Chapter 4

RAM Programs, Turing Machines, and the Partial Recursive Functions

4.1. Partial Functions and RAM Programs

We define an abstract machine model for computing functions

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet. Numerical functions $f: \mathbb{N}^n \to \mathbb{N}$ can be viewed as functions defined over the one-letter alphabet $\{a_1\}$, using the bijection $m \mapsto a_1^m$.

Let us recall the definition of a partial function.

A binary relation $R \subseteq A \times B$ between two sets A and B is *functional* iff, for all $x \in A$, and $y, z \in B$,

$$(x, y) \in R \quad \text{and} \quad (x, z) \in R \quad \text{implies that} \quad y = z.$$

A *partial function* is a triple $f = \langle A, G, B \rangle$, where A and B are arbitrary sets (possibly empty) and G is a functional relation (possibly empty) between A and B, called the *graph* of f.

Hence, a partial function is a functional relation such that every argument has at most one image under f.

The graph of a function f is denoted as $graph(f)$. When no confusion can arise, a function f and its graph are usually identified.

A partial function $f = \langle A, G, B \rangle$ is often denoted as $f \colon A \to B$.

The *domain* $dom(f)$ of a partial function $f = \langle A, G, B \rangle$ is the set

$$dom(f) = \{x \in A \mid \exists y \in B,\ (x, y) \in G\}.$$

For every element $x \in dom(f)$, the unique element $y \in B$ such that $(x, y) \in graph(f)$ is denoted as $f(x)$. We say that $f(x)$ *converges*, also denoted as $f(x) \downarrow$.

If $x \in A$ and $x \notin dom(f)$, we say that $f(x)$ *diverges*, also denoted as $f(x) \uparrow$.

Intuitively, if a function is partial, it does not return any output for any input not in its domain. This corresponds to an infinite computation.

A partial function $f\colon A \to B$ is a *total function* iff $dom(f) = A$. It is customary to call a total function simply a function.

We now define a model of computation know as the *RAM programs*, or *Post machines*. RAM programs are written in a sort of assembly language involving simple instructions manipulating strings stored into registers.

Every RAM program uses a fixed and finite number of registers denoted as $R1, \ldots, Rp$, with no limitation on the size of strings held in the registers.

RAM programs can be defined either in flowchart form or in linear form. Since the linear form is more convenient for coding purposes, we present RAM programs in linear form.

A RAM program P (in linear form) consists of a finite sequence of instructions using a finite number of registers $R1, \ldots, Rp$.

Instructions may optionally be labeled with line numbers denoted by $N1, \ldots, Nq$.

It is neither mandatory to label all instructions, nor to use distinct line numbers!

Thus, the same line number can be used in more than one line. As we will see later on, this makes it easier to concatenate two different programs without performing a renumbering of line numbers.

Every instruction has four fields, not necessarily all used. The main field is the **op-code**.

Definition 4.1.1 RAM programs are constructed from seven types of instructions shown below:

(1_j)	N		$\texttt{add}_j$	Y
(2)	N		$\texttt{tail}$	Y
(3)	N		$\texttt{clr}$	Y
(4)	N	Y	$\leftarrow$	X
$(5a)$	N		$\texttt{jmp}$	$N1a$
$(5b)$	N		$\texttt{jmp}$	$N1b$
$(6_j a)$	N	Y	$\texttt{jmp}_j$	$N1a$
$(6_j b)$	N	Y	$\texttt{jmp}_j$	$N1b$
(7)	N		$\texttt{continue}$	

An instruction of type (1_j) concatenates the letter a_j to the right of the string held by register Y $(1 \leq j \leq k)$. The effect is the assignment

$$Y := Ya_j$$

An instruction of type (2) deletes the leftmost letter of the string held by the register Y. This corresponds to the function $tail$, defined such that

$$\begin{aligned} tail(\epsilon) &= \epsilon, \\ tail(a_j u) &= u. \end{aligned}$$

The effect is the assignment

$$Y := tail(Y)$$

An instruction of type (3) clears register Y, i.e., sets its value to the empty string ϵ. The effect is the assignment

$$Y := \epsilon$$

An instruction of type (4) assigns the value of register X to register Y. The effect is the assignment

$$Y := X$$

An instruction of type (5a) or (5b) is an unconditional jump.

The effect of (5a) is to jump to the closest line number $N1$ occurring above the instruction being executed, and the effect of (5b) is to jump to the closest line number $N1$ occurring below the instruction being executed.

