Chapter 5

RAM Programs, Turing Machines, and the Partial Recursive Functions

5.1 Partial Functions and RAM Programs

We define an abstract machine model for computing functions

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

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet.

Numerical functions $f\colon \mathbb{N}^n \to \mathbb{N}$ 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 as $N1, \ldots, Nq$.

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**.

Here is an example of a RAM program to concatenate two strings x_1 and x_2.

```
     R3    ←       R1
     R4    ←       R2
N0   R4    jmp$_a$  N1b
     R4    jmp$_b$  N2b
           jmp     N3b
N1         add$_a$  R3
           tail    R4
           jmp     N0a
N2         add$_b$  R3
           tail    R4
           jmp     N0a
N3   R1    ←       R3
        continue
```

Definition 5.1. *RAM programs* are constructed from seven types of *instructions* shown below:

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

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_ja) or (6_jb) is a conditional jump. Let *head* be the function defined as follows:

$$\begin{aligned} head(\epsilon) &= \epsilon, \\ head(a_ju) &= a_j. \end{aligned}$$

The effect of (6_ja) 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_jb) 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_jb) 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 5.2. A *RAM program* P is a finite sequence of instructions as in Definition 5.1, 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.

For the purpose of computing a function $f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$ 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     ←        R1
        R4     ←        R2
N0      R4     jmp$_a$  N1b
        R4     jmp$_b$  N2b
               jmp      N3b
N1             add$_a$  R3
               tail     R4
               jmp      N0a
N2             add$_b$  R3
               tail     R4
               jmp      N0a
N3      R1     ←        R3
            continue
```

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

Definition 5.3. 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 $u \in \Sigma^*$:

```
add_j        R1
continue
```

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 $1 \leq i \leq n$:

```
R1 ←         Ri
   continue
```

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.

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

Lemma 5.1. *Every RAM program can be converted to an equivalent program only using the following type of instructions:*

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

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

$$Ri \leftarrow Rj$$

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

5.2 Definition of a Turing Machine

We define a Turing machine model for computing functions

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

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 $\Sigma \subseteq \Gamma$. 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.

Definition 5.4. 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. $\Sigma \subseteq \Gamma$, $K \cap \Gamma = \emptyset$, 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

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

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.

Here is an example of a Turing machine.

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

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

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

The instructions in δ are:

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

5.3 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 5.5. 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 5.6. 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 machine computes a partial function

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

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 5.7. 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

$$q_0 w_1 \mathbf{,} w_2 \mathbf{,} \ldots \mathbf{,} w_n$$

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 5.8. 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 5.9. A Turing machine

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

computes the partial function

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

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:

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

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 $bbbabaa$.

5.4 RAM-computable functions are Turing-computable

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

Theorem 5.2. *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 5.1 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.

5.5 Turing-computable functions are RAM-computable

RAM programs can also simulate Turing machines.

Theorem 5.3. *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 (see Section 5.8).

For this, we need to define the *primitive recursive functions*.

Informally, a primitive recursive function is a total recursive function that can be computed using only **for** loops, that is, loops in which the number of iterations is fixed (unlike a **while** loop).

A formal definition of the primitive functions is given in Section 5.7.

Definition 5.10. Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *partial recursive functions* is the class of partial functions (over Σ^*) that can be computed by RAM programs (or equivalently by Turing machines).

The class of *(total) recursive functions* is the subset of the class of partial recursive functions consisting of functions defined for every input (i.e., total functions).

We can also deal with languages.

5.6 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 5.11. 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 (they are a slight generalization of Post system).

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