Chapter 4

RAM Programs, Turing Machines, and the Partial Recursive Functions

4.1 Introduction

Anyone with some programming experience has an intuituive idea of the notion of "algorithm". Even Euclid's algorithm was called an algorithm long before the invention of modern computers. However, it was not until the 1930's that logicians such as Church, Gödel, Kleene, Turing, and Post, put forth formal definitions for the notions of *effective procedure*, *computable function*, and *algorithm*.

There are surprisingly many different formalizations of the notion of an algorithm. A remarkable fact is that all of these definitions have been shown to be equivalent, in the sense that a function computable in one of these formulations is also computable in all of the others. For most computer scientists, the notion of algorithm is synonymous with that of a program written in some general purpose programming language.

To be more accurate, an algorithm refers to a program that *halts* for all inputs. A program that halts for some inputs but diverges for others is called a *procedure*. One of the characterizations of the computable functions is that they are computed by programs written in a very simple programming language, the language of *RAM programs, also called Post machines*.

Another goal of the theory of computation is to explore the limitations of the computational power of programs. For example, one can ask whether there exists an algorithm that could be used as a debugging tool, to test whether any given program halts on any given input. Another useful program would be one to test whether any two given programs are equivalent for all inputs. As we shall see, such programs do not exist. We have stumbled upon some *undecidable problems*.

Why is a question undecidable, that is, not answerable by a program halting for all inputs? What power must a programming language (or formal system) have, in order that some questions about it are undecidable?

We shall be concerned with these issues as we develop a technical formulation of what is an algorithm.

Before embarking on an extensive study of notions such as algorithms and procedures, a few crucial remarks are in order. Firstly, a program is a *finite* object. It may use a very large amount of memory, but still a *bounded* amount. However, there is no bound on the size of data (strings) held in the registers used by programs. Secondly, all the programming languages under consideration have the property that programs can be effectively *coded* as strings or numbers. This means that there is an algorithm that assigns a code to each program, and conversely, that there is an algorithm that, given a purported code name, tells whether or not the code name represents a program, and if so, which program. For example, we shall see that RAM programs can be encoded as positive natural numbers.

Since we will be dealing with algorithms working on strings or natural numbers, we will have the ability to give as input to a program input data that stand either for a true data, or an encoding for a program. This situation is analogous to that in assembly languages, where a memory word either stands for a data or for an instruction, depending on its interpretation.

It turns out that it is the ability of encoding programs into numbers (or strings) and to decode numbers back into programs, that is often the cause for the undecidability of a question.

In the following Chapters, we study various algorithmic systems. We begin with RAM programs, and continue with Turing machines. It turns out that RAM programs and Turing machine compute precisely the same clas of (partial functions). This famous class of function is called the class of *partial recursive functions*.

RAM programs (flowchart form)

The instructions are the following:

1. START

2. Transfer statements: $y \leftarrow x$

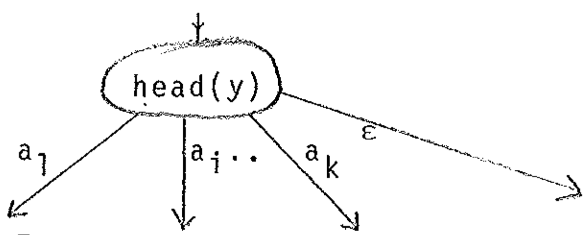
3. add statements: $y \leftarrow y\, a_j$

   where a_j is in Σ.

4. clear statements: $y \leftarrow \varepsilon$

5. delete statements: $y \leftarrow \text{tail}(y)$

6. test statements:



7. Stop

We define the functions head and tail as follows:

$$\mathrm{head}(\varepsilon) = \varepsilon$$

$$\mathrm{head}(a_i u) = a_i$$

$$\mathrm{tail}(\varepsilon)=\varepsilon$$

$$\mathrm{tail}(a_i u) = u$$

1.2.1 Definition A RAM flowchart program is a graph obtained by interconnecting statements in such a way that:

1) There is a single START

2) There is a single STOP

3) Every entry point of a statement is connected to an exit point of some statement and every exit point of a statement is connected to the entry point of some statement.

The flowchart below is a program to concatenate two strings x_1 and x_2. The output is returned in x_1. The variables x and y are "working variables". We usually assume that programs have input variables x_1, .., x_m and that programs return a single output in x_1. However, this is merely a convenience and we could allow programs returning more than one output. For our purposes, it will be necessary to also have a "linear representation" of our flowchart programs. We now proceed with the definition.

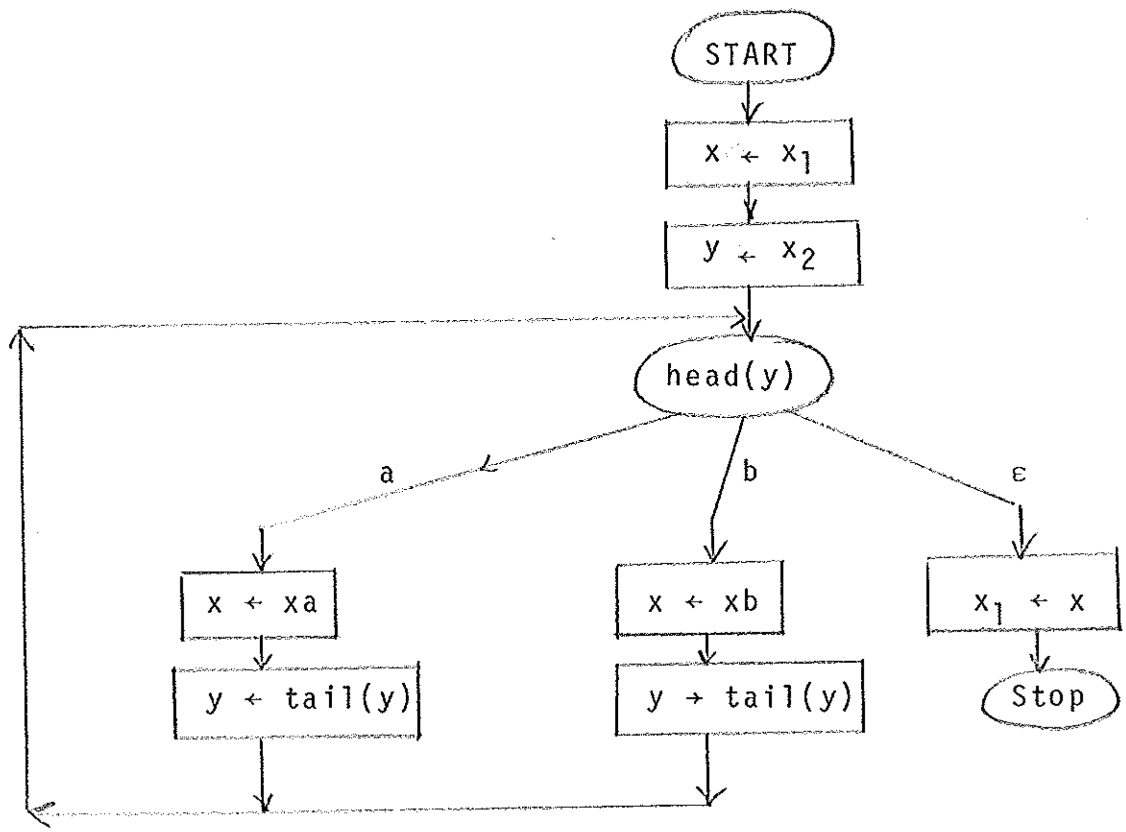
1.2.2 Definition RAM programs in linear form

RAM programs in linear form use a finite number of registers denoted R1, R2, Rn, and instructions may be labelled with line labels of the form N0, N1, .. Nk. Note that instructions do not have to be labelled, and that the same label can be reused in several places (thus, the term line label is rather unfortunate, but we will use it for the lack of a better name).

The allowable instructions are the following:

Example 1

Program to concatenate two strings x_1 and x_2 over $\{a,b\}^*$



1_j.	N	add_j Y
2.	N	del Y
3.	N	clr Y
4.	N	Y ← X
5.	N	jmp N'a or N jmp N'b
6_j.	N	Y jmp_j N'a or
	N	Y jmp_j N'b
7.	N	continue

N is a label or nothing.

X, Y are register names. N' is a label and a stands for above and b stands for below. The meaning of the instructions is as follows:

1_j corresponds to $y \leftarrow ya_j$

2 corresponds to $y \leftarrow \text{tail}(y)$

3 corresponds to $y \leftarrow \varepsilon$

4 corresponds to $y \leftarrow x$

5 is a jump statement (like a goto). Its effect is to transfer control to the closest instruction above labelled with the label N' in case we have jmp N'a, or transfer to the closest instruction below labelled N' in case we have jmp N'b.

6_j is a conditional jump.

A jump to the closest address labelled N' occurs depending on the suffix a or b, if and only if the head of register y is a_j. Otherwise, the next statement is executed.

Finally, continue is a no-op which does nothing.

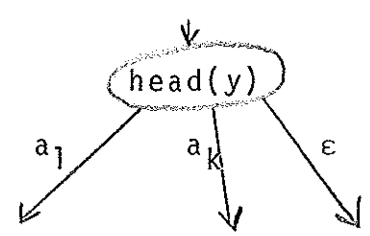
A RAM program is a finite sequence of instructions such that each jump has a target (e.g. if an instruction Y jmp_j N'b occurs in the program, some statement must be labelled N'), and the last instruction is a continue statement.

It may not be immediately clear to the reader why we allow the same label to be used in several places and why we are using the jump statements defined above. It is also not completely obvious that the flowchart form and the linear form are equivalent.

First, the reason for allowing multiple occurrences of labels is that we want to be able to concatenate programs without having to rename the labels. But then, to avoid ambiguities, we adopt jumps where the target address is relative to the address of the jump. In the present case, we jump to the closest address above or below. This may look a bit strange, but we shall see that this choice is actually very convenient later on in certain proofs.

As to the equivalence of the flowchart form and the linear form, we now sketch the proof. We first sketch the translation from a flowchart to a linear program.

First, we assign distinct labels to all the statements in the flowchart except START. Then we translate the flowchart into a linear program, starting from START. The translation is not necessarily unique but this doesn't matter. Clearly, one of the problems is to translate the tests of the form



Assume that the target labels are

N1, .., Nk, N(k+1).

Then we have the translation:

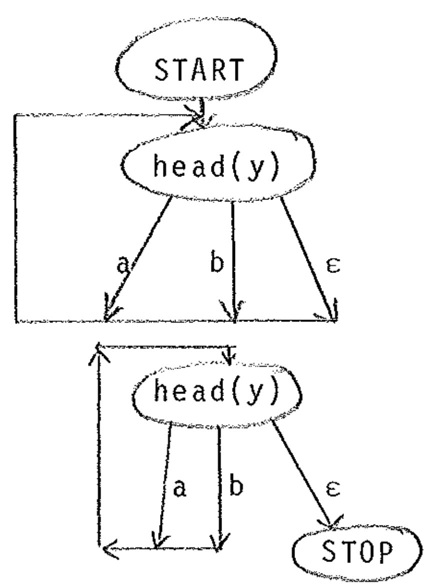
Y	jmp_1	N1c
.		
.		
Y	jmp_k	Nkc
Y	jmp	N(k+1)c

where c is either a or b, depending whether Ni already occurs above or not.

Note that it may also be necessary to use a number of additional (unconditional) jump statements to perform the translation correctly. We leave the details to the reader. Also, the last statement must be a continue.

Conversely, translating a linear program into a flowchart is fairly obvious and we leave the details to the reader.

A careful reader may have noticed that our definition of a RAM program, either in flowchart or linear form, does not exclude some rather strange programs which are not even connected such as:



We could fix the definition to avoid such cases, but such pathological cases will not be a problem so we don't go into this trouble now.

As an example of a program in linear form, the following program is a linear version of the flowchart of example 1.