An instruction of type $(6_j a)$ or $(6_j b)$ is a conditional jump. Let *head* be the function defined as follows:

$$head(\epsilon) = \epsilon,$$
$$head(a_j u) = a_j.$$

The effect of $(6_j a)$ is to jump to the closest line number $N1$ occurring above the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction (the one immediately following the instruction being executed).

The effect of $(6_j b)$ is to jump to the closest line number $N1$ occurring below the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction.

When computing over $\mathbb{N}$, instructions of type $(6_j b)$ jump to the closest $N1$ above or below iff Y is nonnull.

An instruction of type (7) is a no-op, i.e., the registers are unaffected. If there is a next instruction, then it is executed, else, the program stops.

Obviously, a program is syntactically correct only if certain conditions hold.

Definition 4.1.2 A *RAM program* P is a finite sequence of instructions as in Definition 4.1.1, and satisfying the following conditions:

(1) For every jump instruction (conditional or not), the line number to be jumped to must exist in P.

(2) The last instruction of a RAM program is a `continue`.

The reason for allowing multiple occurences of line numbers is to make it easier to concatenate programs without having to perform a renaming of line numbers. The technical choice of jumping to the closest address $N1$ above or below comes from the fact that it is easy to search up or down using primitive recursion, as we will see later on.

It is fairly obvious that linear RAM programs can be represented in flowchart form, and that the two models are equivalent. We will not worry about this in this Chapter.

For the purpose of computing a function $f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$ using a RAM program P, we assume that P has at least n registers called *input registers*, and that these registers $R1, \ldots, Rn$ are initialized with the input values of the function f. We also assume that the output is returned in register $R1$.

The following RAM program concatenates two strings x_1 and x_2 held in registers $R1$ and $R2$:

	$R3$	$\leftarrow$	$R1$
	$R4$	$\leftarrow$	$R2$
$N0$	$R4$	$\texttt{jmp}_a$	$N1b$
	$R4$	$\texttt{jmp}_b$	$N2b$
		`jmp`	$N3b$
$N1$		$\texttt{add}_a$	$R3$
		`tail`	$R4$
		`jmp`	$N0a$
$N2$		$\texttt{add}_b$	$R3$
		`tail`	$R4$
		`jmp`	$N0a$
$N3$	$R1$	$\leftarrow$	$R3$
		`continue`	

Since $\Sigma = \{a, b\}$, for more clarity, we wrote $\texttt{jmp}_a$ instead of $\texttt{jmp}_1$, $\texttt{jmp}_b$ instead of $\texttt{jmp}_2$, $\texttt{add}_a$ instead of $\texttt{add}_1$, and $\texttt{add}_b$ instead of $\texttt{add}_2$.

Definition 4.1.3 A RAM program P computes the partial function $\varphi\colon (\Sigma^*)^n \to \Sigma^*$ if the following conditions hold: For every input $(x_1, \ldots, x_n) \in (\Sigma^*)^n$, having initialized the input registers $R1, \ldots, Rn$ with $x_1, \ldots, x_n$, the program eventually halts iff $\varphi(x_1, \ldots, x_n)$ converges, and if and when P halts, the value of $R1$ is equal to $\varphi(x_1, \ldots, x_n)$. A partial function φ is *RAM-computable* iff it is computed by some RAM program.

For example, the following program computes the *erase function* E defined such that

$$E(u) = \epsilon$$

for all $u \in \Sigma^*$:

```
clr          R1
continue
```

The following program computes the *jth successor function* S_j defined such that

$$S_j(u) = ua_j$$

for all $u \in \Sigma^*$:

$$\begin{array}{ll} \texttt{add}_j & R1 \\ \texttt{continue} & \end{array}$$

The following program (with n input variables) computes the *projection function* P_i^n defined such that

$$P_i^n(u_1, \ldots, u_n) = u_i,$$

where $n \geq 1$, and $1 \leq i \leq n$:

$$\begin{array}{lll} R1 & \leftarrow & Ri \\ & \texttt{continue} & \end{array}$$

Note that P_1^1 is the identity function.

Having 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 programs don't do much, but this is not so. Indeed, we will see shortly that the class of RAM-computable functions is quite extensive.

One way of getting new programs from previous ones is via composition. Another one is by primitive recursion. We will investigate these constructions after introducing another model of computation, *Turing machines*.

Remarkably, the classes of (partial) functions computed by RAM programs and by Turing machines are identical. This is the class of *partial recursive function*. This class can be given several other definitions. We will present the definition of the so-called *μ-recursive functions* (due to Kleene).

