{r^*}(i)$ is defined and $f_{r^*}(i) = f_i(i)+1$).

If we consider $i = r^*$ we get that:

if $f_{r^*}(r^*)$ is defined *then* ($f_{r^*}(r^*)$ is defined and $f_{r^*}(r^*) = f_{r^*}(r^*)+1$),
which is a contradiction because f_{r^*} is total and, thus, $f_{r^*}(r^*)$ is defined. □

From Definition 16 it follows that given a partial recursive function g of $k+1$ arguments which is total, then the function $\lambda x_1, \ldots, x_k.(\mu y.[g(x_1, \ldots, x_k, y) = 0])$ is a partial recursive function which is not necessarily total because it may be the case that for all $x_1, \ldots, x_k$, no y exists such that $g(x_1, \ldots, x_k, y) = 0$.

14 Partial Recursive Functions and Recursively Enumerable Sets

Definition 19. A set A is *recursively enumerable*, r.e. for short, iff either $A = \emptyset$ or there exists a total p.r.f. f from N to N such that $A = \mathit{range}(f)$. □

This definition motivates the name 'recursively enumerable' given to the set A. Indeed, assuming that A is not empty, we have that the elements of the recursively enumerable set A are 'enumerated, possibly with repetitions,' by a total p.r.f. f in the sense that $A = \{f(i) \mid i \in N\}$. The meaning of 'possibly with repetitions' is that in the sequence: $f(0)$, $f(1)$, $f(2), \ldots$, we may have identical elements of the set A.

Let us now introduce the notion of *dove-tailing evaluation* of a partial recursive function.

For this purpose we first need to consider a suitable *model of computation* and a notion of *computation step* relative to that model. We assume that the notion of computation step is a primitive notion and, thus, we need not to specify it. With reference to Definition 16, we will only say that a computation step can be understood as:
- in cases (i.1)-(i.3) the evaluation of a zero function, or the successor function, or a projection function, or
- in cases (ii.1) and (ii.2) the replacement of a left-hand-side by the corresponding right-hand-side, or else
- in case (ii.3) the computation step of the function t relative to the minimalization operator μ.

Given a partial recursive function f from N to N, we say that the *bounded evaluation of* $f(i)$ for j computation steps is:
- $f(i)$ if performing *at most* j computation steps the evaluation of $f(i)$ gives us a number in N,
- the distinguished value # (not in N), otherwise.

Given a partial recursive function f from N to N, its *dove-tailing evaluation* is the construction of a two-dimensional matrix where at row $i\,(\geq 0)$ and column $j\,(\geq 0)$ we place the bounded evaluation of $f(i)$ for j computation steps (see also Figure 17). This matrix is constructed element by element, in the increasing order of the entries which we denote by $(0), (1), (2), \ldots$, called *position numbers*. For any $i, j \in N$, at row i and column j we have the unique position number $D(i,j) = ((i+j)^2 + 3i + j)/2$.

There are two functions r and c from N to N such that for any position number $d\,(\geq 0)$ we have that d occurs in the dove-tailing matrix at row $r(d)$ and column $c(d)$. The two functions r and c are primitive recursive, because as the reader may verify, $r(d) = d - S$ and $c(d) = s - r(d)$, where

$S = quot((s^2+s), 2)$ and

$s = quot(integersquareroot(8d+1)+1, 2) - 1$.

computation step j:	0	1	2	3	...
argument i: 0	(0)	(1)	(3)	(6)	...
1	(2)	(4)	(7)	...	
2	(5)	(8)	...		
3	(9)	...			
...	...				

Fig. 17. The matrix of the position numbers for the dove-tailing evaluation of a p.r.f. from N to N.

Now we state an important result about the dove-tailing evaluation of any p.r.f.: there exists a partial recursive function, call it I, from PRF$\times N \times N$ to $N \cup \{\#\}$ which given any partial recursive function f and the numbers i and j, computes the value of the bounded evaluation of $f(i)$ for j computation steps. The function I is a total function as we will see below.