Example 2:

		R3 ← R1
		R4 ← R2
N0	R4	jmp_a N1b
	R4	jmp_b N2b
		jmp N3b
N1		add_a R3
		del R4
		jmp N0a
N2		add_b R3
		del R4
		jmp N0a
N3		R1 ← R3
		continue

1.2.3 Definition

A program P computes the partial function ϕ if when the initial contents of the registers R1, R2, ..., Rn are $x_1, \ldots, x_n$, P eventually halts if and only if $\phi(x_1, \ldots, x_n)$ is defined, and if and when P halts, the final contents of R1 are $\phi(x_1, \ldots, x_n)$.

We say that a partial function ϕ is RAM-computable if some function computes it. For instance, the concatenation function is RAM computable.

Here are some other programs computing the functions E, S_j and P_i^n .

1. clr R1

   continue

2. add_j R1

   continue

3. R1 ← Ri

   continue

E is the erase function such that $E(x) = \varepsilon$ for all x.

S_j is the j-th successor function such that $S_j(x) = xa_j$ for all x. P_i^n is the projection function on the i-th coordinate such that

$P_i^n\ (x_1, \ldots, x_n) = x_i$ for $1 \le i \le n$. Note that P_1^1 is the identity function.

Now that we have a programming language, we would like to know how powerful it is, that is, we would like to know what kind of functions are RAM computable. At first glance, RAM program don't do much, but this is not so. Indeed, we will see shortly that the class of RAM-computable functions is very extensive.

One way of getting other programs from given ones is to compose them.

1.2.4 Definition

Let g be a function of $m \geq 1$ arguments and $h_1, \ldots, h_m$ be m functions each of $n \geq 1$ arguments.

The function f of $n \geq 1$ arguments also denoted $g\circ(h_1, \ldots, h_m)$ obtained from $h_1, \ldots, h_m$ and g by composition is the function such that for all $x_1, \ldots, x_n$,
$f(x_1, \ldots, x_n) = g(h_1(x_1,\ldots,x_n), \ldots, h_m(x_1, \ldots, x_n))$.

Note that if g or the h_i are partial functions, the function is defined for some $x_1, \ldots, x_n$ if and only if both all $h_i(x_1,\ldots,x_n)$ are defined and g is defined for $(h_1(x_1,\ldots,x_n), \ldots h_m(x_1,\ldots,x_n))$. Also, two partial function ϕ and ψ are equal if and only if for all $x_1, \ldots, x_n$, either both $\phi(x_1, \ldots x_n)$ and $\psi(x_1,\ldots,x_n)$ are undefined, or both are defined and $\phi(x_1,\ldots,x_n) = \psi(x_1,\ldots,x_n)$.

1.2.5 Proposition

If $\psi, \theta_1, \ldots, \theta_m$ are RAM-computable and ϕ is obtained from them by composition, then ϕ is RAM-computable.

Proof: Let $R, P_1, \ldots, P_m$ be programs computing $\psi, \theta_1, \ldots, \theta_m$. Let n be the number of arguments in the θ_i and ϕ. The idea is to use $P_1, \ldots, P_m$ as "subroutines" to R. Let q be the least integer greater than m and n and such that no register past Rq is used in $R, P_1, \ldots, P_m$.

Then, the following program computes ϕ.

```
Rq + 1 ← R1      ⎤
      .          ⎥
      .          ⎥ save inputs
      .          ⎥
Rq + n ← Rn      ⎦

clr     Rn+1     ⎤
      .          ⎥
      .          ⎥ initialize for P_1
      .          ⎥
clr     Rq       ⎦
       P_1      compute θ_1(x_1,...,x_n)

Rq + n+1 ← R1        store θ_1(x_1,...,x_n)
    .
    .
    .
R1 ← Rq+1      ⎤
               ⎥
.              ⎥
.              ⎥
               ⎥
Rn ← Rq+n      ⎥        initialize for P_m
clr  Rn+1      ⎥
               ⎥
.              ⎥
.              ⎥
.              ⎥
               ⎥
clr  Rq        ⎦
  P_m                    compute θ_m(x_1,...,x_n)
Rq+n+m <- R1  save  θ_m(x_1,...,x_n)
R_1  <- Rq+n+1  ⎤
    .           ⎥
    .           ⎥
    .           ⎥
Rm  <- Rq+n+m   ⎥     initialize for R
clr   Rm+1      ⎥
clr   Rq        ⎦     compute
 R                   φ(x_1, ..., x_n)
```

Now, the reader probably understands why we are using relative addresses in the jumps - this allows us to simply "plug in" the programs acting as subroutines in the right places. The other instructions simply make sure that programs are correctly initialized.

Suppose we want to write a program to compute the function $f(x_1, x_2) = x_1^{|x_2|}$, where $x_1^{|x_2|}$ denotes the string $\underbrace{x_1 \ldots x_1}_{|x_2| \text{ times}}$

It has a simple recursive definition.

Namely, $f(x_1,\varepsilon) = \varepsilon$ and

$$f(x_1,x_2a_i) = f(x_1,x_2)x_1$$

Using the concatenation function explicitely,

$f(x_1,x_2a_i) = con(f(x_1,x_2),x_1)$.

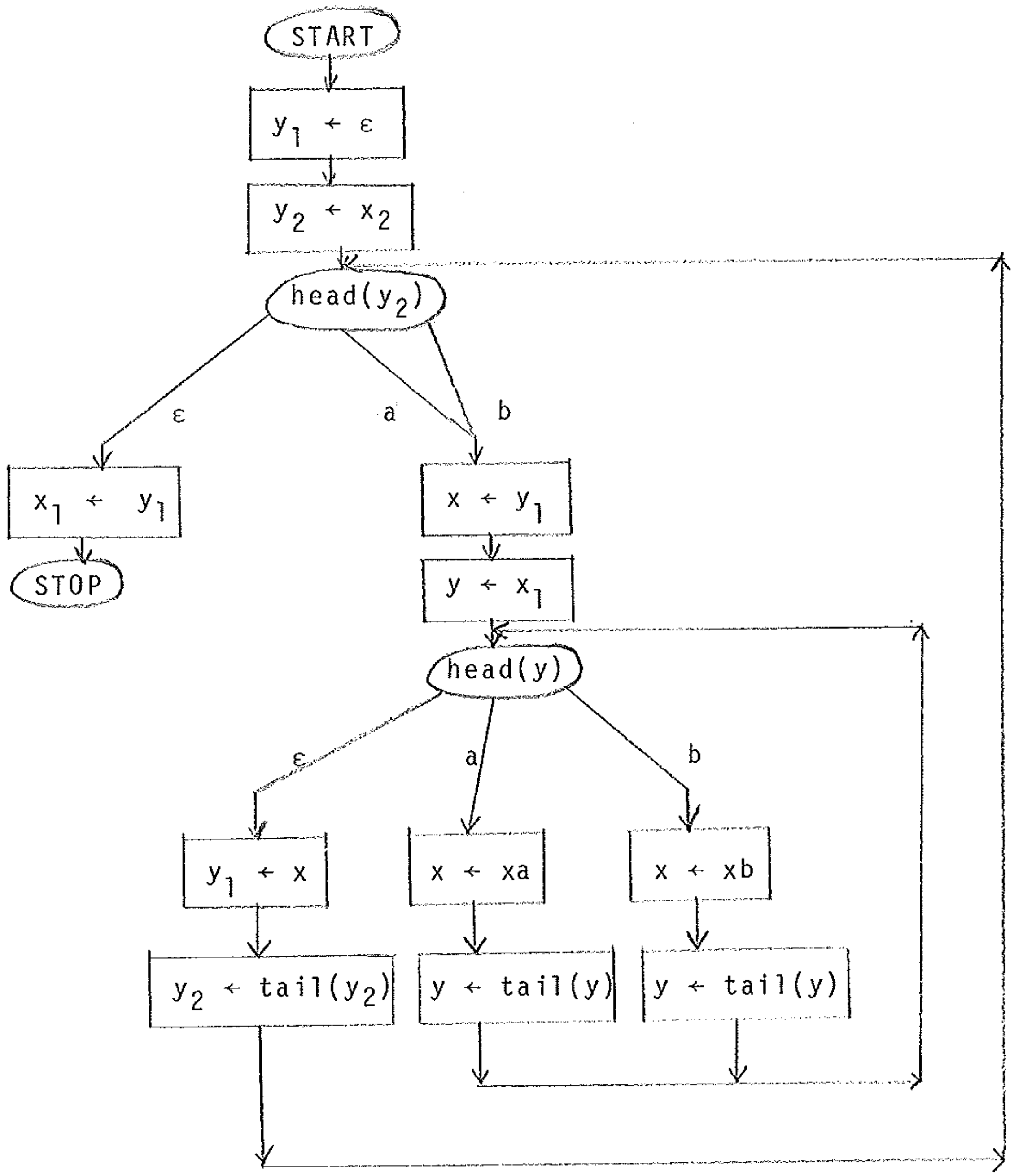
Since we already have a program to compute con, the problem boils down to expressing recursion. The following program computes f.

Examples: Program to compute

$$f(x_1, x_2) = x_1^{|x_2|}$$

$$f(x_1, \varepsilon) = \varepsilon$$

$$f(x_1, x_2 a_i) = con(f(x_1, x_2), x_1)$$



The type of recursive definition used above can be generalized as follows.

1.2.6 Definition

Let g be a function of n-1 arguments where $n \geq 2$ and let $h_1, \ldots, h_k$ be functions of n+1 arguments. The function f is obtained from g and $h_1, \ldots, h_k$ by primitive recursion if, for all y, for all $x_2, \ldots, x_n$ in Σ^*, (where

$\Sigma = \{a_1, \ldots, a_k\}$)

$$f(\varepsilon, x_2, \ldots, x_n) = g(x_2, \ldots, x_n)$$
$$f(ya_1, x_2, \ldots, x_n) = h_1(y, f(y, x_2, \ldots, x_n), x_2, \ldots, x_n)$$
$$f(ya_i, x_2, \ldots, x_n) = h_i(y, f(y, x_2, \ldots, x_n), x_2, \ldots, x_n)$$
$$f(ya_k, x_2, \ldots, x_n) = h_k(y, f(y, x_2, \ldots, x_n), x_2, \ldots, x_n)$$

If n = 1 the definition is:

$$f(\varepsilon) = u \text{ for some } u \in \Sigma^*$$
$$f(ya_i) = h_i(y, f(y)) \text{ for } 1 \leq i \leq k.$$

1.2.7 Proposition

If $\psi, \theta_1, \ldots, \theta_k$ are RAM-computable and ϕ is defined from them by primitive recursion, then ϕ is RAM-computable.

Proof: The key to the problem is that ϕ can actually be computed iteratively. Recall that ϕ is defined as follows:

$$\phi(\varepsilon, \bar{x}) = \psi(\bar{x})$$
$$\phi(ya_i, \bar{x}) = \theta_i(y, \phi(y, \bar{x}), \bar{x})$$

for $1 \leq i \leq k$, where $\bar{x}$ is an abbreviation for $(x_2, \ldots, x_n)$. Consider the following iterative sequence:

$$u_0 = \varepsilon \qquad v_0 = \psi(\bar{x})$$

$$u_1 = u_0 a_{i_1} \qquad v_1 = \theta_{i_1}(u_0, v_0, \bar{x})$$

$$\vdots$$

$$j \geq 1 \qquad u_j = u_{j-1} a_{i_j} \qquad v_j = \theta_{i_j}(u_{j-1}, v_{j-1}, \bar{x})$$

$$u_{m+1} = ya_i \qquad v_{m+1} = \theta_i(y, v_m, \bar{x})$$

where $\quad y = a_{i_1} \ldots a_{i_m}$.

It is easy to see that $v_j = \phi(u_j, \bar{x})$ and so

$$v_{m+1} = \phi(ya_i, \bar{x}).$$

The following program implements the above computation.

It is a good exercise to write the linear version of the program.

There is another operation which is analogous to a search process, under which the RAM programs are closed. This operation is similar to the while statement.