The following Lemma will be needed to simplify the encoding of RAM programs as numbers:

Lemma 4.1.4 *Every RAM program can be converted to an equivalent program only using the following type of instructions:*

(1_j)	N		$\texttt{add}_j$	Y
(2)	N		$\texttt{tail}$	Y
$(6_j a)$	N	Y	$\texttt{jmp}_j$	$N1a$
$(6_j b)$	N	Y	$\texttt{jmp}_j$	$N1b$
(7)	N		$\texttt{continue}$	

The proof is fairly simple. For example, instructions of the form

$$Ri \leftarrow Rj$$

can be eliminated by tranferring the contents of Rj in reverse order into an auxiliary register Rk, and then by transferring the contents of Rk in reverse order into Ri.

4.2. Definition of a Turing Machine

We define a Turing machine model for computing functions

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_N\}$ is some input alphabet. We only consider deterministic Turing machines.

A Turing machine also uses a *tape alphabet* Γ such that $\Sigma \subset \Gamma$. The tape alphabet contains some special symbol $B \notin \Sigma$, the *blank*.

In this model, a Turing machine uses a single tape. This tape can be viewed as a string over Γ. The tape is both an input tape and a storage mechanism.

Symbols on the tape can be overwritten, and the tape can grow either on the left or on the right. There is a read/write head pointing to some symbol on the tape.

Unlike Pushdown automata or NFA's, the read/write head can move left or right.

Definition 4.2.1 A (deterministic) *Turing machine* (or *TM*) M is a sextuple $M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$, where

- K is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *tape alphabet*, s.t. $\Sigma \subset \Gamma$, $K \cap \Gamma = \emptyset$, and with blank $B \notin \Sigma$;
- $q_0 \in K$ is the *start state* (or *initial state*);
- δ is the *transition function*, a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

such that for all $(p, a) \in K \times \Gamma$, there is at most one triple $(b, m, q) \in \Gamma \times \{L, R\} \times K$ such that $(p, a, b, m, q) \in \delta$.

A quintuple $(p, a, b, m, q) \in \delta$ is called an *instruction*. It is also denoted as

$$p, a \rightarrow b, m, q.$$

The effect of an instruction is to switch from state p to state q, overwrite the symbol currently scanned a with b, and move the read/write head either left or right, according to m.

4.3. Computations of Turing Machines

To explain how a Turing machine works, we describe its action on *Instantaneous descriptions*. We take advantage of the fact that $K \cap \Gamma = \emptyset$ to define instantaneous descriptions.

Definition 4.3.1 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

an *instantaneous description* (for short an *ID*) is a (nonempty) string in $\Gamma^* K \Gamma^+$, that is, a string of the form

$$upav,$$

where $u, v \in \Gamma^*$, $p \in K$, and $a \in \Gamma$.

The intuition is that an ID $upav$ describes a snapshot of a TM in the current state p, whose tape contains the string uav, and with the read/write head pointing to the symbol a.

Thus, in $upav$, the state p is just to the left of the symbol presently scanned by the read/write head.

We explain how a TM works by showing how it acts on ID's.

Definition 4.3.2 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

the *yield relation (or compute relation)* $\vdash$ is a binary relation defined on the set of ID's as follows:

For any two ID's ID_1 and ID_2, we have $ID_1 \vdash ID_2$ iff either

(1) $(p, a, b, R, q) \in \delta$, and either

(a) $ID_1 = upacv$, $c \in \Gamma$, and $ID_2 = ubqcv$, or

(b) $ID_1 = upa$ and $ID_2 = ubqB$;

or

(2) $(p, a, b, L, q) \in \delta$, and either

(a) $ID_1 = ucpav$, $c \in \Gamma$, and $ID_2 = uqcbv$, or

(b) $ID_1 = pav$ and $ID_2 = qBbv$.

Note how the tape is extended by one blank after the rightmost symbol in case (1)(b), and by one blank before the leftmost symbol in case (2)(b).

As usual, we let $\vdash^+$ denote the transitive closure of $\vdash$, and we let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$.

We can now explain how a Turing function computes a partial function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*.$$

Since we allow functions taking $n \geq 1$ input strings, we assume that Γ contains the special delimiter **,** not in Σ, used to separate the various input strings.

It is convenient to assume that a Turing machine "cleans up" its tape when it halts, before returning its output. For this, we will define proper ID's.