The proof of the existence of the function I (given in [5, Chapter 5] and [27]) relies on the existence of the Universal Turing Machine and on the fact (not proved here) that there exists a bijection between the set PRF of the partial recursive functions and the set of the sequences of instructions for Turing Machines, that is, the set of the Turing Machines themselves (recall also that an instruction for a Turing Machine can be encoded as a quintuple of numbers). This bijection exists because of: (i) the injection from PRF to the set of the Turing Machines, (ii) the injection from the set of the Turing Machines to PRF, and (iii) the Bernstein Theorem [23].

Since a sequence of instructions can be encoded by a sequence of numbers (see, for instance, [23]), and consequently by a single number (see the encoding function τ above), we have that any partial recursive function, say f, can be encoded by a number, say $\tau^*(f)$, which is called the *number encoding* of f.

Thus, when we say that I takes as input a p.r.f. f we actually mean that it takes as input the number $\tau^*(f)$. The function I is a total function from PRF$\times N \times N$ to $N \cup \{\#\}$. It can also be considered as a function from N to N because of the bijection between PRF and N, the bijection between N^3 and N, and the bijection between $N \cup \{\#\}$ and N.

The function I extracts from the value of $\tau^*(f)$ the sequence of instructions corresponding to the f, and then it behaves as an interpreter which uses that sequence as a program for evaluating $f(i)$ for j computation steps.

An informal proof of the existence of the function I. The proof is based on the fact that the function I can be constructed by referring to the set of recursive equations which defines any given p.r.f.

Indeed, given a p.r.f., say f, it can be viewed as a set, call it Eq_f, of recursive equations constructed from the zero, successor, and projection functions, by using composition, primitive recursion, and minimalization. In particular, a function $\lambda x.g(x)$ defined by the minimalization operation of the form $\mu y.[t(x,y) = 0]$ is equivalent to the following two equations:

$g(x) = aux(x, 0)$
$aux(x, y) = if\ \ t(x, y) = 0\ \ then\ y\ \ else\ \ aux(x, y{+}1)$

Then the set Eq_f of recursive equations can be viewed as a program for the function f. We can encode that program by a number which we also denote, by abuse of language, $\tau^*(f)$.

Let us consider the computational model related to the leftmost-innermost rewriting of subexpressions and let us consider a computation step to be a rewriting step. The initial expression e_0 (that is, the expression at computation step 0) is the number which encodes $f(i)$, for any given p.r.f. f and input number i. Then, given the number which encodes the expression e_k at the generic computation step $k\,(\geq 0)$, the function I produces the number which encodes the expression e_{k+1} at the next computation step $k{+}1$ as follows. The function I finds in the program a recursive equation, say $L = R$, such that the leftmost-innermost subexpression u of e_k is equal to $(L\,\vartheta)$ for some substitution ϑ, and it replaces the subexpression u in e_k by $(R\,\vartheta)$.

We hope that the reader may convince himself that the operations of: (i) finding the leftmost-innermost subexpression, (ii) finding the recursive equation to be applied, and (iii) rewriting that subexpression, are all operations which can be performed starting from the number $\tau^*(f)$ and the number which encodes the expression e at hand, by the application of a suitable total partial recursive function which operates on numbers. □

Theorem 17. A set A is r.e. iff A is the domain of a p.r.f.

Proof. ($\Rightarrow$) Let us call p the p.r.f. to be found, whose domain is A. If $A = \emptyset$ the function p from N to N is the everywhere undefined partial recursive function. This function can be denoted by the lambda term $\lambda n.(\mu y.[1 = 0])$. (Obviously, infinite many other lambda terms denote the everywhere undefined partial recursive function.)

