CENG 280 - Chapter 4: Turing Machines

Burak Metehan Tunçel - June 2022

1 The Definition of a Turing Machine

A **Turing machine** consists of a *finite control*, a *tape*, and a *head* that can be used for *reading or writing* on that tape.

So the Turing machines seem to form a *stable* and *maximal* class of computational devices, in terms of the computations they can perform. The important points to remember by way of introduction are that Turing machines are designed to satisfy simultaneously these three criteria:

(a) They should be automata; that is, their construction and function should be in the same general spirit as the devices previously studied.

(b) They should be as simple as possible to describe, define formally, and reason about.

(c) They should be as general as possible in terms of the computations they can carry out.

In essence, a Turing machine consists of a finite-state control unit and a tape (see Figure 1).

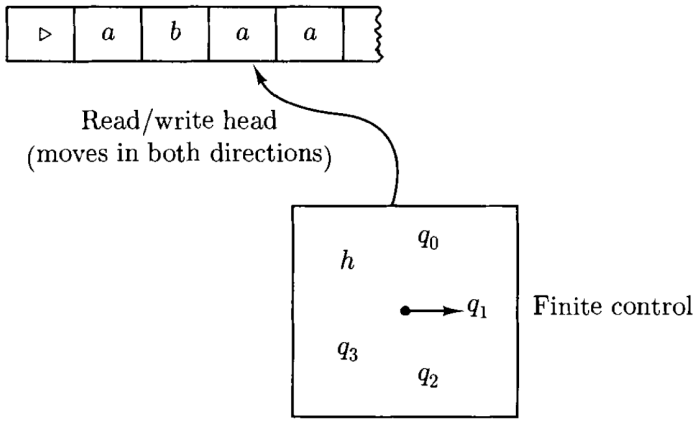

Figure 1

Communication between the two is provided by a single head, which reads symbols from the tape and is also used to change the symbols on the tape. The control unit operates in discrete steps; at each step it performs two functions in a way that depends on its current state and the tape symbol currently scanned by the read/write head:

1. Put the control unit in a new state.
2. Either:
   (a) Write a symbol in the tape square currently scanned, replacing the one already there; or
   (b) Move the read/write head one tape square to the left or right.

The tape has a left end, but it extends indefinitely to the right. To prevent the machine from moving its head off the left end of the tape, we assume that the leftmost end of the tape is always marked by a special symbol denoted by $\triangleright$; we assume further that all of our Turing machines are so designed that, whenever the head reads a $\triangleright$, it immediately moves to the right. Also, we shall use the distinct symbols $\leftarrow$ and $\rightarrow$ to denote movement of the head to the left or right; we assume that these two symbols are not members of any alphabet we consider.

A Turing machine is supplied with input by inscribing that input string on tape squares at the left end of the tape, immediately to the right of the $\triangleright$ symbol. The rest of the tape initially contains ***blank*** symbols, denoted $\sqcup$. The machine is free to alter its input in any way it sees fit, as well as to write on the unlimited blank portion of the tape to the right. Since the machine can move its head only one square at a time, after any finite computation only finitely many tape squares will have been visited.

We can now present the formal definition of a Turing machine.

Definition 1.1

A Turing machine is a quintuple $(K, \Sigma, \delta, s, H)$, where

- K is a *finite set of* ***states***;
- Σ is an *alphabet*, containing the ***blank symbol*** $\sqcup$ and the ***left end symbol*** $\triangleright$, but not containing the symbols $\leftarrow$ and $\rightarrow$;
- $s \in K$ is the ***initial state***;
- $H \subseteq K$ is the *set of* ***halting states***;
- δ, the ***transition function***, is a function from $(K - H) \times \Sigma$ to $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ such that,
  (a) for all $q \in K - H$, if $\delta(q, \triangleright) = (p, b)$, then $b = \rightarrow$
  (b) for all $q \in K - H$ and $a \in \Sigma$, if $\delta(q, a) = (p, b)$ then $b \neq \triangleright$.