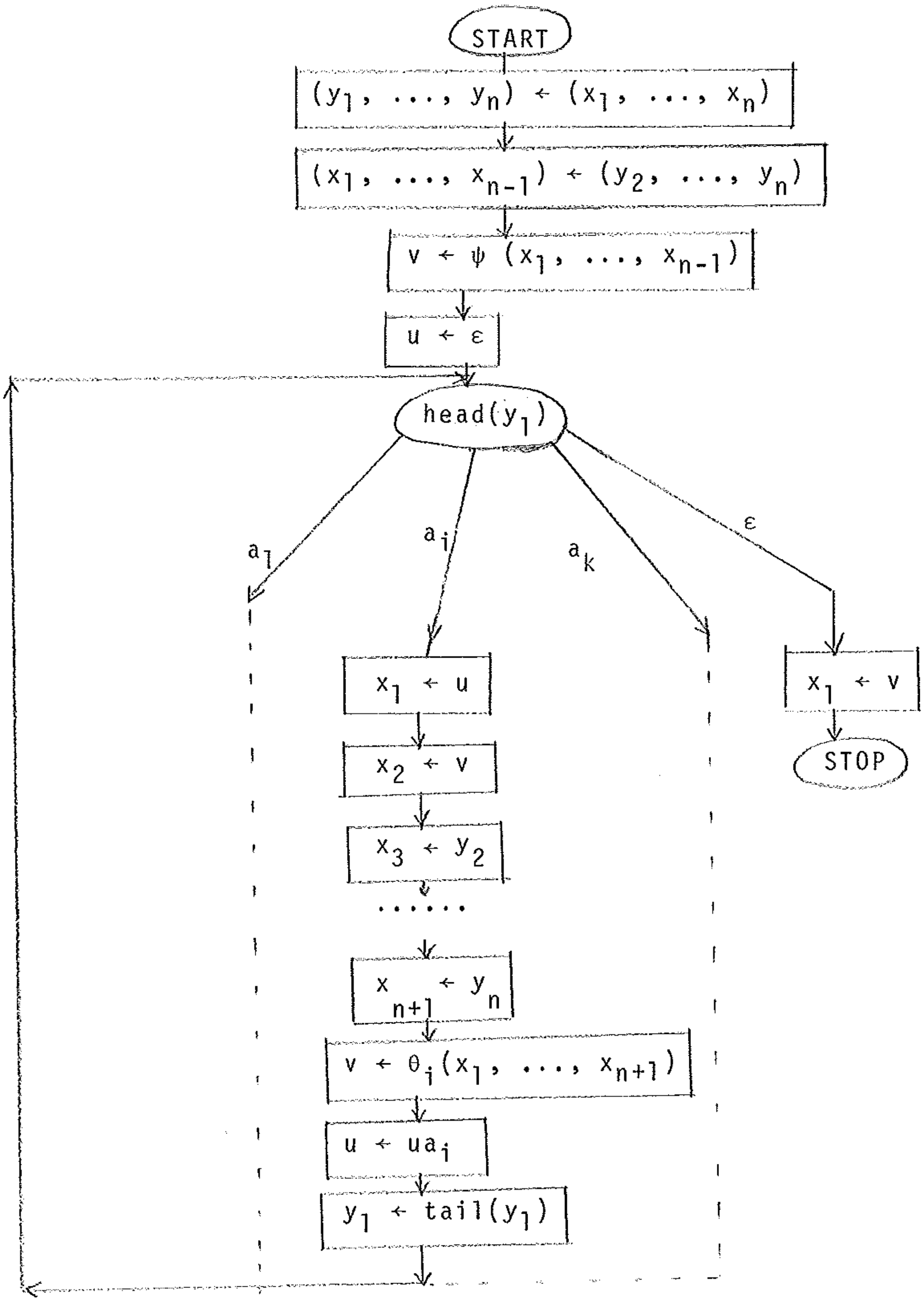
1.2.8 Definition

Let ψ be a given partial function of n arguments $y, x_1, \ldots x_{n-1} (n \geq 1)$.

ϕ is obtained from ψ by minimization over $\{a_j\}^*$ if for all $x_1, \ldots, x_{n-1}$:

1. $\phi(x_1, \ldots, x_{n-1})$ is defined if and only if there is an integer $m \geq 0$ such that for all p, $0 \leq p \leq m$, $\psi(a_j^p, x_1, \ldots, x_{n-1})$ is defined and $\psi(a_j^m, x_1, \ldots, x_{n-1}) = \varepsilon$
2. If m exists above, then it is the least integer q such that $\psi(a_j^q, x_1, \ldots, x_{n-1}) = \varepsilon$ and then $\phi(x_1, \ldots, x_{n-1}) = a_j^m$.

We write $\phi(x_1, \ldots, x_{n-1}) = \min_j y[\psi(y, x_1, \ldots, x_{n-1}) = \varepsilon]$.



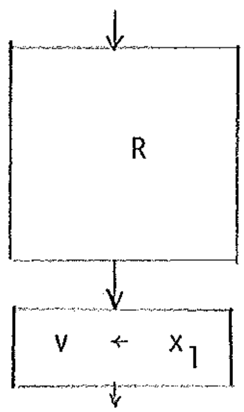
ϕ is defined by primitive recursion from ψ and $\theta_1, \ldots, \theta_k$

Note that the statements

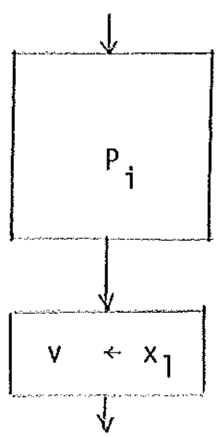
$v \leftarrow \psi(x_1, \ldots, x_{n-1})$ and

$v \leftarrow \theta_i(x_1, \ldots, x_{n+1})$

are really abbreviation for macrosubstitutions. If ψ is computed by the program R, $v \leftarrow \psi(x_1, \ldots, x_{n-1})$ stands for the piece of program



where it is assumed that the variables used by R (except $x_1, \ldots, x_{n-1}$) are not used elsewhere in the program. Similarly, $v \leftarrow \theta_i(x_1, \ldots, x_{n+1})$ stands for



where P_i computes θ_i.

1.2.9 <u>Proposition</u>

Let ψ be a RAM computable function. Then the function ϕ obtained from ψ by minimization over $\{a_j\}^*$ is RAM-computable.

<u>Proof:</u> The program below computes ϕ.

It is a good exercise to write the linear RAM program.

It will be useful later to know that the RAM's can be programmed with a smaller instruction set.

1.2.10 <u>Proposition</u>

Every RAM program can be effectively transformed into one which uses only instructions of the form

1. N add_j Y
2. N del Y
3. N Y jmp_j N'c
4. continue

and which computes the same function.

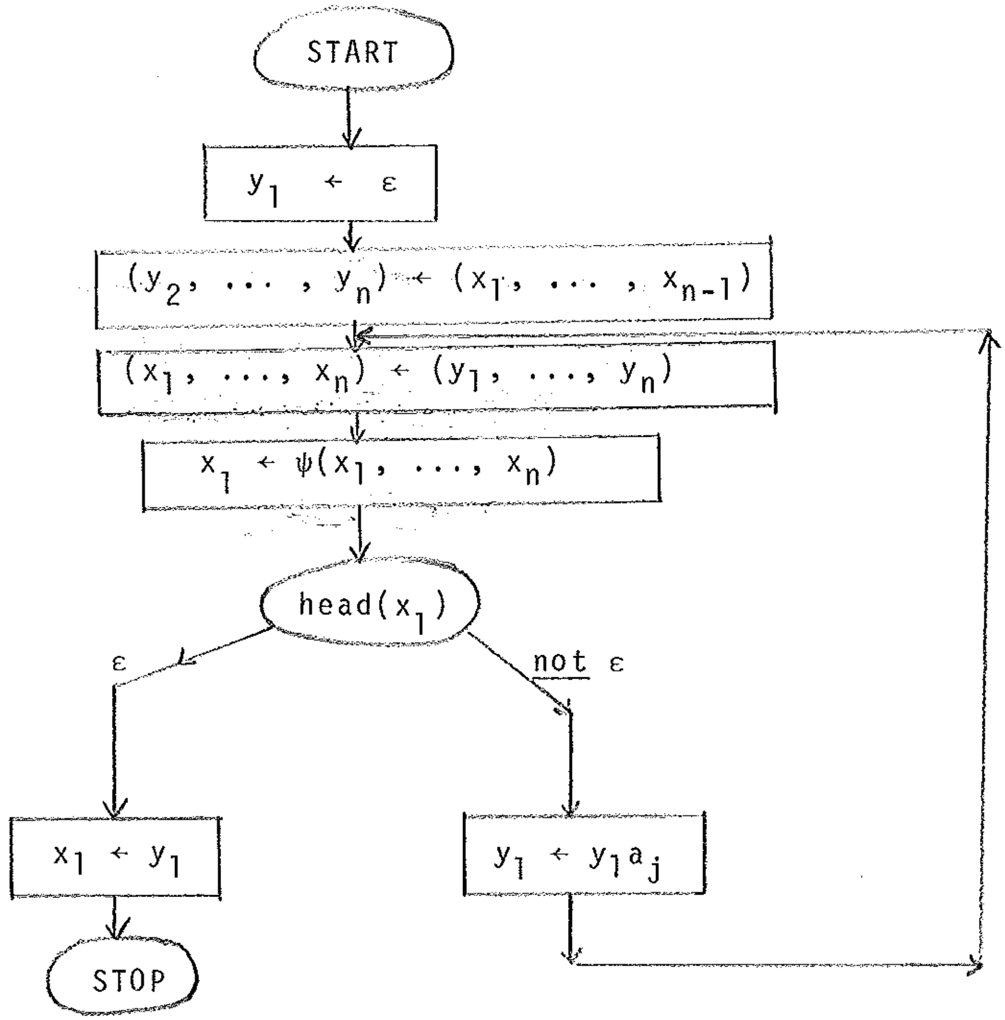
<u>Proof:</u> We first eliminate instructions of the form Rf $\leftarrow$ Rg. Any instruction Rf $\leftarrow$ Rf can be replaced by continue.

To replace Rf $\leftarrow$ Rg with $f \neq g$, we use a new register Rm not named in the original program, transfer Rg into Rm (destructively) and then transfer Rm into Rg and Rf (destructively).

We leave to the reader the elimination of the statements clr and jmp N'a or jmp N'b.

We now turn to a mathematical characterization of the effectively computable functions.

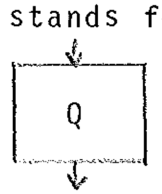
Minimization



We have $\phi(x_1, \dots, x_{n-1}) = \min_j y(\psi(y, x_1, \dots, x_{n-1}) = \varepsilon)$