If $A \neq \emptyset$ then, by definition, A is the range of a *total* p.r.f., say f. A is the domain of the partial recursive function p which is $\lambda x.\mu y.[f(y) - x = 0]$. Indeed, $x \in A$ iff $\exists y. f(y) = x$ iff p is defined in x.

($\Leftarrow$) We have that A is the domain of a p.r.f., call it g. By definition, for all $x \in A$, $g(x)$ is defined, i.e., $g(x) \in N$. We have to show that A is r.e.

If $A = \emptyset$ then there is nothing to prove because A is r.e. by definition.

If $A \neq \emptyset$ then we show that A is r.e. by constructing a total p.r.f., call it h, such that A is the range of h.

Let us first consider a list of pairs of numbers, call it G, which is constructed as follows. The initial value of G is the empty list. Then the value of G is updated by inserting in it new pairs of numbers, as we now indicate.

We first perform the dove-tailing evaluation of the function g. We will see below that we can perform the dove-tailing evaluation of the function g in an incremental way, while computing the (possibly infinite) set of pairs that constitutes the function h. Let us assume that by dove-tailing evaluation of g, we get a table whose initial, left-upper portion is depicted in Figure 18.

computation step j:	0	1	2	3	4	5	...
argument i: 0	#	#	#	#	#	#	...
1	#	#	#	#	#	...	
...	...						
x	#	#	m	m	...		
...	...						

Fig. 18. Dove-tailing evaluation of the function g.

When a value of $g(x)$, say m, different from #, is obtained in row x and column y, we insert at the right end of the list G the pair $\langle x, D(x,y)\rangle$ (here we assume that the list G grows to the rigth). (Recall that: $D(x,y) = ((x+y)^2 + 3x + y)/2$.) By definition of dove-tailing, if $\langle x, D(x,y)\rangle$ is in G then in G there are infinitely many pairs whose first component is x (because there are infinitely many m's in row x in Figure 18). In particular, if $m \in N$ is in row x and column y then for each column $k \geq y$, m is in row x and column k.

Since A is the domain of g, $g(x)$ is convergent for each $x \in A$, and therefore for every $x \in A$ there exists $y \in N$ (actually, there exist infinitely many y's) such that the pair $\langle x, D(x,y)\rangle$ is inserted into G.

Since A is not empty, the list G has at least one pair. Let us consider the pair of G with the smallest second component, that is, the pair which is first inserted into G during the dove-tailing evaluation of g. Let that pair be $\langle x_0, d_0\rangle$. (Recall that, by definition of the function r which computes the row of any given position number, we have that $x_0 = r(d_0)$.) Thus, the list G is of the form:

$G = [\langle x_0, d_0\rangle, \langle x_1, d_1\rangle, \ldots, \langle x_n, d_n\rangle, \ldots]$

where $\langle d_0, d_1, \ldots, d_n, \ldots\rangle$ is an infinite sequence of increasing numbers, that is, $d_0 < d_1 < \ldots < d_n < \ldots$

Notice that the list G is infinite even if A is a finite set (by hypothesis, A is non-empty). We also have that for all $x \in A$ and for all $d \geq 0$ there exists $d_1 > d$ such that if $\langle x, d\rangle$ is in G then $\langle x, d_1\rangle$ is also in G.

Let us define the function h from N to N, as follows:

(1) $h(0) = x_0$

(2) $h(n+1) = \mathit{if}\ \langle x, n+1\rangle$ is in G $\mathit{then}\ x\ \mathit{else}\ x_0$

Thus, for every $n \in N$, we have that $h(n) = r(n)$ if the bounded evaluation of $g(r(n))$ returns a number within $c(n)$ computation steps, otherwise $h(n) = x_0$. As a consequence, since $A \neq \emptyset$ and the pair $\langle x_0, d_0\rangle$ occurs in G, we have that for every $n \in N$, in order to compute $h(n)$, we only need to compute a finite portion of the dove-tailing of the function g.

We have the following three facts.