If $q \in K - H$, $a \in \Sigma$, and $\delta(q, a) = (p, b)$, then M, when in state q and scanning symbol a, will enter state p, and

1. if b is a symbol in Σ, M will rewrite the currently scanned symbol a as b, or
2. if b is $\leftarrow$ or $\rightarrow$, M will move its head in direction b. Since δ is a function, the operation of M is deterministic and will stop only when M enters a halting state.

Notice these

- the requirement (a) on δ: When it sees the left end of the tape $\triangleright$, it must move right. This way the leftmost $\triangleright$ is never erased, and M never falls off the left end of its tape.
- By (b), M never writes a $\triangleright$, and therefore $\triangleright$ is the unmistakable sign of the left end of the tape.

In other words, we can think of $\triangleright$ simply as a "*protective barrier*" that prevents the head of M from inadvertently falling off the left end, which does not interfere with the computation of M in any other way. Also notice that δ is not defined on states in H; when the machine reaches a halting state, then its operation stops.

Example 1.1

Consider the Turing machine $M = (K, \Sigma, \delta, s, \{h\})$, where

- $K = \{q_0, q_1, h\}$,
- $\Sigma = \{a, \sqcup, \triangleright\}$,
- $s = q_0$,

and δ is given by the following table.

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, \sqcup)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	(q_0, a)
q_1	$\sqcup$	$(q_0, \rightarrow)$
q_1	$\triangleright$	$(q_1, \rightarrow)$

When M is started in its initial state q_0, it scans its head to the right, changing all a's to $\sqcup$'s as it goes, until it finds a tape square already containing $\sqcup$; then it halts. (Changing a nonblank symbol to the blank symbol will be called ***erasing*** the nonblank symbol.)

To be specific, suppose that M is started with its head scanning the first of four a's, the last of which is followed by a $\sqcup$. Then M will go back and forth between states q_0 and q_1 four times, alternately changing an a to a $\sqcup$ and moving the head right; the first and fifth lines of the table for δ are the relevant ones during this sequence of moves.

At this point, M will find itself in state q_0 scanning $\sqcup$ and, according to the second line of the table, will halt. Note that the fourth line of the table, that is, the value of $\delta(q_1, a)$, is irrelevant, since M can never be in state q_1 scanning an a if it is started in state q_0. Nevertheless, some value must be associated with $\delta(q_1, a)$ since δ is required to be a function with domain $(K - H) \times \Sigma$.

Example 1.2

Consider the Turing machine $M = (K, \Sigma, \delta, s, \{h\})$, where

- $K = \{q_0, h\}$,
- $\Sigma = \{a, \sqcup, \triangleright\}$,
- $s = q_0$,

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_0, \leftarrow)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$

This machine scans to the left until it finds a $\sqcup$ and then halts. If every tape square from the head position back to the left end of the tape contains an a, and of course the left end of the tape contains a $\triangleright$, then M will go to the left end of the tape, and from then on it will indefinitely go back and forth between the left end and the square to its right. Unlike other deterministic devices that we have encountered, *the operation of a Turing machine may never stop.*

We now formalize the operation of a Turing machine. To *specify the status* of a Turing machine computation, we need to *specify the state*, the *contents of the tape*, and *the position of the head*. Since all but a finite initial portion of the tape will be blank, the contents of the tape can be specified by a finite string. We choose to break that string into two pieces:

- the part to the left of the scanned square, including the single symbol in the scanned square; and
- the part, possibly empty, to the right of the scanned square.

Moreover, so that no two such pairs of strings will correspond to the same combination of head position and tape contents, we insist that the second string not end with a blank (all tape squares to the right of the last one explicitly represented are assumed to contain blanks anyway). These considerations lead us to the following definitions.