Definition 4.3.3 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

where Γ contains some delimiter **,** not in Σ in addition to the blank B, a *starting ID* is of the form

$$q_0 w_1, w_2, \ldots, w_n$$

where $w_1, \ldots, w_n \in \Sigma^*$ and $n \geq 2$, or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$.

A *blocking (or halting) ID* is an ID $upav$ such that there are no instructions $(p, a, b, m, q) \in \delta$ for any $(b, m, q) \in \Gamma \times \{L, R\} \times K$.

A *proper ID* is a halting ID of the form

$$B^k p w B^l,$$

where $w \in \Sigma^*$, and $k, l \geq 0$ (with $l \geq 1$ when $w = \epsilon$).

Computation sequences are defined as follows:

Definition 4.3.4 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

a *computation sequence (or computation)* is a finite or infinite sequence of ID's

$$ID_0, ID_1, \ldots, ID_i, ID_{i+1}, \ldots,$$

such that $ID_i \vdash ID_{i+1}$ for all $i \geq 0$.

A computation sequence *halts* iff it is a finite sequence of ID's, so that $ID_0 \vdash^* ID_n$ and ID_n is a halting ID.

A computation sequence *diverges* if it is an infinite sequence of ID's.

We now explain how a Turing machine computes a partial function.

Definition 4.3.5 A Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$$

computes the partial function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$$

iff the following conditions hold:

(1) For every $w_1, \ldots, w_n \in \Sigma^*$, given the starting ID

$$ID_0 = q_0 w_1, w_2, \ldots, w_n$$

or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$, the computation sequence of M from ID_0 halts in a proper ID iff $f(w_1, \ldots, w_n)$ is defined.

(2) If $f(w_1, \ldots, w_n)$ is defined, then M halts in a proper ID of the form

$$ID_n = B^k p f(w_1, \ldots, w_n) B^h,$$

which means that it computes the right value.

A function f (over Σ^*) is *Turing computable* iff it is computed by some Turing machine M.

Note that by (1), the TM M may halt in an improper ID, in which case $f(w_1, \ldots, w_n)$ must be undefined. This corresponds to the fact that we only accept to retrieve the output of a computation if the TM has cleaned up its tape, i.e., produced a proper ID. In particular, intermediate calculations have to be erased before halting.

Example.

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

$\Sigma = \{a, b\}$;

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

The instructions in δ are:

$$
\begin{aligned}
q_0, B &\to B, R, q_3, \\
q_0, a &\to b, R, q_1, \\
q_0, b &\to a, R, q_1, \\
q_1, a &\to b, R, q_1, \\
q_1, b &\to a, R, q_1, \\
q_1, B &\to B, L, q_2, \\
q_2, a &\to a, L, q_2, \\
q_2, b &\to b, L, q_2, \\
q_2, B &\to B, R, q_3.
\end{aligned}
$$

The reader can easily verify that this machine exchanges the a's and b's in a string. For example, on input $w = aaababb$, the output is $bbbabaa$.

4.4. RAM-computable functions are Turing-computable

Turing machines can simulate RAM programs, and as a result, we have the following Theorem:

Theorem 4.4.1 *Every RAM-computable function is Turing-computable. Furthermore, given a RAM program P, we can effectively construct a Turing machine M computing the same function.*

The idea of the proof is to represent the contents of the registers $R1, \ldots Rp$ on the Turing machine tape by the string

$$\#r1\#r2\#\cdots\#rp\#,$$

Where # is a special marker and ri represents the string held by Ri, We also use Lemma 4.1.4 to reduce the number of instructions to be dealt with.

The Turing machine M is built of blocks, each block simulating the effect of some instruction of the program P. The details are a bit tedious, and can be found in the notes or in Machtey and Young.

4.5. Turing-computable functions are RAM-computable

RAM programs can also simulate Turing machines.

Theorem 4.5.1 *Every Turing-computable function is RAM-computable. Furthermore, given a Turing machine M, one can effectively construct a RAM program P computing the same function.*

The idea of the proof is to design a RAM program containing an encoding of the current ID of the Turing machine M in register $R1$, and to use other registers $R2, R3$ to simulate the effect of executing an instruction of M by updating the ID of M in $R1$.

The details are tedious and can be found in the notes.

Another proof can be obtained by proving that the class of Turing computable functions coincides with the class of *partial recursive functions*. Indeed, it turns out that both RAM programs and Turing machines compute precisely the class of partial recursive functions.

First, we define the *primitive recursive functions*.

4.6. The Primitive Recursive Functions

The class of primitive recursive functions is defined in terms of base functions and closure operations.