(i) The function h is p.r.f. because the function I which for any $x \geq 0$ and any $n \geq 0$, performs the bounded evaluation of $g(x)$ for $c(n)$ computation steps, is a total partial recursive function from N^3 to $N \cup \{\#\}$, and so is the function which for $i = 0, \ldots, n+1$, performs the bounded evaluations of $g(r(i))$ for $c(i)$ computation steps. Thus, the predicate '$\langle x, n+1 \rangle$ is in G' is recursive.
(ii) By definition of h, we have that h is a total p.r.f.
(iii) Moreover, the range of h is A. Indeed, let us consider any $x \in A$. Since A is the domain of the partial recursive function g, for that x there exists a position number d such that: (iii.1) $x = r(d)$, and (iii.2) the bounded evaluation of $g(x)$ returns an element of N (that is, it does not return not $\#$) within $c(d)$ computation steps. By (iii.2) we have that $\langle x, d \rangle$ is in G and, as a consequence, $h(d) = x$ (see Equation (2) above). □

In order to understand the ($\Leftarrow$) part of the proof of Theorem 17 and to clarify the definition of the function h, let us now consider the following example.

Example 3. Let us assume that the dove-tailing evaluation of the function g, depicted in Figure 19 below, produces the following list:

$G = [\langle 2, 8 \rangle, \langle 1, 11 \rangle, \langle 2, 12 \rangle, \langle 1, 16 \rangle, \langle 2, 17 \rangle, \langle 4, 19 \rangle, \ldots]$

$\langle 2, 8 \rangle$ means that at position number 8, that is, at row $r(8)=2$ and column $c(8)=1$, we have computed the value of $g(2)$, which is m_8, with at most 1 computation step,
$\langle 1, 11 \rangle$ means that at position number 11, that is, at row $r(11)=1$ and column $c(11)=3$, we have computed the value of $g(1)$, which is m_{11}, with at most 3 computation steps,
$\langle 2, 12 \rangle$ means that at position number 12, that is, at row $r(12)=2$ and column $c(12)=2$, we have computed (again) the value of $g(2)$, which is $m_{12}=m_8$, with at most 2 computation steps,
. . .,
$\langle 4, 19 \rangle$ means that at position number 19, that is, at row $r(19)=4$ and column $c(19)=1$, we have computed the value of $g(4)$, which is m_{19}, with at most 1 computation step,
. . .

computation step j:	0	1	2	3	4	5	. . .
argument i: 0	(0)	(1)	(3)	(6)	(10)	(15)	. . .
1	(2)	(4)	(7)	m_{11}	m_{16}	. . .	
2	(5)	m_8	m_{12}	m_{17}	. . .		
3	(9)	(13)	(18)	. . .			
4	(14)	m_{19}	. . .				
. . .	. . .						

Fig. 19. Dove-tailing evaluation of the function g. The r.e. set A which is the domain of g is $\{2, 1, 4, \ldots\}$. It is also the range of the function h defined by the Equations (1) and (2).

In the matrix of Figure 19 we have written, instead of the $\#$'s, the corresponding position numbers between parentheses. The entries of that matrix different from $\#$, which

are in the same row, are all equal. Indeed, they are the value of the function g for the same input. In particular, in our matrix of Figure 19:
(i) for row 1, we have: $g(1) = m_{11} = m_{16} = \ldots$,
(ii) for row 2, we have: $g(2) = m_8 = m_{12} = m_{17} = \ldots$,
(iii) for row 4, we have: $g(4) = m_{19} = \ldots$, etc.

Since during the dove-tailing evaluation the first computed value of the function g is $g(2) = m_8$, we have that in the definition of the function h, the value of x_0 is 2. Thus,

n:	0	1	2	...	8	9	10	11	12	...	15	16	17	18	19	...
$h(n)$:	2	2	2	...	2	2	2	1	2	...	2	1	2	2	4	...