ϕ is defined from ψ by minimization over $(a_j)^*$

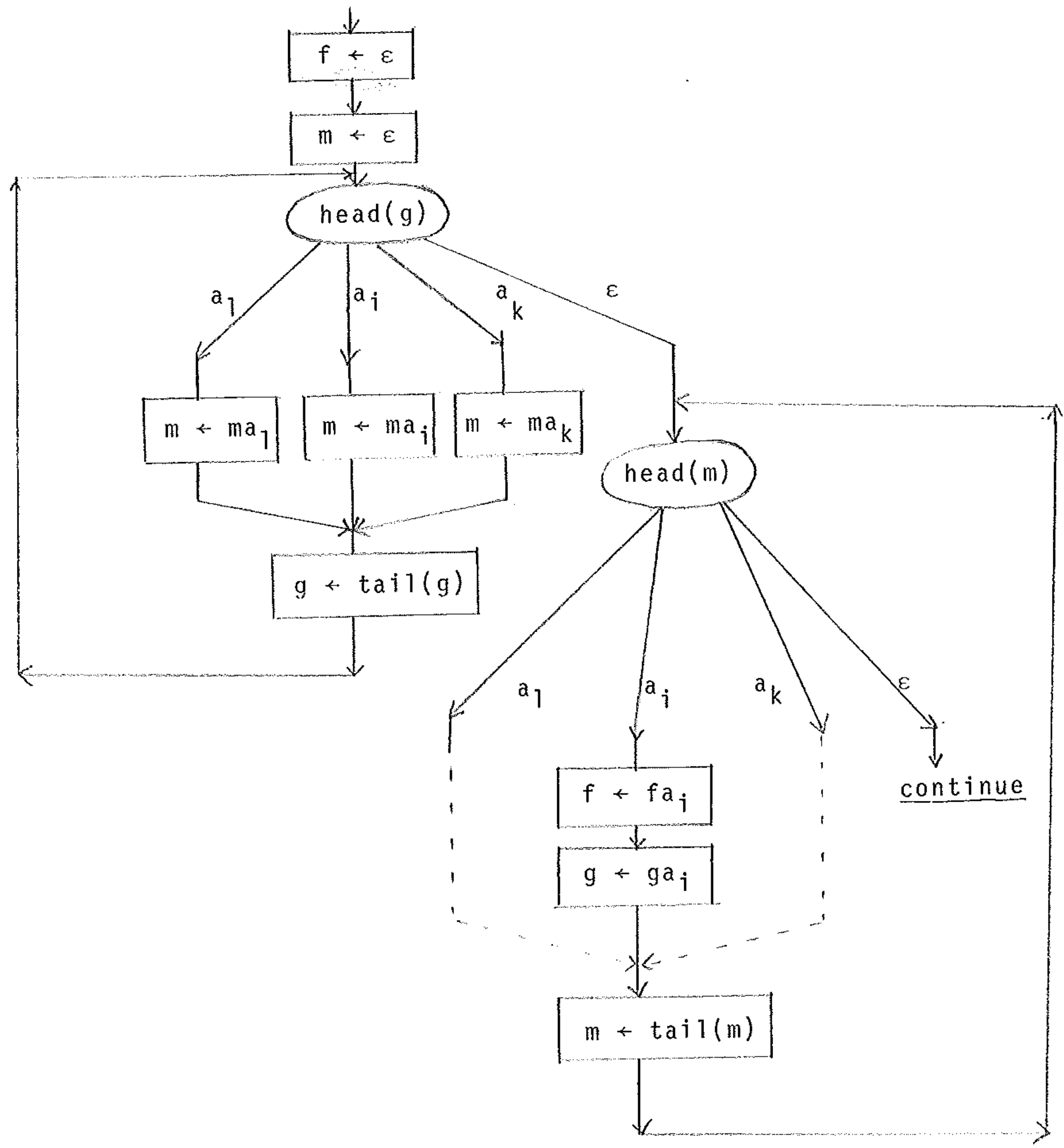
Note: $x_1 \leftarrow \psi(x_1, \dots, x_n)$ stands for the piece of program



where Q computes ψ.

Elimination of instructions $Rf \leftarrow Rg$

use auxiliary m.



```
        clr    Rf
        clr    Rm
        jmp    Nib
Nh      del    Rg
Ni      Rg     jmp_1    Nj_1b

        Rg     jmp_k    Nj_kb          copy Rg into Rm
        jmp    Nib
Nj_1    add_1  Rm
        jmp    Nha

N_jk    add_k  Rm
        jmp    Nha

Nh      del    Rm
Ni      Rm     jmp_1    N_j1 b

        Rm     jmp_k    N_jk b
        jmp    Nib                     copy Rm into
N_j1    add_1  Rf                      Rf and Rg
        add_1  Rg
        jmp    Nha

N_jk    add_k  Rf
        add_k  Rg
        jmp    Nha
Ni      continue
```

1.3 Primitive recursive functions

The idea is to define a set of functions by giving some set of basic functions and then some operations to compute new functions.

1.3.1 Definition

The class of primitive recursive functions over Σ^* is the smallest set of total functions over Σ^* containing the base functions defined below and closed under composition and primitive recursion.

I. Base functions:

1) The erase function E: $E(x) = \varepsilon$ for all x.

2) The j-th successor function S_j:
$S_j(x) = xa_j$ for all x.

3) The projection functions P_i^n:

$P_i^n(x_1, \ldots, x_n) = x_i$, for all $x_1, \ldots, x_n$, $1 \le i \le n$.

II. Composition

Given g a function of $m \ge 1$ arguments and $h_1, \ldots, h_m$ m functions of $n \ge 1$ arguments,

$$g\circ(h_1, \ldots, h_m)(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$$

III. Primitive recursion

Given a function g of n-1 arguments $n \geq 2$, and k functions $h_1, \ldots, h_k$ of n+1 arguments, f is defined by primitive recursion if for all y , $x_2, \ldots, x_n$ we have, abbreviating $\bar{x} = (x_2, \ldots, x_n)$:

$$f(\varepsilon,\bar{x}) = g(\bar{x})$$

$$f(ya_i, \bar{x}) = h_i(y,f(y,\bar{x}),\bar{x}) \quad 1 \leq i \leq k$$

If n = 1, then

$$f(\varepsilon) = u \text{ with } u\varepsilon\Sigma^*$$

$$f(ya_i) = h_i(y, f(y)) \quad 1 \leq i \leq k.$$

Proposition 1.3.2

Every primitive recursive function is RAM-computable.

Proof: We have shown that the base functions are RAM-computable and by proposition 1.2.5 and 1.2.7.

We now show that many usual functions are primitive recursive. Primitive recursive will be abbreviated p.r. .

(i) For each $u = a_{i_1} .. a_{i_n} \varepsilon \Sigma^*$, the constant function f_u such that $f_u(x) = u$ for all x is p.r. $f_u(x) = S_{i_n}(\ldots(S_{i_2}(S_{i_1}(E(x)))\ldots)$

To be rigorous, observe that $f_\varepsilon = E$ and $f_{ua_i} = S_i(f_u(x))$.

(ii) The concatenation function is p.r. . This is more tricky than it looks. For example,

$$\text{con } (x_1,\varepsilon) = P_1^1(x_1)$$

$$\text{con } (x_1,ya_i) = S_i(P_2^3(y,\text{con}(x_1,y),x_1))$$

is not legitimate.

Indeed recursion is only allowed in the first argument. The difficulty can be circumvented using the projection functions.

Define $\mathrm{con}'(\varepsilon,y) = P_1^1(y)$

$\mathrm{con}'(xa_i,y) = S_i(P_2^3(x,\mathrm{con}'(x,y),y))$

(Note that $\mathrm{con}'(x,y) = yx$)

Then, $\mathrm{con}(x,y) = \mathrm{con}'(P_2^2(x,y),P_1^2(x,y))$.

We also define the extended concatenation

$\mathrm{con}_{n+1}\ (x_1, \ldots, x_{n+1}) = x_1 \ldots x_{n+1}$.

$$\mathrm{con}_{n+1}(x_1, \ldots, x_{n+1}) = \mathrm{con}(\mathrm{con}_n(P_1^{n+1}(x_1, \ldots, x_{n+1}), \ldots P_n^{n+1}(x_1, \ldots, x_{n+1})), P_{n+1}^{n+1}(x_1, \ldots, x_{n+1}))$$

iii) $[x]^n = x^n$.

$[x]^n = \mathrm{con}_n\ (P_1^1(x), \ldots, P_1^1(x))$

(iv) The <u>delete</u> <u>last</u> function dell such that:

$\mathrm{dell}(\varepsilon) = \varepsilon$

$\mathrm{dell}(xa_i) = x$.

$\mathrm{dell}(\varepsilon) = \varepsilon$, $\mathrm{dell}(xa_i) = P_1^2(x, \mathrm{dell}(x))$.

(v) The <u>sign</u> function sg.

$$\mathrm{sg}(x) = \begin{cases} \varepsilon \text{ if } x = \varepsilon \\ a_1 \text{ if } x \neq \varepsilon \end{cases}$$

$\mathrm{sg}(\varepsilon) = \varepsilon$, $\mathrm{sg}(xa_i) = f_{a_1}(P_1^2\ (x, \mathrm{sg}(x)))$

(vi) The function $\overline{\mathrm{sg}}$ such that:

$$\overline{\mathrm{sg}}(x) = \begin{cases} a_1 \text{ if } x = \varepsilon \\ \varepsilon \text{ if } x \neq \varepsilon \end{cases}$$

(vii) end_j such that:

$$\mathrm{end}_j(x) = \begin{cases} a_1 \text{ if } x \text{ ends in } a_j \\ \varepsilon \text{ otherwise} \end{cases}$$

(viii) the reverse function rev.

(ix) the tail function such that:

$$\mathrm{tail}(\varepsilon) = \varepsilon$$

$$\mathrm{tail}(a_i x) = x$$

(x) The last function such that:

$$\mathrm{last}(\varepsilon) = \varepsilon$$

$$\mathrm{last}(x a_i) = a_i$$

(xi) The head function such that:

$$\mathrm{head}(\varepsilon) = \varepsilon$$

$$\mathrm{head}(a_i x) = a_i$$

(xii) The function $x-y$ such that:

$x-y = \varepsilon$ if $|x| \leq |y|$, and x minus its first $|y|$ letters if $|x| > |y|$

(xiii) The cond function such that:

$$\mathrm{cond}(x,y,z) = \underline{\text{if}}\ x \neq \varepsilon\ \underline{\text{then}}\ y\ \underline{\text{else}}\ z$$

Exercise: Prove that the functions in (vi) - (xiii) are primitive recursive.

Another useful way of defining primitive recursive functions is to use primitive recursive predicates.

1.3.3.3 Definition

An n-ary predicate P (over Σ^*) is a subset of $(\Sigma^*)^n$. We write a predicate either as $(x_1, \ldots, x_n) \in P$ or as $P(x_1, \ldots, x_n)$. The characteristic function of a predicate P is the function C_P such that:

$$C_P(x_1, \ldots, x_n) = \begin{cases} a_1 & \text{iff } P(x_1, \ldots, x_n) \\ \varepsilon & \text{iff } \underline{\text{not}}\ P(x_1, \ldots, x_n) \end{cases}$$

A predicate P is primitive recursive iff its characteristic function is primitive recursive.

We can take Boolean combinations of predicates:

P <u>or</u> Q, P <u>and</u> Q, <u>not</u> P.

1.3.4 <u>Proposition</u> If P, Q are primitive recursive predicates, then so are <u>not</u> P, P <u>or</u> Q, P <u>and</u> Q

<u>Proof</u>:

$$C_{\underline{\text{not}}\ P}(\bar{x}) = \overline{sg}\,(C_P(\bar{x}))$$

$$C_{P\ \underline{\text{or}}\ Q} = sg\,(con(C_P(\bar{x}), C_Q(\bar{x}))$$

$$C_{P\ \underline{\text{and}}\ Q} = dell\,(con(C_P(\bar{x}), C_Q(\bar{x}))$$

It is also useful to be able to define functions by cases as shown below:

1.3.5 <u>Proposition</u>

If $P_1, \ldots, P_n$ are pairwise disjoint primitive recursive predicates and $f_1, \ldots, f_n, f_{n+1}$ are primitive recursive functions, then the function g defined below is primitive recursive.

$$g(\bar{x}) = \begin{cases} f_1(\bar{x}) & \text{iff } P_1(\bar{x}) \\ f_n(\bar{x}) & \text{iff } P_n(\bar{x}) \\ f_{n+1}(\bar{x}) & \text{otherwise} \end{cases}$$

Proof: We do the proof when n = 2 and there is only one argument, but the proof extends immediately.

Define the function # such that

$$x \# y = \begin{cases} \varepsilon \text{ if } x = \varepsilon \\ y \text{ if } x \neq \varepsilon \end{cases}$$

is p.r since: $\varepsilon \# y = E(y)$ and

$$xa_i \# y = P_3^3(x, x \# y, y)$$

Then $g(x) = con_3(C_{P_1}(x) \# f_1(x), C_{P_2}(x) \# f_2(x),$

$$C_{\underline{not}(P_1 \ \underline{or}\ P_2)}(x) \# f_3(x))$$

We also define the application of "bounded quantifiers" to predicates.

1.3.6 Definition (bounded quantification)

If P is a predicate with n+1 arguments, then

$\exists y/x\ P(y,\bar{z})$ holds if and only if some prefix y of x makes $P(y,\bar{z})$ true.

$\forall y/x\ P(y, \bar{z})$ holds if and only if all prefixes y of x make $P(y,\bar{z})$ true

$(\bar{z} = (z_1, \ldots, z_n))$.

1.3.7 Proposition

If P is a primitive recursive predicate, then $\exists y/x\ P$ and $\forall y/x\ P$ are primitive recursive predicates.

<u>Proofs:</u> Let $C_{\exists/P}$ be the characteristic function of $\exists y/xP$.