Definition 1.2

A **configuration** of a Turing machine $M = (K, \Sigma, \delta, s, H)$ is a member of

$$K \times \triangleright \Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{e\})$$

That is, all configurations are assumed to start with the left end symbol, and never end with a blank -unless the blank is currently scanned. Thus $(q, \triangleright a, aba)$, $(h, \triangleright \sqcup\sqcup\sqcup, \sqcup a)$, and $(q, \triangleright \sqcup a \sqcup\sqcup, e)$ are configurations (see Figure 2), but $(q, \triangleright baa, a, bc\sqcup)$ and $(q, \sqcup aa, ba)$ are not. A configuration whose state component is in H will be called a ***halted configuration***.

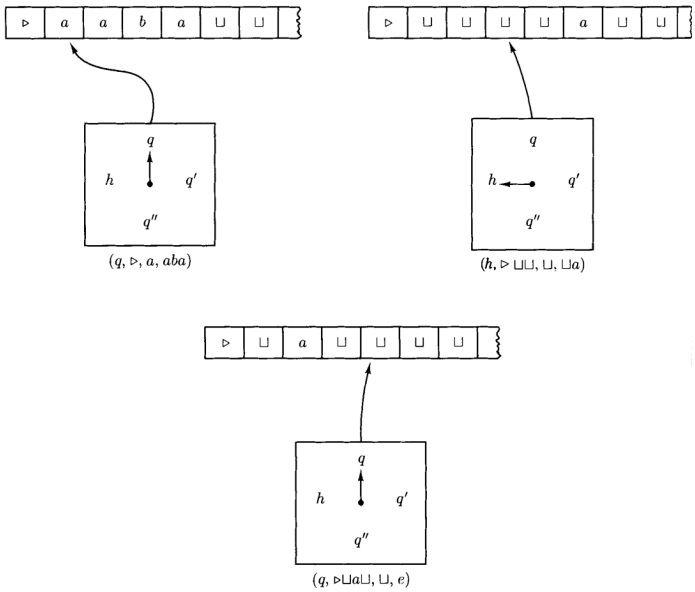

Figure 2

We shall use a simplified notation for depicting the tape contents (including the position of the head): We shall write $w\underline{a}u$ for the tape contents of the configuration (q, wa, u); the underlined symbol indicates the head position. For the three configurations illustrated in Figure 2, the tape contents would be represented as "$\triangleright\underline{a}aba$", "$\triangleright \sqcup \sqcup\underline{\sqcup}\sqcup a$", and "$\triangleright\sqcup a\sqcup\underline{\sqcup}$".

Also, we can write configurations by including the state together with the notation for the tape and head position. That is, we can write (q, wa, u) as $(q, w\underline{a}u)$. Using this convention, we would write the three configurations shown in Figure 2 as "$(q, \triangleright\underline{a}aba)$", "$(h, \triangleright \sqcup \sqcup\underline{\sqcup}\sqcup a)$", and "$(q, \triangleright\sqcup a\sqcup\underline{\sqcup})$".

Definition 1.3

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine and consider two configurations of M, $(q_1, w_1\underline{a_1}u_1)$ and $(q_2, w_2\underline{a_2}u_2)$, where $a_1, a_2 \in \Sigma$. Then

$$(q_1, w_1\underline{a_1}u_1) \vdash_M (q_2, w_2\underline{a_2}u_2)$$

if and only if, for some $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, $\delta(q_1, a_1) = (q_2, b)$, and either

1. $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$, and $a_2 = b$, or
2. $b = \leftarrow$, $w_1 = w_2a_2$, and either
   (a) $u_2 = a_1u_1$, if $a_1 \neq \sqcup$ or $u_1 \neq e$, or
   (b) $u_2 = e$, if $a_1 = \sqcup$ and $u_1 = e$
3. $b = \rightarrow$, $w_2 = w_1a_1$, and either
   (a) $u_1 = a_2u_2$, or
   (b) $u_1 = u_2 = e$, and $a_2 = \sqcup$

- In Case 1, M rewrites a symbol without moving its head.
- In Case 2, M moves its head one square to the left; if it is moving to the left off blank tape, the blank symbol on the square just scanned disappears from the configuration.
- In Case 3, M moves its head one square to the right; if it is moving onto blank tape, a new blank symbol appears in the configuration as the new scanned symbol.