To sum up, in order to compute $h(n)$ we first perform the dove-tailing evaluation of the function g until the following two conditions are satisfied:
(i) the position at row $r(n)$ and column $c(n)$ has been filled with the value of the bounded evaluation of $g(r(n))$ with at most $c(n)$ computation steps (it may be either a number or #), and
(ii) the value of g has been computed for at least one element in A, call it x_0, (recall that A is not empty, by hypothesis) with x_0 different from #. □

Theorem 18. A set A is r.e. iff A is the range of a p.r.f.

Proof. ($\Rightarrow$) Let us call p the p.r.f. to be found whose range is A. If $A = \emptyset$ the function p is the everywhere undefined partial recursive function. This function can be denoted by the lambda term $\lambda n.(\mu y.[1 = 0])$.

If $A \neq \emptyset$ the p.r.f which has A as range, is the total p.r.f. which exists because A is, by definition, the range of a total p.r.f.

($\Leftarrow$) We have that A is the range of a p.r.f., call it g.

If $A = \emptyset$ then A is r.e. by definition.

If $A \neq \emptyset$ then we show that A is r.e. by constructing a total p.r.f., call it h, such that A is the range of h.

The proof of this part of the theorem is like part ($\Leftarrow$) of the proof of Theorem 17, with the exception that:

(i) during the construction of the list G, when the value of $g(x)$, say m, different from #, is obtained in row x and column y, we insert at the right end of the list G the pair $\langle m, D(x, y)\rangle$, instead of the pair $\langle x, D(x, y)\rangle$, and

(ii) the function h is defined as follows:

(1′) $h(0) = m_{in}$

(2′) $h(n+1) = \mathit{if}\ \langle m, n+1\rangle$ is in G $\mathit{then}\ m\ \mathit{else}\ m_{in}$

where m_{in} is the first component of the pair which is first inserted into G. Notice that, since $A \neq \emptyset$, G must have at least one pair.

By definition, the function h is total.

We also have that A is the range of h. Indeed, for every $x \in A$, x is by hypothesis in the range of g, and thus, there exists k such that the bounded evaluation of $g(r(k))$ gives us the number x within $c(k)$ computation steps. As a consequence, for every $x \in A$ there exists $k \in N$ such that since $\langle x, k\rangle$ is in G and thus, by definition of h (see Equation 2 above), we have that for all $x \in A$ there exists $k \in N$ such that $h(k) = x$. This shows that A is the range of h. □

In order to understand the ($\Leftarrow$) part of the proof of Theorem 18, let us now consider the following example.

Example 4. Let us assume that the dove-tailing evaluation of the function g, depicted in Figure 19 above, produces the following list:

$G = [\langle m_8, 8\rangle, \langle m_{11}, 11\rangle, \langle m_8, 12\rangle, \langle m_{11}, 16\rangle, \langle m_8, 17\rangle, \langle m_{19}, 19\rangle, \ldots]$

This value of G means that during the dove-tailing evaluation
- at position 8 (and $12, 17, \ldots$) we have computed $g(2) = m_8 = m_{12} = m_{17} = \ldots$,
- at position 11 (and $16, \ldots$) we have computed $g(1) = m_{11} = m_{16} = \ldots$,
- at position 19 we have computed $g(4) = m_{19} = \ldots$, etc.

In our case we have that $m_{in} = m_8$ and $h(0) = m_8$. Thus, the values of the function h are as follows:

n:	0	1	2	...	8	9	10	11	12	...	15	16	17	18	19	...
$h(n)$:	m_8	m_8	m_8	...	m_8	m_8	m_8	m_{11}	m_8	...	m_8	m_{11}	m_8	m_8	m_{19}	...

□

Definition 20. A set $A \subseteq N$ is *recursive* iff there exists a total p.r.f., say f_A, such that $\forall x \in N.$ *if* $x \in A$ *then* $f_A(x) = 1$ *and if* $x \notin A$ *then* $f_A(x) = 0$.