Then:

$$C_{\exists/P}(\varepsilon,\bar{z}) = C_P(\varepsilon,\bar{z})$$

$$C_{\exists/P}(xa_i,\bar{z}) = sg(con(C_{\exists/P}(x,\bar{z}),\ C_P(xa_i,\bar{z})))$$

This says that $\exists y/\varepsilon\ P(y,\bar{z})$ iff

$P(\varepsilon,\bar{z})$ and

$\exists y/xa_i P(y,\bar{z})$ iff $\exists y/x\ P(y,\bar{z})$ <u>or</u>

$P(xa_i,\ \bar{z})$.

Also, $\forall\ y/x\ P(y,\bar{z})$ iff <u>not</u> $\exists y/x$ <u>not</u> $P(y,\bar{z})$.

As an application, we show that the equality predicate is p.r. First, the predicate end(x) = end(y) is p.r. because:

$end(\varepsilon) = end(y)$ iff $y=\varepsilon$

$end(xa_i) = end(y)$ iff $end_i(y)$

$|x| = |y|$ is p.r. because $|x| = |y|$ iff

$x \dot{-} y = \varepsilon$ and $y \dot{-} x = \varepsilon$.

Finally, $x = y$ iff

$|x| = |y|$ and $\forall z/x\ [end(z) = end(rev(rev(y) \dot{-} (x \dot{-} z)))]$. This reads: x and y have the same length, and every prefix of x ends in the same symbol as the corresponding prefix of same length in y.

The following propositions are very useful and are left as exercises.

1.3.8 Proposition

The predicate $\exists y \leq x\ P$ defined such that $\exists y \leq x\ P(y,\bar{z})$ holds iff there is some y such that $|y| \leq |x|$ and $P(y,\bar{z})$ holds is primitive recursive.

1.3.9 Proposition (bounded minimization)

The function min $y/x\ P$ defined such that:
min $y/x\ P(y,\bar{z})$ is the shortest prefix y of x such that $P(y,\bar{z})$ holds if y exists, and xa_1 otherwise is p.r.
Similarly, max $y/x\ P(y,\bar{z})$ is the longest prefix y of x such that $P(y,\bar{z})$ holds and xa_1 otherwise is p.r. This last operation is called bounded maximization.

The following propositions are quite useful to simplify the definition of primitive recursive functions. They show that arguments can be permuted, identified, replaced by constants, or that apparent variables can be adjoined.

1.3.10 Proposition

Let f be a p.r function of $n \geq 1$ arguments. Let $\pi: \{1, \ldots, n\} \to \{1, \ldots, n\}$ be a permutation of $\{1, \ldots n\}$
Then the function g such that $g(x_1, \ldots, x_n) = f(x_{\pi(1)}, \ldots x_{\pi(n)})$ for all $x_1, \ldots, x_n$ is p.r.

Proof: $g = f\mathrm{o}(P^n_{\pi(1)}, \ldots, P^n_{\pi(n)})$

1.3.11 Proposition

Let f be a p.r. function of $n \geq 2$ arguments. Then the function g of n-1 arguments such that

$g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, x_1)$ for all

$x_1, \ldots, x_{n-1}$ is p.r.

Proof: $g = fo\,(P_1^{n-1}, \ldots, P_{n-1}^{n-1}, P_1^{n-1})$

Also the function

$g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, u)$ with $u \in \Sigma^*$ is p.r.

Proof: We know that the function of one argument f_u such that $f_u(x) = u$ for all x is p.r. Let $f_u^n(x_1, \ldots, x_n) = f_u(P_1^n(x_1, \ldots x_n))$. Then $g = fo(P_1^{n-1}, \ldots, P_{n-1}^{n-1}, f_u^{n-1})$

1.3.12 Proposition

Let g be a p.r. function of $n \geq 1$ arguments. Then the function f of n+1 arguments such that $f(x_1, \ldots, x_{n+1}) = g(x_1, \ldots x_n)$ for all $x_1, \ldots, x_{n+1}$ is p.r.

Proof: $f = go(P_1^{n+1}, \ldots, P_n^{n+1})$.

Finally, we leave to the reader to show that primitive recursive definitions on any variable (not just the first one) are allowable and that in primitive recursive definitions,

$f(ya_i, x_2, \ldots, x_n) = h_i(y, f(y, x_2, \ldots, x_n), x_2, \ldots x_n)$

some or all of the arguments may be missing ($f(y, x_2, \ldots, x_n)$ is considered as an argument in itself) or permuted.

1.4 The partial recursive functions

It should be noted that all primitive recursive functions are total. This results from standard set theoretic arguments in the case of recursion. Now, it is easy to show that there are countably many primitive recursive functions. Consequently, they can be enumerated. Actually, it can be shown that they can be enumerated by a recursive function. However, they cannot be enumerated by a primitive recursive function.

We can prove a stronger result. Let A be any countable set of <u>total</u> functions containing the base functions and closed under composition and primitive recursion (then A contains all the primitive recursive functions). We say that a function f of 2 arguments is <u>universal</u> for the one-argument functions in A, if for every one-argument function g in A, there exists some $n \in N$ such that $f(a_1^n, u) = g(u)$ for all $u \in \Sigma^*$.

1.4.1 Proposition

If A is any set of <u>total</u> functions containing the base functions and closed under composition and primitive recursion, if f is a universal function for all the one-argument functions in A, then f is <u>not</u> in A.

<u>Proof</u>: Assume the universal function f is in A.
Let $g(u) = f(a_1{}^{|u|}, u) \cdot a_1$ for all $u \in \Sigma^*$. We claim that g is in A. It suffices to show that the function $h(u) = a_1^{|u|}$ is primitive recursive, which is left as an exercise.

But then, there is some m such that $g(u) = f(a_1^m, u)$ for all $u \in \Sigma^*$. Let $u = a_1^m$. Then $g(a_1^m) = f(a_1^m, a_1^m)$ = (by definition of g) $f(a_1^m, a_1^m).a_1$, a contradiction.

The above theorem shows that if we restrict ourselves to total functions, then either universal functions do not exist or else they cannot be total. This suggests to consider partial functions in order to expand our horizon. Note that this also corresponds to our intuitive idea of an effectively computable function: The program computing a function may diverge for some input. It will turn out that universal functions do exist, but they are only partial functions from the above theorem.

1.4.2 Definition The class of partial recursive functions (over Σ^*) is the smallest class of partial functions containing the base functions E, S_j, $1 \le j \le k$ and P_i^n, $1 \le i \le n$ and closed under the following operations:

I) Composition extended to partial functions

II) Primitive recursion extended to partial function.

III) Minimization.

Recall that if ψ is a given partial function of n+1 arguments, than ϕ is obtained from ψ by minimization over $\{a_j\}^*$ if for all $x_1, \ldots, x_n$:

1) $\phi(x_1, \ldots, x_n)$ is defined iff there exists an $m \in N$ such that for all p, $0 \le p \le m$, $\psi(a_j^p, x_1, \ldots, x_n)$ is defined and

$\psi(a_j^m, x_1, \ldots, x_n) = \varepsilon$

2) If such an m exists, then it is the least integer q such that

$$\psi(a_j^q, x, \ldots, x_n) = \varepsilon \text{ and } \phi(x_1, \ldots, x_n) = a_j^m.$$

We write

$$\phi(x_1, \ldots, x_n) = \min_j y\, [\psi(y, x_1, \ldots, x_n) = \varepsilon]$$

A partial recursive function which is total is called a recursive function. Also, a predicate whose characteristic function is recursive is called a recursive predicate.

It is important to realize that condition 1 cannot be relaxed to:

1'. There exists some m such that

$$\psi(a_j^m, x_1, \ldots, x_n) = \varepsilon.$$

The problem is that this does not preclude the existence of some $p < m$ for which $\psi(a_j^p, x_1, \ldots, x_n)$ diverges. We leave as an exercise to prove that functions which are <u>not</u> partial recursive can be obtained if 1 is relaxed to 1'.

Using propositions 1.2.9 and 1.3.2 we have the important Theorem:

1.4.3 <u>Theorem</u>

Every partial recursive function is RAM-computable.

The converse is also true and will be established later. The following proposition is quite obvious.

1.4.4 <u>Proposition</u>

Every primitive recursive function is a total recursive function. The recursive predicates are closed under the Boolean operations <u>and</u>, <u>or</u>, <u>not</u> and under bounded quantification (see 1.3.6). The total recursive functions are closed under definitions by cases 1.3.5 and bounded minimization (1.3.9) (and bounded maximization).

We now turn to Turing machines.

1.5 Turing Machines

A Turing machine is an idealized computer whose memory is a two-way infinite tape divided into squares or "tape symbols", and has a finite state control consisting of internal states. It has a read-write head which can move along the tape, scanning a single square at any given time. A Turing machine uses a finite set of tape symbols. The operations of a Turing machine are the following:

1) erase the symbol under the head and print a new symbol (overprint);
2) move right or move left one square on the tape;
3) change state.
4) halt.

Formalisms for Turing machines differ a great deal with no difference in computing power (unless one is concerned with the "complexity" of a computation:

number of steps, number of squares visited, etc...). We describe a Turing machine in quintuple form.

Turing machines are defined over a specified finite set of symbols which will be called the tape alphabet and is denoted Γ. Each such alphabet contains a distinguished symbol called the "blank" symbol and is denoted B. We often assume that Γ consists of two kinds of symbols: the input symbols form an alphabet $\Sigma \subset \Gamma$ and the "working" symbols which belong to $\Gamma - \Sigma$. Note that B is in $\Gamma - \Sigma$. At any given time, the tape contains only a finite number of squares being non blank. We assume that $\Gamma = \{a_1, \ldots a_k,$ "B"$\}$ where B is the blank symbol.

1.5.1 Definition

A Turing machine is a quintuple $M = (K,\Gamma,\Delta,\delta,q_0)$ where,

- K is a finite set of states
- Γ is a finite alphabet with blank B
- $\Delta = \Gamma \cup \{L,R\}$ where L and R are two symbols not in K or Γ
- δ is a set of quintuples, that is a subset of $K \times \Gamma \times \Gamma \times \{R,L\} \times K$ which is weakly deterministic in the first two components: For all $(p,a) \in K \times \Gamma$, there is at most one triple $(b,m,q) \in \Gamma \times \{L,R\} \times K$.
- q_0 is a distinguished state called the initial state.

The action of a Turing Machine on some input is described using the notion of an instantaneous description (ID).

1.5.2 Definition

An instantaneous description (ID) relative to a Turing machine M is a word in $\Gamma^* K \Gamma^+$, that is, a word of the form uqav where u, v are (possibly null) strings in Γ^*; q is a state in K and a is a symbol in Γ. (Note, a is not the null string).

Intuitively, instead of viewing the tape as an infinite tape, we can view it as a finite tape succeptible to extend itself at either ends during a computation. Then if a Turing Machine M is in an ID uqav, this means that it is currently in state q, scanning the tape symbol a under its reading head, and that its current tape contents is uav.

We now define the action of a Turing Machine by describing how instantaneous descriptions are modified in a single-step move.

1.5.3 Definition

Let M be a Turing machine. Let ID_1 and ID_2 be two ID's for M. We say that ID_1 yields ID_2 denoted $ID_1 \to ID_2$ (or M moves from ID_1 to ID_2) if the following conditions hold, where $a,b,c \in \Gamma$, $p,q \in K$, $u,v \in \Gamma^*$.

(1) (i) $(p,a,b,R,q) \in \delta$ and $ID_1 = upav$, $v \neq \varepsilon$, $ID_2 = ubqv$ or

(ii) $ID_1 = upa$, $ID_2 = ubqB$

(2) (i) $(p,a,b,L,q) \in \delta$ and $ID_1 = ucpav$, $c \in \Gamma$, $ID_2 = uqcbv$ or

(ii) $ID_2 = pav$, $ID_2 = qBbv$

Note that in (1) (ii) and (2) (ii), it is necessary to "extend" the tape with a blank to prevent the reading head from "falling off" the tape.

Sometimes, in defining Turing Machines moves in which the position of the head remains unchanged are allowed. In this case, in addition to L and R, we are allowed to use N in an instruction (p,a,b,N,q) having the following effect on ID's:

If $ID_1 = upav$ then $ID_2 = uqbv$.