Notice that all configurations, except for the halted ones, yield exactly one configuration.

Example 1.3

To illustrate these cases, let $w, u \in \Sigma^*$, where u does not end with a $\sqcup$, and let $a, b \in \Sigma$.

- *Case 1*: $\delta(q_1, a) = (q_2, b)$.
  Example: $(q_1, w\underline{a}u) \vdash_M (q_2, w\underline{b}u)$.
- *Case 2*: $\delta(q_1, a) = (q_2, \leftarrow)$.
  Example for (a): $(q_1, wb\underline{a}u) \vdash_M (q_2, w\underline{b}au)$.
  Example for (b): $(q_1, wb\underline{\sqcup}) \vdash_M (q_2, w\underline{b})$.
- *Case 3*: $\delta(q_1, a) = (q_2, \rightarrow)$.
  Example for (a): $(q_1, w\underline{a}bu) \vdash_M (q_2, wa\underline{b}u)$.
  Example for (b): $(q_1, w\underline{a}) \vdash_M (q_2, wa\underline{\sqcup})$.

Definition 1.4

For any Turing machine M, let, $\vdash_M^*$ be the reflexive, transitive closure of $\vdash_M$; we say that configuration C_1 **yields** configuration C_2 if $C_1 \vdash_M^* C_2$. A **computation** by M is a sequence of configurations $C_0, C_1, \ldots, C_n$, for some $n \geq 0$ such that

$$c_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_n$$

We say that the computation is of length n or that it has n steps, and we write $C_0 \vdash_M^n C_n$.

Consider the Turing machine M described in Example 1.1. If M is started in configuration $(q_1, \triangleright\underline{\sqcup}aaaa)$, its computation would be represented formally as follows.

$$\begin{aligned}
(q_1, \triangleright\underline{\sqcup}aaaa) &\vdash_M (q_0, \triangleright\sqcup\underline{a}aaa)\\
&\vdash_M (q_1, \triangleright\sqcup\underline{\sqcup}aaa)\\
&\vdash_M (q_0, \triangleright\sqcup\sqcup\underline{a}aa)\\
&\vdash_M (q_1, \triangleright\sqcup\sqcup\underline{\sqcup}aa)\\
&\vdash_M (q_0, \triangleright\sqcup\sqcup\sqcup\underline{a}a)\\
&\vdash_M (q_1, \triangleright\sqcup\sqcup\sqcup\underline{\sqcup}a)\\
&\vdash_M (q_0, \triangleright\sqcup\sqcup\sqcup\sqcup\underline{a})\\
&\vdash_M (q_1, \triangleright\sqcup\sqcup\sqcup\sqcup\underline{\sqcup})\\
&\vdash_M (q_0, \triangleright\sqcup\sqcup\sqcup\sqcup\sqcup\underline{\sqcup})\\
&\vdash_M (h, \triangleright\sqcup\sqcup\sqcup\sqcup\sqcup\underline{\sqcup})
\end{aligned}$$

The computation has 10 steps.

1.1 A Notation for Turing Machines

We need a notation for Turing machines that is more graphic and transparent. we shall use a *hierarchical* notation, in which more and more complex machines are built from simpler materials. To this end, we shall define a very simple repertoire of *basic machines*, together with *rules for combining machines*.

The Basic Machines. We start from very humble beginnings: The *symbol-writing machines* and the *head-moving machines*.

Let us fix the alphabet Σ of our machines. For each $a \in \Sigma \cup \{\rightarrow, \leftarrow\} - \{\triangleright\}$, we define a Turing machine $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$, where for each $b \in \Sigma - \{\triangleright\}$, $\delta(s, b) = (h, a)$.

Naturally, $\delta(s, \triangleright)$ is still always $(s, \rightarrow)$. That is, the only thing this machine does is to perform action a (writing symbol a if $a \in \Sigma$, moving in the direction indicated by a if $a \in \{\leftarrow, \rightarrow\}$) and then to immediately halt.