It follows from the definition that for any given recursive set A the corresponding function f_A (which is a total p.r.f.) can be used for deciding whether or not a number is an element of A, that is, for deciding the membership problem relative to the set A.

By definition, recursive sets are the largest collection of sets for which it is possible to decide the membership problem via a total p.r.f.

Related to Definition 20 we also have the following theorem.

Theorem 19. A set A is r.e. iff there exists a partial recursive function, say f_A, such that: $\forall x \in N.$ *if* $x \in A$ *then* $f_A(x) = 1$ *else* undefined. (†)

Proof. Let us consider the partial recursive function p which we have constructed in the proof of part ($\Rightarrow$) of Theorem 17. We have that A is the domain of $p(x)$. Thus, $p(x) \in N$ iff $x \in A$, and if $x \notin a$ then $p(x)$ is undefined. Let us also consider the following function f_A:

$f_A(x) = nsg(0 \neg p(x))$.

Now we have that: (i) $f_A(x)$ is a p.r.f. because $p(x)$ is a p.r.f. and the class of the partial recursive functions are closed w.r.t. composition, and (ii) $f_A(x)$ satisfies (†) because by the properties of the function $p(x)$ we have that $f_A(x)=1$ iff $x \in A$, and if $x \notin A$ then f_A is undefined. (Recall that expressions are evaluated in the call-by-value mode.) □

There are sets which are r.e. and are not recursive. One such set can be constructed as follows.

Let us consider the following injective encoding, say τ^*, from the set of Pascal programs into the set N of natural numbers. Since every Pascal program is a finite sequence of characters, for each program there exists a unique corresponding finite sequence of numbers, and that sequence can be encoded into a single number (via, for instance, the above mentioned τ function).

Let us consider the set

$D = \{\tau^*(p) \mid p \text{ is a Pascal program and } p \text{ halts when given the input } \tau^*(p)\}.$

Thus, a number n is in D iff n is the encoding of a Pascal program which halts when its input is n. One can show that D is a r.e. subset of N, but it is not a recursive subset.

Obviously, instead of Pascal programs, we may equivalently consider C++C++ or Java programs. Indeed, the choice of the programming language does not matter as long as we can write in it a program for any partial recursive function.

15 Recursive Enumerable Sets and Turing Computable Functions

An alternative definition of recursively enumerable sets can be given in terms of Turing Machines, instead of p.r.f.'s. Indeed, as we will indicate in Section 17, it can be shown that a function is a p.r.f. from N to N iff it can be computed by a Turing Machine, having chosen a suitable encoding of numbers by strings of tape symbols.

One can show that r.e. sets are exactly those subsets of N which can be *enumerated* by Turing Machines in the following sense: for any r.e. set $A \subseteq N$ there exists a Turing Machine T_A such that if A is the empty subset of N then for any input $n \in N$, T_A does not halt, and if A is not empty then

(i) for any $n \in N$, when given n as input, the Turing Machine T_A halts and gives us back an element of A, and

(ii) for any a if $a \in A$ there exists $n \in N$ such that when given n as input, the Turing Machine T_A halts and gives us back the element a.

Notice that if a set A is r.e. there exists an infinite number of Turing Machines which enumerate A, and in particular, there exists an infinite number of Turing Machines which do not halt for all input $n \in N$.

16 Partial Recursive Functions on Strings

Instead of considering p.r.f.'s from N to N we may consider p.r.f.'s from a generic domain $M1$ to a generic domain $M2$. In particular, we may choose $M1 = M2 =$ the integers, or $M1 = M2 = \Sigma^* = \{a, b\}^*$. In the latter case the set of p.r.f.'s from $\bigcup_{k \geq 0} (\Sigma^*)^k$ to Σ^* can be defined as the smallest set of functions such that it includes:

(i.1) the functions of k arguments $\lambda s_1 \ldots s_k. \varepsilon$, where ε is the empty string, for all $k \geq 0$,