We leave as an exercise to the reader to prove that such instructions although convenient are not needed.

We now explain how a Turing machine computes a partial function ϕ, where ϕ: $(\Sigma^*)^n \to \Sigma^*$. We assume that Turing machines computing a function over Σ have a tape alphabet Γ such that $\Sigma \subset \Gamma$, $B \notin \Sigma$ and Γ also contains the special symbol comma (,).

First, we define a computation and a halting ID.

1.5.4 Definition

An ID upav is a halting ID if there is no quintuple (p,a,b,m,q) in δ starting with p and a. This is also called a "blocking" ID.

A computation of M is a sequence of ID's ID_0, $ID_1, \ldots$ such that either

(1) the sequence is finite, its last member ID_n is a halting ID, and $ID_{i-1} \rightarrow ID_i$ for all i, $0 \leq i \leq n-1$ or

(2) the sequence is infinite and $ID_{i-1} \rightarrow ID_i$ for all $i \geq 1$.

This is a infinite, or diverging computation. A starting ID is an ID of the form $q_0 au$ where q_0 is the initial state, $a \in \Gamma$, $u \in \Gamma^*$.

For ID's $ID_0, ID_1, \ldots, ID_n$, if $ID_{i-1} \rightarrow ID_i$ for all $i \geq 1$, we write $ID_0 \overset{*}{\rightarrow} ID_n$. This also includes the case $ID_0 \overset{*}{\rightarrow} ID_0$. Otherwise, we write $ID_0 \overset{+}{\rightarrow} ID_n$ $(n \geq 1)$.

1.5.5 Definition

Let ϕ be a partial function $\phi : (\Sigma^*)^n \rightarrow \Sigma^*$.

A Turing machine M computes ϕ if $\Sigma \subset \Gamma$, $B \notin \Sigma$, "," $\in \Gamma$ and:

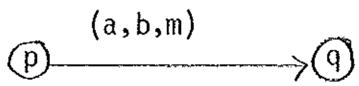
(1) for every input $(x_1, \ldots, x_n) \in \Sigma^*$, if M is started in a starting ID, $ID_0 = q_0\, x_1, x_2 \ldots, x_n$, $\phi(x_1, \ldots, x_n)$ is defined iff M reaches a halting ID of the form $B^k q \phi(x_1, \ldots, x_n) B^l$ for some state q in K and some integers $k, l \geq 0$.

(2) if $\phi(x_1, \ldots, x_n)$ is undefined, then either M halts in an ID not in the form described in (1) (that is, the output is "garbage"), or the computation does not halt.

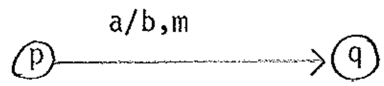
A halting ID of the form $B^k\, q\, u\, B^l$, $u \in \Sigma^*$, $k, l \geq 0$ is called proper. Otherwise, it is called improper.

Note that we are assuming in (1) that M "cleans up" its tape of all the working symbols used during the computation and returns the output string surrounded by a number of blanks (possibly null). This is not strictly necessary but it makes life easier. We say that ϕ is Turing-computable.

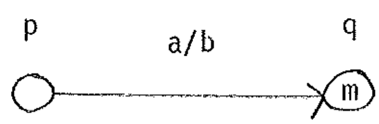
It is convenient to describe Turing machines using diagrams. We can use a labeled graph representation where each transition (p,a,b,m,q) is represented as:



or

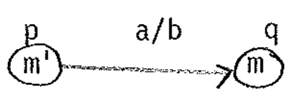


There is another convenient notation which can be used if for each state, all transitions entering that state cause the head to move in the same direction. If this condition is not satisfied, by splitting states, an equivalent Turing machine can be effectively constructed and we leave the construction as an exercise. The situation is now the following. If (p,a,b,m,q) ε δ we have the diagram:

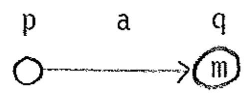


There is a slight problem if p is not entered by any transition. But then, either p is the initial state in which case we use the notation

or else state p is inaccessible and we can get rid of the quintuple starting with p. Otherwise, all transitions entering p cause the tape to move in the same direction m' and we write



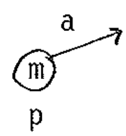
Further simplifications are possible. When no confusion arises, we can omit the state names. Transitions (p,a,a,m,q) are represented as



and

transitions (p,a,a,m,p) are simply omitted. In other words, "self loops" are omitted.

For all "blocking pairs" (p,a) such that no quintuple in δ begins with (p,a) we draw an outgoing arrow from state p labeled a.



Example: $M = (K, \Gamma, \Delta, \delta , q_0)$

$K = \{q_0, q_1, q_2, q_3\}$

$\Gamma = \{a, b, B\}$

δ is composed of the following quintuples:

(q_0, B, B, R, q_3)

(q_0, a, b, R, q_1)

(q_0, b, a, R, q_1)

(q_1, a, b, R, q_1)

(q_1, b, a, R, q_1)

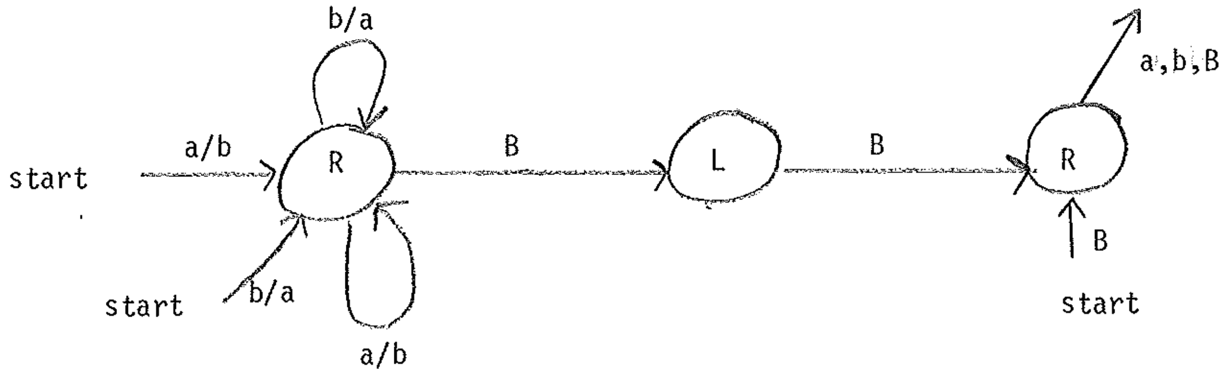
(q_1, B, B, L, q_2)

(q_2, a, a, L, q_2)

(q_2, b, b, L, q_2)

(q_2, B, B, R, q_3)

Modified diagram:



For any input u ε {a,b}*, the output of the computation is the string v obtained from u by changing each "a" into a "b" and each "b" into a "a".

We now prove that every RAM - computable function is Turing-computable.

1.5.6 Theorem

Every RAM-computable function is Turing - computable. Furthermore, given a RAM program, we can effectively find a Turing machine which computes the same function.

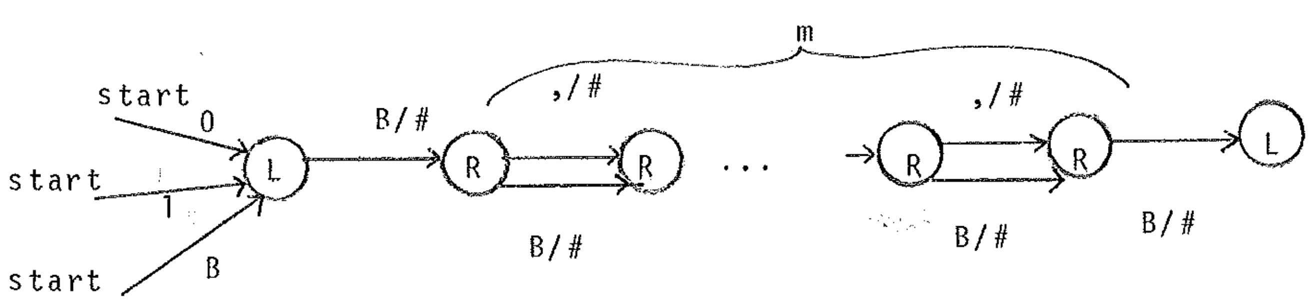
Proof: Let P be a RAM program using exactly m registers R1, ..., Rm and having n instructions. The contents r1, ..., rm of the registers will be represented on the Turing machine tape by the string #r1#r2# ... #rm#, where # is a special marker. Recall that we may assume that RAM programs use only instructions of the form:

add_j Y, del Y, Y jmp_j N'a (or N'b) and continue.

The simulating Turing machine M is built of n blocks connected for the same "flow of control" as the n instructions in P. The j-th block of the Turing machine simulates the j-th instruction in P.

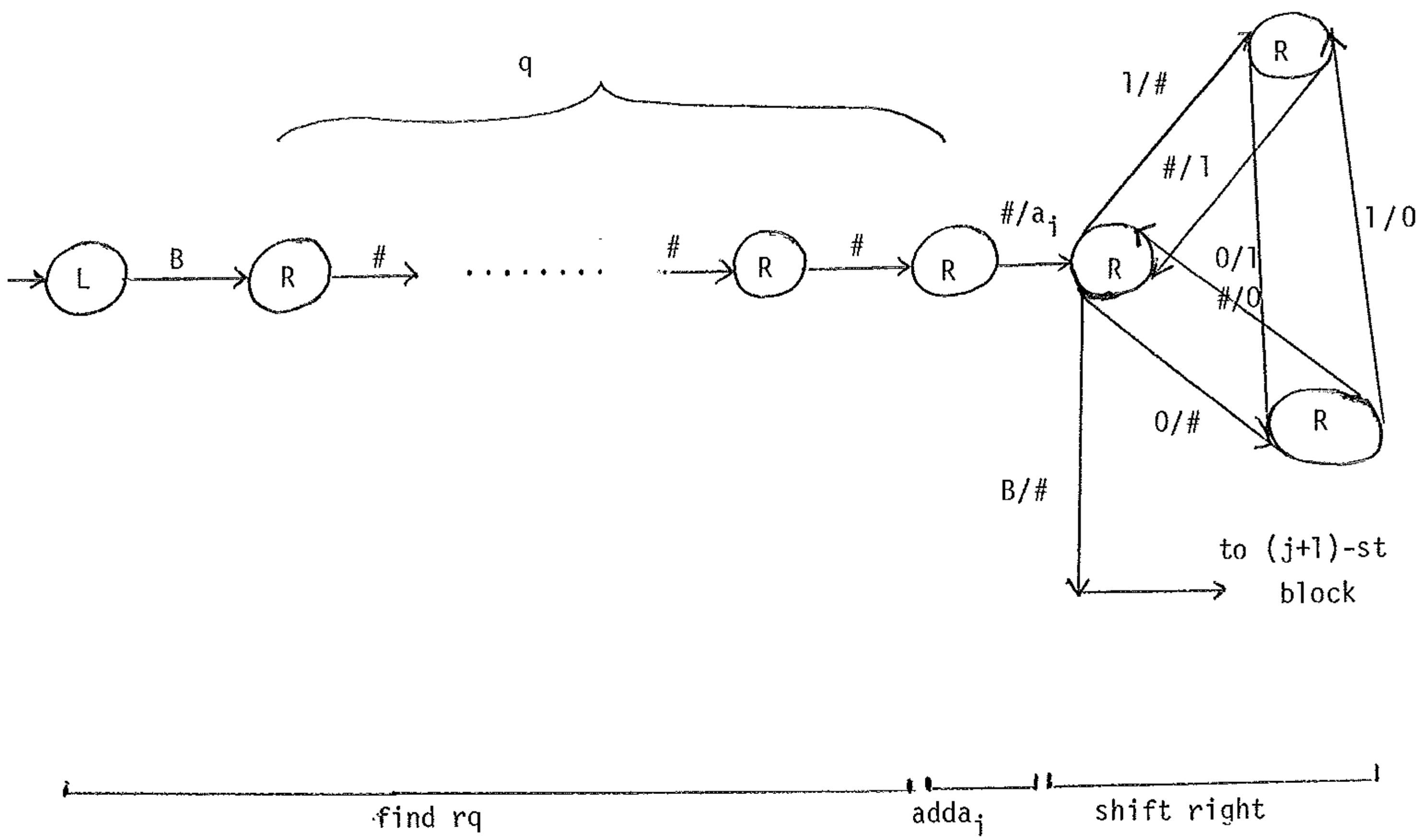
The machine M begins with some initialization, whose purpose is to make sure that the simulation starts with a tape representing m registers. Indeed, the RAM program P could have a number of input variables $t < m$, and it is necessary to add m+2-t symbols # to the input string. Also, since the input is $x_1, x_2, \ldots, x_t$ commas have to be changed to #.