Definition 4.6.1 Let $\Sigma = \{a_1, \ldots, a_N\}$. The *base functions* over Σ are the following functions:

(1) The *erase function* E, defined such that $E(w) = \epsilon$, for all $w \in \Sigma^*$;

(2) For every j, $1 \leq j \leq N$, the *j-successor function* S_j, defined such that $S_j(w) = wa_j$, for all $w \in \Sigma^*$;

(3) The *projection functions* P_i^n, defined such that

$$P_i^n(w_1, \ldots, w_n) = w_i,$$

for every $n \geq 1$, every i, $1 \leq i \leq n$, and for all $w_1, \ldots, w_n \in \Sigma^*$.

Note that P_1^1 is the identity function on Σ^*. Projection functions can be used to permute the arguments of another function.

A crucial closure operation is (extended) composition.

Definition 4.6.2 Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

and any m functions

$$h_i: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

the *composition of* g *and the* h_i is the function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

denoted as $g \circ (h_1, \ldots, h_m)$, such that

$$f(w_1, \ldots, w_n) = g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n)),$$

for all $w_1, \ldots, w_n \in \Sigma^*$.

As an example, $f = g \circ (P_2^2, P_1^2)$ is such that

$$f(w_1, w_2) = g(w_2, w_1).$$

Another crucial closure operation is primitive recursion.

Definition 4.6.3 Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m-1} \to \Sigma^*,$$

where $m \geq 2$, and any N functions

$$h_i: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

the function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *primitive recursion from* g *and* $h_1, \ldots, h_N$, if

$$\begin{aligned}
f(\epsilon, w_2, \ldots, w_m) &= g(w_2, \ldots, w_m), \\
f(ua_1, w_2, \ldots, w_m) &= h_1(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m), \\
&\ldots = \ldots \\
f(ua_N, w_2, \ldots, w_m) &= h_N(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m),
\end{aligned}$$

for all $u, w_2, \ldots, w_m \in \Sigma^*$.

When $m = 1$, for some fixed $w \in \Sigma^*$, we have

$$\begin{aligned} f(\epsilon) &= w, \\ f(ua_1) &= h_1(u, f(u)), \\ \ldots &= \ldots \\ f(ua_N) &= h_N(u, f(u)), \end{aligned}$$

for all $u \in \Sigma^*$.

For numerical functions (i.e., when $\Sigma = \{a_1\}$), the scheme of primitive recursion is simpler:

$$\begin{aligned} f(0, x_2, \ldots, x_m) &= g(x_2, \ldots, x_m), \\ f(x+1, x_2, \ldots, x_m) &= h_1(x, f(x, x_2, \ldots, x_m), x_2, \ldots, x_m), \end{aligned}$$

for all $x, x_2, \ldots, x_m \in \mathbb{N}$.

The *successor function* S is the function

$$S(x) = x + 1.$$

Addition, multiplication, exponentiation, and super-exponentiation can, be defined by primitive recursion as follows (being a bit loose, we should use some projections ...):

$$\begin{aligned}
add(0, n) &= n, \\
add(m + 1, n) &= S(add(m, n)), \\
mult(0, n) &= 0, \\
mult(m + 1, n) &= add(mult(m, n), n), \\
rexp(0, m) &= 1, \\
rexp(m + 1, n) &= mult(rexp(m, n), n), \\
exp(m, n) &= rexp \circ (P_2^2, P_1^2), \\
supexp(0, n) &= 1, \\
supexp(m + 1, n) &= exp(n, supexp(m, n)).
\end{aligned}$$

As an example over $\{a, b\}^*$, the following function $g\colon \Sigma^* \times \Sigma^* \to \Sigma^*$, is defined by primitive recursion:

$$g(\epsilon, v) = P_1^1(v),$$
$$g(ua_i, v) = S_i \circ P_2^3(u, g(u, v), v),$$

where $1 \leq i \leq N$. It is easily verified that $g(u, v) = vu$. Then,

$$f = g \circ (P_2^2, P_1^2)$$

computes the concatenation function, i.e. $f(u, v) = uv$.

Definition 4.6.4 Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *primitive recursive functions* is the smallest class of functions (over Σ^*) which contains the base functions and is closed under composition and primitive recursion.

We leave as an exercise to show that every primitive recursive function is a total function. The class of primitive recursive functions may not seem very big, but it contains all the total functions that we would ever want to compute.