(i.2) the two unary successor functions $\lambda s.\, sa$ and $\lambda s.\, sb$, and

(i.3) the projection functions $\lambda s_1 \ldots s_k . s_i$ for all $k \geq 1$ and for all $i = 1, \ldots, k$,

and it is closed under

(ii.1) composition, defined as in Definition 14 (iv),

(ii.2) primitive recursion defined as follows:
if the functions h_a and h_b of $k{+}1$ arguments and a function g of $k{-}1$ arguments are p.r.f.'s for some $k \geq 1$, then also the function f of k arguments such that:

$$
\begin{aligned}
f(\varepsilon, s_2, \ldots, s_k) &= g(s_2, \ldots, s_k) \\
f(s_1 a,\, s_2, \ldots, s_k) &= h_a(s_1, s_2, \ldots, s_k,\, f(s_1, s_2, \ldots, s_k)) \\
f(s_1 b,\, s_2, \ldots, s_k) &= h_b(s_1, s_2, \ldots, s_k,\, f(s_1, s_2, \ldots, s_k))
\end{aligned}
$$

is a p.r.f., and

(ii.3) minimalization in the sense that given any *total* the function p.r.f. $\lambda xs.\, t(x, s)$ from $(\Sigma^*)^{k+1} \times \Sigma^*$ to Σ^*, then the p.r.f. of arity k of the form

$$\lambda x.(\mu s.[t(x, s) = \varepsilon])$$

where $x \in (\Sigma^*)^k$ and $s \in \Sigma^*$, belongs to the set of p.r.f.'s from $(\Sigma^*)^k$ to Σ^*. In order to apply the minimalization operator μ we need an ordering among the strings in Σ^* and we choose the canonical one, that is, $\varepsilon < a < b < aa < ab < ba < bb < aaa < \ldots$, which corresponds to the assumption that $a < b$.

It is not difficult to show that under suitable encodings of strings into numbers, the theory of the p.r.f.'s where each function is from $(\Sigma^*)^k$ to Σ^*, for some $k \geq 0$, is isomorphic to the theory of the p.r.f.'s where each function is from N to N.

17 Partial Recursive Functions and Turing Computable Functions

We have the following theorem which identifies the set of partial recursive functions and the set of Turing computable functions, when we represent the input/output numbers and the input/output strings as indicated in Definitions 9 and 10 of Section 2. In particular, any natural number n is represented by 01^n0 and for any $k \geq 0$, any k-tuple $\langle n_1, \ldots, n_k \rangle \in N^k$ is represented by $01^{n_1}0 \ldots 01^{n_k}0$.

Theorem 20. (i) For any $k\,(\geq 0)$, any Turing Machine M whose initial configuration is $q_0 0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0$, computes a partial recursive function from N^k to N, and any partial recursive function from N^k to N can be computed by a Turing Machine whose initial configuration is $q_0 0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0$.

(ii) For any $k\,(\geq 0)$ any Turing Machine M whose initial configuration is $q_0 w$ with $w \in (\Sigma_1^*)^k$, computes a partial recursive function f from $(\Sigma_1^*)^k$ to Σ_2^*, and any partial recursive function from $(\Sigma_1^*)^k$ to Σ_2^* can be computed by a Turing Machine whose initial configuration is $q_0 w$.

We do not give the proof of this Thereom here. Theorem 20 tells us that there exists a bijection between partial recursive functions and Turing Machines.

Theorem 20 also holds if instead of Turing Machines partial recursive functions, we respectively consider: (i) Turing Machines which always halt with a configuration which represents an output value according our conventions, and (ii) partial recursive functions which are total. Thus, the notion of a total partial recursive function is equivalent to the notion of an 'always terminating' Turing Machine.

As we will see later, a consequence of this fact is that the undecidability of the Halting Problem implies that there is no partial recursive function which given any partial recursive function f, tells us whether or not f is total.