Initialization:

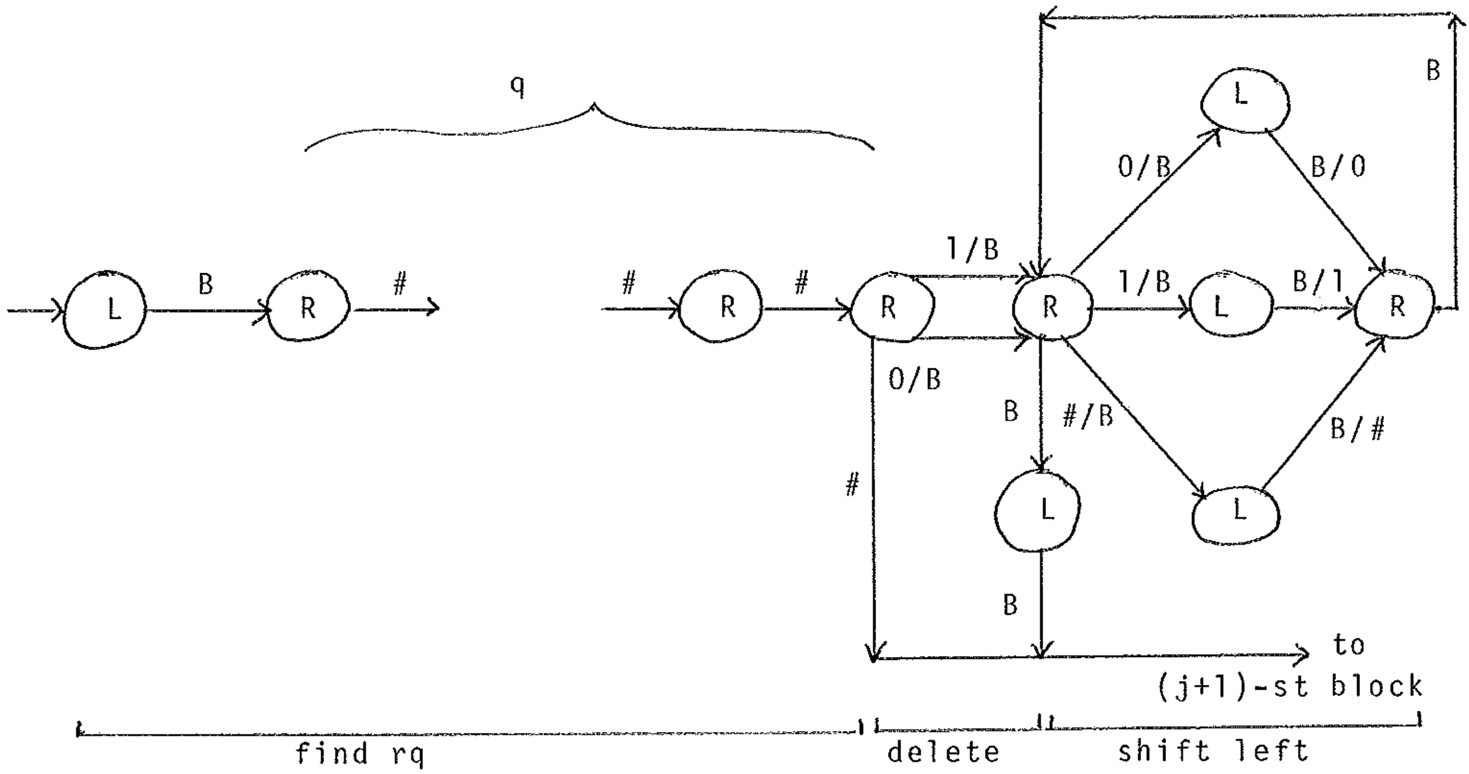


To simplify our diagrams, we assume that the RAM alphabet is $\Sigma = \{0, 1\}$. Then the alphabet of the Turing Machine is $\Gamma = \{0, 1, \#, B\}$. Each RAM statement is translated as a Turing machine block as follows: We have four blocks, one for each instruction.

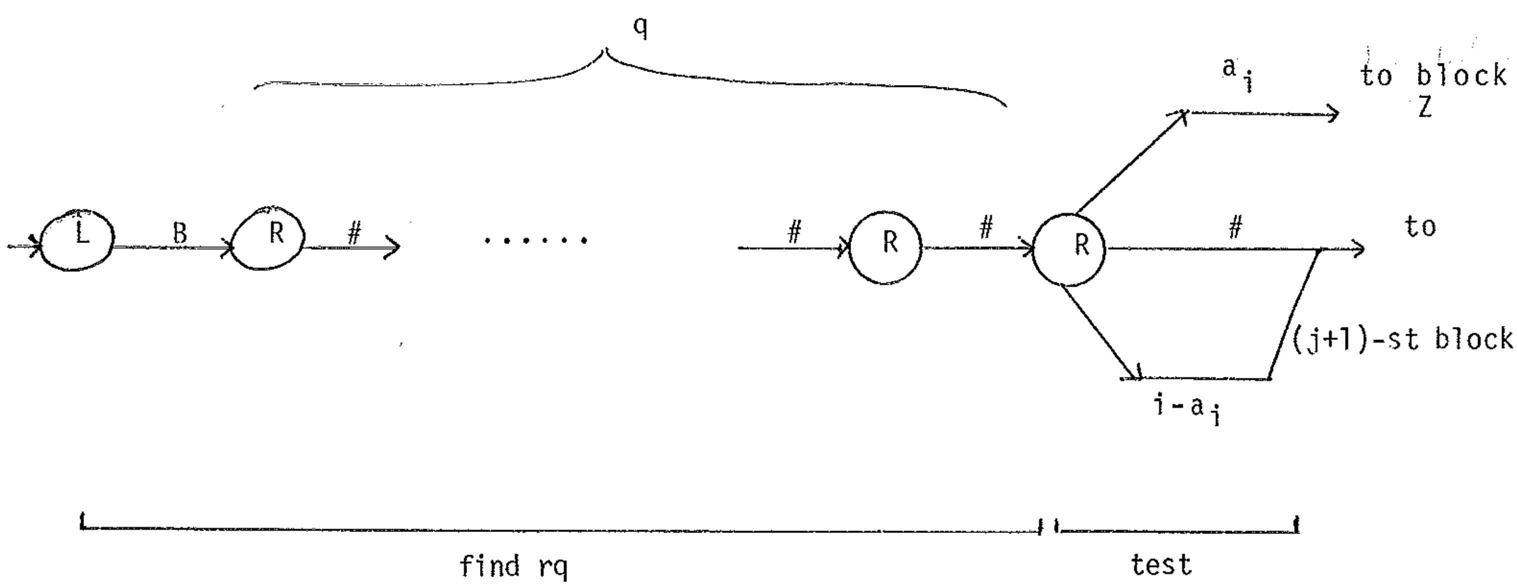
(a) add_i Rq



(b) del Rq

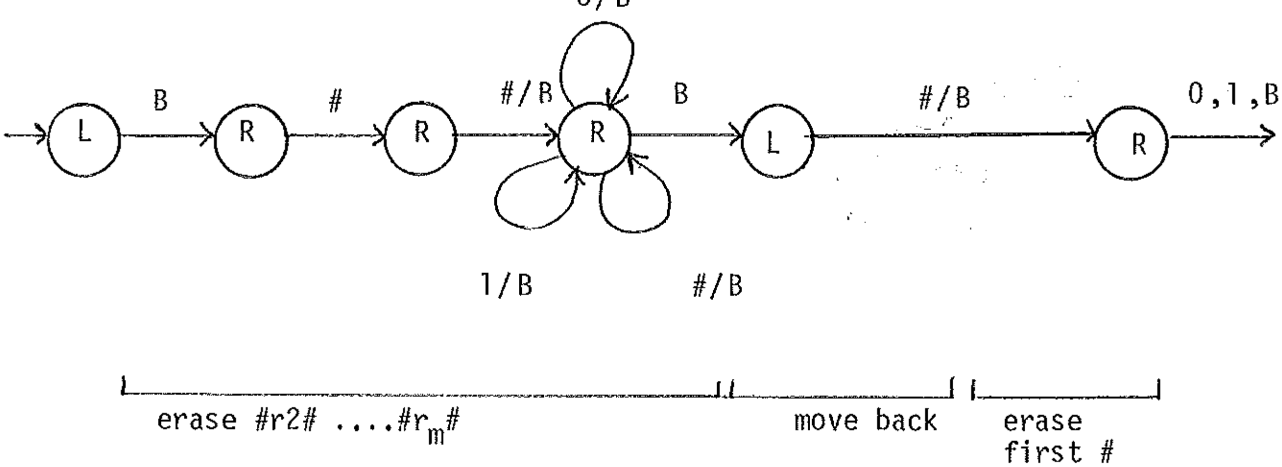


(c) jump$_i$Z



Finally, we clean up the tape by erasing all but the contents of R1 from the tape. This block corresponds to the last continue statement

(d)



Finally, a continue statement which is not the last continue in the RAM program is translated as an arrow from the exit of the j-th block to the entry of the (j+1)-th block.

Notice that the Turing machine produced by the translation has the nice property that it never moves left of the blank square immediately to the left of its leftmost # - In other words, the tape needs only be unbounded to the right. We leave as an exercise to prove that every Turing-computable function is computable by a Turing machine which never moves more than one square to the left of its starting position.

Our next goal is to show that every Turing computable function is RAM-computable. This shows that RAM's and Turing machines compute exactly the same class of functions. We will also show that every RAM computable function is partial recursive, proving that the partial recursive functions are exactly the class of functions computed by RAM's and Turing machine. This provides evidence for the Church - Turing thesis. The Church-Turing thesis asserts that the class of partial recursive functions is exactly the intuitively conceived class of effectively computable functions, also called algorithmically computable functions. The Church-Turing thesis is usually accepted on intuitive grounds by Computer Scientists and Mathematicians. It is strengthened by the fact that any known definition of the notion of effectively computable functions has been shown to be equivalent to the notion of partial recursive function. Furthermore, the translation from one system to another is always effective (but sometimes very tedious).

In order to prove that every Turing - computable function is RAM-computable, recall that we showed that the concatenation function con is RAM computable. Also, for any $n \geq 2$, the extended concatenation function con_n such that $con_n(x_1, \ldots, x_n) = x_1 \ldots x_n$ for all $x_1, \ldots, x_n$ is primitive recursive and consequently RAM-computable. Finally, RAM programs are closed under composition. This allows us to write RAM programs as a composition of "blocks", avoiding the tedious task of writing the program in full.

1.5.7 Theorem

Every Turing-computable function is RAM-computable. Furthermore, given a Turing machine M computing ϕ, we can effectively construct a RAM program P computing ϕ.

Proof: The notion of an instantaneous description (ID) will be crucial in the translation. Let's abbreviate Turing Machine by T.M.

Let M be a given T.M.

$M = (K, \Gamma, \Delta, \delta, q_0)$ where

$K = \{q_0, \ldots, q_m\}$, $\Gamma = \{a_1, \ldots, a_k, B, ","\}$.

Assume that M computes the partial function ϕ of n arguments. We construct a RAM program P simulating M and computing ϕ. The program P, after some initialization, contains the current ID of M in register R1. For each move of M, P updates the current ID to the next ID.

Initially, P takes the n inputs $x_1, \ldots, x_n$ and creates $\#ID_0\#=\#q_0x_1,x_2, \ldots, x_n\#$ in R1 (ID_0 surrounded by the marker #). Then, P simulates M. If and when M halts in a halting configuration B^kqwB^l, P places the output w in R1 and stops. If the output is improper, P loops forever.