Although it is rather tedious to prove, the following theorem can be shown:

Theorem 4.6.5 *For an alphabet $\Sigma = \{a_1, \ldots, a_N\}$, every primitive recursive function is Turing computable.*

The best way to prove the above theorem is to use the computation model of RAM programs. Indeed, it was shown in Theorem 4.4.1 that every Turing machine can simulate a RAM program.

It is also rather easy to show that the primitive recursive functions are RAM-computable.

In order to define new functions it is also useful to use predicates.

Definition 4.6.6 An *n-ary predicate* P (over Σ^*) is any subset of $(\Sigma^*)^n$. We write that a tuple $(x_1, \ldots, x_n)$ *satisfies* P 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 $C_P \colon (\Sigma^*)^n \to \{a_1\}^*$ defined by

$$C_p(x_1, \ldots, x_n) = \begin{cases} a_1 & \text{iff } P(x_1, \ldots, x_n) \\ \epsilon & \text{iff } \textbf{not } P(x_1, \ldots, x_n). \end{cases}$$

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

We leave to the reader the obvious adaptation of the the notion of primitive recursive predicate to functions defined over $\mathbb{N}$. In this case, 0 plays the role of ϵ and 1 plays the role of a_1.

It is easily shown that if P and Q are primitive recursive predicates (over $(\Sigma^*)^n$), then $P \vee Q$, $P \wedge Q$ and $\neg P$ are also primitive recursive.

As an exercise, the reader may want to prove that the predicate (defined over $\mathbb{N}$):
prime(n) iff n is a prime number, is a primitive recursive predicate.

For any fixed $k \geq 1$, the function:
ord(k, n) = exponent of the kth prime in the prime factorization of n, is a primitive recursive function.

We can also define functions by cases.

Lemma 4.6.7 *If $P_1, \ldots, P_n$ are pairwise disjoint primitive recursive predicates (which means that $P_i \cap P_j = \emptyset$ for all $i \neq j$) and $f_1, \ldots, f_{n+1}$ are primitive recursive functions, the function g defined below is also primitive recursive:*

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

(writing $\overline{x}$ for $(x_1, \ldots, x_n)$.)

It is also useful to have bounded quantification and bounded minimization.

Definition 4.6.8 If P is an $(n+1)$-ary predicate, then the *bounded existential predicate* $\exists y/x P(y, \overline{z})$ holds iff some prefix y of x makes $P(y, \overline{z})$ true.

The *bounded universal predicate* $\forall y/x P(y, \overline{z})$ holds iff every prefix y of x makes $P(y, \overline{z})$ true.

Lemma 4.6.9 *If P is an $(n+1)$-ary primitive recursive predicate, then $\exists y/x P(y, \overline{z})$ and $\forall y/x P(y, \overline{z})$ are also primitive recursive predicates.*

As an application, we can show that the equality predicate, $u = v?$, is primitive recursive.

Definition 4.6.10 If P is an $(n+1)$-ary predicate, then the *bounded minimization of* P, $\min y/x\, P(y, \overline{z})$, is the function defined such that $\min y/x\, P(y, \overline{z})$ is the shortest prefix of x such that $P(y, \overline{z})$ if such a y exists, xa_1 otherwise.

The *bounded maximization of* P, $\max y/x\, P(y, \overline{z})$, is the function defined such that $\max y/x\, P(y, \overline{z})$ is the longest prefix of x such that $P(y, \overline{z})$ if such a y exists, xa_1 otherwise.

Lemma 4.6.11 *If P is an $(n+1)$-ary primitive recursive predicate, then $\min y/x\, P(y, \overline{z})$ and $\max y/x\, P(y, \overline{z})$ are primitive recursive functions.*

So far, the primitive recursive functions do not yield all the Turing-computable functions. In order to get a larger class of functions, we need the closure operation known as minimization.

4.7. The Partial Recursive Functions

The operation of minimization (sometimes called minimalization) is defined as follows:

Definition 4.7.1 Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq N$, the function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *minimization over* $\{a_j\}^*$ *from* g, if the following conditions hold for all $w_1, \ldots, w_m \in \Sigma^*$:

(1) $f(w_1, \ldots, w_m)$ is defined iff there is some $n \geq 0$ such that $g(a_j^p, w_1, \ldots, w_m)$ is defined for all p, $0 \leq p \leq n$, and

$$g(a_j^n, w_1, \ldots, w_m) = \epsilon.$$

(2) When $f(w_1, \ldots, w_m)$ is defined,

$$f(w_1, \ldots, w_m) = a_j^n,$$

where n is such that

$$g(a_j^n, w_1, \ldots, w_m) = \epsilon$$

and

$$g(a_j^p, w_1, \ldots, w_m) \neq \epsilon$$

for every p, $0 \leq p \leq n - 1$.