The alphabet Σ for P is $\Sigma = \Gamma \cup K \cup \{\#\}$ where it is assumed that $\Gamma \cap K = \emptyset$ and # is neither in Γ nor K.

We will denote $B = a_{k+1}$ and $\# = a_{k+2}$

When P simulates a move of M by updating the ID, R1 contains the current ID which is of the form ua_jpa_iv and is such that: if $u = \varepsilon$ then $a_j = \#$, v is always nonempty, but if v is a single symbol, then $v = \#$. In the first phase in updating the ID, P transfers u into R2 and a_j into R3. Then it reads a_i, and depending on (p,a_i) it simulates the action of M. In order to remember p and a_i, P has labels of the form jp and jpi. Right moves are accomplished at the addresses jpiR and jpiR#. Left moves are accomplished at the addresses jpiL and jpiL#. The updated ID is placed back in R1. When a halting ID is found, P checks that it is proper. If the halting ID is proper, the output is returned in R1, otherwise P loops forever. For simplicity, we adopt a "subroutine notation". We also omit the suffix a or b in the target labels of jumps, which is not a problem since all jumps in P are uniquely defined.

<u>Program P</u>

The initialization of P is:

$\text{con}_{2n+2}(\#, q_0, x_1, ",", \ldots, ",", x_n, \#)$

				initialize
BEGIN	clr	R2		
	clr	R3		
	jmp	TEST		
NU	del	R1		
TEST	R1	jmp_1	A1	
	.			
	.			
	R1	jmp_{k+2}	A(k+2)	
	R1	jmp_{q_0}	Q0	
	.			
	.			
	R1	jmp_{q_m}	Qm	

Subroutine Ai:

Ai	R3	jmp_1	ui1	
	.			
	.			
	R3	jmp_{k+2}	ui(k+2)	update R2 and R3
	add_i	R3		$R3 \leftarrow a_i$
	jmp	NU		
ui1	add_1	R2		
	jmp	upr3		R2 = con(R2,R3)
	.			
ui(k+2)	add_{k+2}	R2		
	jmp	upr3		

```
upr3      clr        R3              update R3
          add_i      R3              R3 ← a_i
          jmp        NU
```

For each p, $0 \le p \le m$ we have:

```
Qp        R3         jmp_1       1p        to remember
          .                                a_j p
          .
          R3         jmp_{k+2}   (k+2)p
```

For each jp, $i \le j \le k+2$, $0 \le p \le m$ we have:

```
jp             del        R1                 to remember
     R1        jmp_1      jp1                a_j p a_i
               .
               .
     R1        jmp_{k+1}  jp(k+1)
```

We have 3 cases:

1) If $(p,a_i,b,q,R) \in \delta$ then jpi is:

```
jpi            del        R1
               R1         jmp_1       jpiR
               .
               R1         jmp_{k+1}   jpiR
               R1         jmp_#       jpiR#
jpiR           con_3(R2, a_j bq, R1)
               jmp        BEGIN
```

This simulates the transition:

$$u a_j p a_i v \to u a_j b q v \quad \text{where } v \ne \#$$

```
jpiR#          con_2(R2, a_j bqB#)
               jmp        BEGIN
```

This simulates the transition:

$ua_j pa_i\# \rightarrow ua_j bqB\#$

2) If $(p,a_i,b,q,L) \in \delta$ then jpi is:

```
jpi     del     R1
        R3      jmp_1         jpiL
         .
         .
        R3      jmp_k+1       jpiL
        R3      jmp_#         jpiL#
jpiL    con_3(R2, qa_jb,R1)
        jmp       BEGIN
```

This simulates the transition:

$ua_j pa_i v \rightarrow uqa_j bv$ where $a_j \neq \#$

```
jpiL#   con_2(#qBb,R1)
        jmp       BEGIN
```

This simulates the transition:

$\#pa_i v \rightarrow \#qBbv$

3) If no quintuple begins with (p,a_i) then $upa_i v$ is a halting configuration. We test if it is proper. For each such jpi we have:

```
jpi         del     R1
            jmp     PROPER
PROPER      con3(R2, ajpai,R1)
            R2 ←    R1
            R2      jmp#              B        ⎤
            jmp     Loop                       ⎥
HEAD        R2      jmpB              B        ⎥
            R2      jmpq0             Q        ⎥   test if
             .                                 ⎥   ID starts
             .                                 ⎥   with
             .                                 ⎥   #B^kq
            R2      jmpqm             Q        ⎥
            jmp                       LOOP     ⎥
B           del     R2                         ⎥
            jmp                       HEAD     ⎦
Q           clr     R1                         ⎤
MORE        del     R2                         ⎥
            R2      jmp1              RES1     ⎥   to put
             .                                 ⎥   result
             .                                 ⎥   in R1
             .                                 ⎥
            R2      jmpk              RESk     ⎥
            R2      jmpB              BTAIL    ⎦
            R2      jmp#              STOP
            jmp                       LOOP
```

For each RESi, $1 \le i \le k$, we have:

RESi	add_i	R1		adds a_i to output
	jmp		MORE	
BTAIL	del	R2		test if ID ends with $B^\ell\#$
	R2	jmp_B	BTAIL	
	R2	$jmp_\#$	STOP	
	jmp		LOOP	
LOOP	jmp		LOOP	
STOP	continue			

This concludes the program P.

The last phase of the program (PROPER) is not necessary if we start with a Turing machine which only halts in proper ID's (it it halts). We leave the following proposition as an exercise to the reader.

1.5.8 <u>Proposition</u>

Given a Turing machine M computing a function ϕ, we can effectively construct a T.M. M' computing ϕ with the following additional properties:

1) M' halts in a proper ID iff M halts in a proper ID

2) M' diverges iff either M diverges or M halts in an improper ID.

The construction is possible because the TM M' can check whether or not a halting ID of M is proper, and if improper, it loops forever.

From now on, we assume that our Turing machines satisfy proposition 1.5.8.

We conclude this chapter by proving that every Turing-computable function is partial recursive. This will close the circle, establishing the equivalence of the partial recursive functions, the RAM-computable functions and the Turing-computable functions.

Again the main technique will be to use instantaneous descriptions. We are going to define a primitive recursive function which simulates the transitions of a T.M. from a starting ID. Instantaneous descriptions will be represented as #upav# where p is a state, $a \in \Gamma$ and $u, v \in \Gamma^*$.

Given a T.M. $M = (K, \Gamma, \Delta, \delta, q_0)$ we associate the following pairs of ID's describing the transitions of M.

For every $(p, a, b, R, q) \in \delta$, we have the pairs:

(paa_1, bqa_1)

.

.

(paa_k, bqa_k)

$(pa\#, bqB\#)$

For every $(p, a, b, L, q) \in \delta$, we have the pairs:

(a_1pa, qa_1b)

.

.

(a_kpa, qa_kb)

$(\#pa, \#qBb)$

The set of pairs is called TRANS and is assumed to be ordered in some way. Each pair will be denoted $\ell_i \to r_i$. We have N such pairs.

We also need the list BLOCKED of pairs of symbols pa, such that no quintuple in δ starts with pa. They are ordered as follows: $p_{i_1} a_{i_1}, \ldots, p_{i_m} a_{i_m}$.

Next, we need the following primitive recursive functions.

1.5.9 Proposition

The following functions are primitive recursive:

1. $Oc(x,y)$, where $Oc(x,y)$ holds iff x is a substring of y.
2. $u(x,z)$ = the prefix of z to the left of the leftmost occurrence of x in z if $Oc(x,z)$.
3. $v(x,z)$ = the suffix of z to the right of the leftmost occurrence of x in z if $Oc(x,z)$
4. $rep(x,y,z)$ = the result of replacing the leftmost occurrence of x by y in z if $Oc(x,z)$.

Proof:
1. $Oc(x,y)$ iff $\exists z/y \ \exists w/y[z=wx]$
2. $u(x,z) = \min y/z \ \exists w/z[yx=w]$
3. $v(x,z) = z - u(x,z)x$
4. $rep(x,y,z) = u(x,z)y\ v(x,z)$

Note that for every "legal" ID, there is at most one occurrence of either ℓ_i or r_i for some $\ell_i \to r_i$ in TRANS. This is why it doesn't hurt to pick the leftmost occurrence.

1.5.10 Proposition

For any Turing machine M, the following are primitive recursive:

1. The function T such that $T(ID_0, y) = ID$ iff $ID_0 \overset{*}{\underset{|y|}{\rightarrow}} ID$ in $|y|$ steps.
2. HALT(ID) iff ID is a halting ID.
3. STOP(ID,y) iff M halts in a halting ID after $|y|$ steps.

Proof: Note that we actually do not care what T, HALT and STOP do if ID_0 and ID are not proper representations of ID's. T is defined as follows.

1. $T(x,\varepsilon) = x$

$$T(x,ya_i) = \begin{cases} rep(\ell_1, r_1, T(x,y)) & \text{iff } Oc(\ell_1,T(x,y)) \\ rep(\ell_2,r_2,T(x,y)) & \text{iff } Oc(\ell_2,T(x,y)) \\ & \quad \underline{\text{and}}\ \underline{\text{not}}\ Oc(\ell_1,T(x,y)) \\ \vdots & \\ rep(\ell_N,r_N, T(x,y)) & \text{iff } Oc(\ell_N,T(x,y)) \\ & \quad \underline{\text{and}}\ \underline{\text{not}}\ Oc(\ell_1, T(x,y)) \ldots \\ & \quad \underline{\text{and}}\ \underline{\text{not}}\ Oc(\ell_{N-1}, T(x,y)) \\ T(x,y) & \text{otherwise} \end{cases}$$

2. HALT(x) iff

$$[OC(p_{i_1} a_{i_1}, x)\ \underline{\text{or}}\ \ldots\ldots\ \underline{\text{or}}\ OC(P_{i_m} a_{i_m},x)]$$

3. STOP(x,y) iff HALT(T(x,y))

The starting ID is defined as:

$ID_0 = \# q_0 x_1, x_2, \ldots, x_n \#$

(for a T.M. computing a function of n arguments).

Let INIT be the function such that $INIT(x_1,\ldots,x_n) = \#q_0 x_1,\ldots,x_n\#$. Clearly, INIT is primitive recursive. Then, for all $x_1, \ldots, x_n$, we have:

$ID_0 \overset{*}{\underset{|y|}{\rightarrow}}$ ID and ID is a halting ID iff:

$T(INIT(x_1, \ldots, x_n), \min_1 y[STOP(INIT(x_1, \ldots, x_n), y)]) = ID$

Let RES be the function which cleans up a halting ID to produce the output.

RES is defined by primitive recursion as follows: (recall that rev is the reverse function)

$RES(\varepsilon) = \varepsilon$

$RES(x\#) = RES(x)$

$RES(xB) = RES(x)$

$1 \leq i \leq k \quad RES(xa_i) = con(RES(x), a_i)$

$RES(xq) = RES(rev(x))$ for all $q \varepsilon K$

For any halting ID $\#B^k quB^\ell\#$ with $u \varepsilon \Sigma^*$, it is easily seen that $RES(\#B^k quB^\ell\#) = u$.

Therefore, we have shown the Theorem:

1.5.11 <u>Theorem</u>

Every T.M. - computable function ϕ of n arguments is partial recursive. Moreover, given a T.M. M, we can effectively find a definition of ϕ of the following form:

$\phi(x_1, \ldots, x_n) =$

$RES(T(INIT(x_1, \ldots, x_n), \min_1 y[STOP(INIT(x_1, \ldots, x_n), y)]))$

1.5.12 <u>Corollary</u>

Every partial recursive function ϕ can be effectively obtained in the form $\phi = f o \min_1 g$
where f and g are primitive recursive functions.

Consequently, every partial recursive function has a definition in which minimization is applied at most once.

In the next section, we turn to the problem of encoding RAM programs, aiming to prove that "universal programs" exist. In the course of the proof, we will reprove that every RAM-computable function is partial recursive.

A key technical result needed in the proof is the fact that pairs of integers can be encoded into integers using "pairing functions". This is the object of the next section.