We also write

$$f(w_1, \ldots, w_m) = min_j u[g(u, w_1, \ldots, w_m) = \epsilon].$$

Note: When $f(w_1, \ldots, w_m)$ is defined,

$$f(w_1, \ldots, w_m) = a_j^n,$$

where n is the smallest integer such that condition (1) holds. It is very important to require that all the values $g(a_j^p, w_1, \ldots, w_m)$ be defined for all p, $0 \leq p \leq n$, when defining $f(w_1, \ldots, w_m)$. Failure to do so allows non-computable functions.

Minimization can be viewed as an abstract version of a while loop:

$u := \epsilon$;
while $g(u, w_1, \ldots, w_m) \neq \epsilon$ **do**
$u := ua_j$;
endwhile
let $f(w_1, \ldots, w_m) = u$

Remark: Kleene used the *μ-notation*:

$$f(w_1, \ldots, w_m) = \mu_j u[g(u, w_1, \ldots, w_m) = \epsilon],$$

actually, its numerical form:

$$f(x_1, \ldots, x_m) = \mu x[g(x, x_1, \ldots, x_m) = 0],$$

The class of partial computable functions is defined as follows.

Definition 4.7.2 Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *partial recursive functions* is the smallest class of functions (over Σ^*) which contains the base functions and is closed under composition, primitive recursion, and minimization. The class of *recursive functions* is the subset of the class of partial recursive functions consisting of functions defined for every input.

One of the major results of computability theory is the following theorem:

Theorem 4.7.3 *For an alphabet* $\Sigma = \{a_1, \ldots, a_N\}$*, every partial recursive function is Turing-computable. Conversely, every Turing-computable function is a partial recursive function. Similarly, the class of recursive functions is equal to the class of Turing-computable functions that halt in a proper ID for every input.*

To prove that every partial recursive function is indeed Turing-computable, since by Theorem 4.4.1, every Turing machine can simulate a RAM program, the simplest thing to do is to show that every partial recursive function is RAM-computable.

For the converse, one can show that given a Turing machine, there is a primitive recursive function describing how to go from one ID to the next. Then, minimization is used to guess whether a computation halts. The proof shows that every partial recursive function needs minimization at most once. The characterization of the recursive functions in terms of TM's follows easily.

There are recursive functions that are not primitive recursive. Such an example is given by Ackermann's function.

Ackermann's function: A recursive function which is **not** primitive recursive:

$$\begin{aligned} A(0,\, y) &= y + 1, \\ A(x+1,\, 0) &= A(x,\, 1), \\ A(x+1,\, y+1) &= A(x,\, A(x+1, y)). \end{aligned}$$

It can be shown that:

$$\begin{aligned} A(0,\, x) &= x + 1, \\ A(1,\, x) &= x + 2, \\ A(2,\, x) &= 2x + 3, \\ A(3,\, x) &= 2^{x+3} - 3, \end{aligned}$$

and

$$A(4, x) = 2^{2^{\cdot^{\cdot^{\cdot^{2^{16}}}}}\}x} - 3,$$

with $A(4, 0) = 16 - 3 = 13$.

For example

$$A(4,1) = 2^{16} - 3, \quad A(4,2) = 2^{2^{16}} - 3.$$

Actually, it is not so obvious that A is a total function. This can be shown by induction, using the lexicographic ordering $\preceq$ on $\mathbb{N} \times \mathbb{N}$, which is defined as follows:

$$\begin{gathered}(m,n) \preceq (m',n') \quad \text{iff either} \\ m = m' \text{ and } n = n', \text{ or} \\ m < m', \text{ or} \\ m = m' \text{ and } n < n'.\end{gathered}$$

We write $(m,n) \prec (m',n')$ when $(m,n) \preceq (m',n')$ and $(m,n) \neq (m',n')$.

We prove that $A(m,n)$ is defined for all $(m,n) \in \mathbb{N} \times \mathbb{N}$ by complete induction over the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$.

In the base case, $(m, n) = (0, 0)$, and since $A(0, n) = n + 1$, we have $A(0, 0) = 1$, and $A(0, 0)$ is defined.

For $(m, n) \neq (0, 0)$, the induction hypothesis is that $A(m', n')$ is defined for all $(m', n') \prec (m, n)$. We need to conclude that $A(m, n)$ is defined.

If $m = 0$, since $A(0, n) = n + 1$, $A(0, n)$ is defined.

If $m \neq 0$ and $n = 0$, since

$$(m - 1, 1) \prec (m, 0),$$

by the induction hypothesis, $A(m - 1, 1)$ is defined, but $A(m, 0) = A(m - 1, 1)$, and thus $A(m, 0)$ is defined.

If $m \neq 0$ and $n \neq 0$, since

$$(m, n-1) \prec (m, n),$$

by the induction hypothesis, $A(m, n-1)$ is defined. Since

$$(m-1, A(m, n-1)) \prec (m, n),$$

by the induction hypothesis, $A(m-1, A(m, n-1))$ is defined. But
$A(m, n) = A(m-1, A(m, n-1))$, and thus $A(m, n)$ is defined.

Thus, $A(m, n)$ is defined for all $(m, n) \in \mathbb{N} \times \mathbb{N}$. It is possible to show that A is a recursive function, although the quickest way to prove it requires some fancy machinery (the recursion theorem).

Proving that A is not primitive recursive is harder.

We can also deal with languages.

4.8. Recursively Enumerable Languages and Recursive Languages

We define the recursively enumerable languages and the recursive languages. We assume that the TM's under consideration have a tape alphabet containing the special symbols 0 and 1.

Definition 4.8.1 Let $\Sigma = \{a_1, \ldots, a_N\}$. A language $L \subseteq \Sigma^*$ is *recursively enumerable (for short, an r.e. set)* iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, either M halts in a proper ID with the output 0, or it runs forever. A language $L \subseteq \Sigma^*$ is *recursive* iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, M halts in a proper ID with the output 0.

Thus, given a recursively enumerable language L, for some $w \notin L$, it is possible that a TM accepting L runs forever on input w. On the other hand, for a recursive language L, a TM accepting L always halts in a proper ID.

When dealing with languages, it is often useful to consider *nondeterministic Turing machines*. Such machines are defined just like deterministic Turing machines, except that their transition function δ is just a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

with no particular extra condition.

It can be shown that every nondeterministic Turing machine can be simulated by a deterministic Turing machine, and thus, nondeterministic Turing machines also accept the class of r.e. sets.

It can be shown that a recursively enumerable language is the range of some recursive function. It can also be shown that a language L is recursive iff both L and its complement are recursively enumerable. There are recursively enumerable languages that are not recursive.

Turing machines were invented by Turing around 1935. The primitive recursive functions were known to Hilbert circa 1890. Gödel formalized their definition in 1929. The partial recursive functions were defined by Kleene around 1934. Church also introduced the λ-calculus as a model of computation around 1934. Other models: Post systems, Markov systems. The equivalence of the various models of computation was shown around 1935/36. RAM programs were only defined around 1963.

A further study of the partial recursive functions requires the notions of pairing functions and of universal functions (or universal Turing machines).

First, we prove the following lemma showing that restricting ourselves to total functions is too limiting:

Let $\mathcal{F}$ be any set of total functions that contains the base functions and is closed under composition and primitive recursion (and thus, $\mathcal{F}$ contains all the primitive recursive functions).

We say that a function $f\colon \Sigma^* \times \Sigma^* \to \Sigma^*$ is *universal* for the one-argument functions in $\mathcal{F}$ iff for every function $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, there is some $n \in \mathbb{N}$ such that

$$f(a_1^n, u) = g(u)$$

for all $u \in \Sigma^*$.

Lemma 4.8.2 *For any countable set $\mathcal{F}$ of total functions containing the base functions and closed under composition and primitive recursion, if f is a universal function for the functions $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, then $f \notin \mathcal{F}$.*

Thus, either a universal function for $\mathcal{F}$ is partial, or it is not in $\mathcal{F}$.

Proof. Assume that the universal function f is in $\mathcal{F}$. Let g be the function such that

$$g(u) = f(a_1^{|u|}, u)a_1$$

for all $u \in \Sigma^*$. We claim that $g \in \mathcal{F}$.

It it enough to prove that the function h such that

$$h(u) = a_1^{|u|}$$

is primitive recursive, which is easily shown.

Then, because f is universal, there is some m such that

$$g(u) = f(a_1^m, u)$$

for all $u \in \Sigma^*$. Letting $u = a_1^m$, we get

$$g(a_1^m) = f(a_1^m, a_1^m) = f(a_1^m, a_1^m)a_1,$$

a contradiction. □