Naturally, there is a unique exception to this behavior: If the scanned symbol is a $\triangleright$, then the machine will dutifully move to the right.

Because the symbol-writing machines are used so often, we abbreviate their names and write simply a instead of M_a. That is, if $a \in \Sigma$, then the a-writing machine will be denoted simply as a. The head-moving machines $M_\leftarrow$ and $M_\rightarrow$ will be abbreviated as L (for "*left*") and R (for "*right*").

The Rules for Combining Machines. Turing machines will be combined in a way suggestive of the structure of a finite automaton. Individual machines are like the states of a finite automaton, and the machines may be connected to each other in the way that the states of a finite automaton are connected together. However, the connection from one machine to another is not pursued until the first machine halts; the other machine is then started from its initial state with the tape and head position as they were left by the first machine.

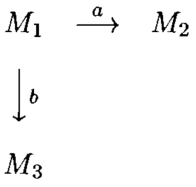

Figure 3

So if M_1, M_2, and M_3 are Turing machines, the machine displayed in Figure 3 operates as follows:

> *Start in the initial state of M_1; operate as M_1 would operate until M_1 would halt; then, if the currently scanned symbol is an a, initiate M_2 and operate as M_2 would operate; otherwise, if the currently scanned symbol is a b, then initiate M_3 and operate as M_3 would operate.*

Let us take the machine shown in Figure 3 above. Suppose that the three Turing machines M_1, M_2, and M_3 are $M_1 = (K_1, \Sigma, \delta_1, s_1, H_1)$, $M_2 = (K_2, \Sigma, \delta_2, s_2, H_2)$ and $M_3 = (K_3, \Sigma, \delta_3, s_3, H_3)$. We shall assume, as it will be most convenient in the context of combining machines, that the sets of states of all these machines are disjoint. The combined machine shown in Figure 3 above would then be $M = (K, \Sigma, \delta, s, H)$, where

$$K = K_1 \cup K_2 \cup K_3,$$

$$s = s_1,$$

$$H = H_2 \cup H_3.$$

For each $\sigma \in \Sigma$ and $q \in K - H$, $\delta(q, \sigma)$ is defined as follows:

(a) If $q \in K_1 - H_1$, then $\delta(q, \sigma) = \delta_1(q, \sigma)$.

(b) If $q \in K_2 - H_2$, then $\delta(q, \sigma) = \delta_2(q, \sigma)$.

(c) If $q \in K_3 - H_3$, then $\delta(q, \sigma) = \delta_3(q, \sigma)$.

(d) Finally, if $q \in H_1$ (the only case remaining) then $\delta(q, \sigma) = s_2$ if $\sigma = a$, $\delta(q, \sigma) = s_3$ if $\sigma = b$, and $\delta(q, \sigma) \in H$ otherwise.

Example 1.4

Figure 4(a) illustrates a machine consisting of two copies of R. The machine represented by this diagram moves its head right one square; then, if that square contains an a, or a b, or a $\triangleright$, or a $\sqcup$, it moves its head one square further to the right.

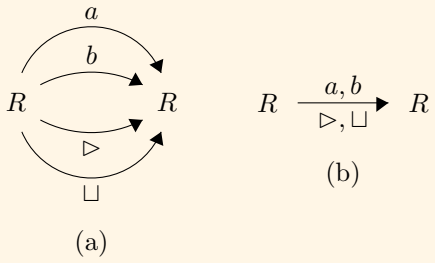

Figure 4

It will be convenient to represent this machine as in Figure 4(b). If an arrow is labeled by *all* symbols in the alphabet Σ of the machines, then the labels can be omitted.

where, by convention, the leftmost machine is always the initial one. Sometimes an unlabeled arrow connecting two machines can be omitted entirely, by juxtaposing the representations of the two machines. Under this convention, the above machine becomes simply RR, or even R^2.

Example 1.5

If $a \in \Sigma$ is any symbol, we can sometimes eliminate multiple arrows and labels by using $\overline{a}$ to mean "any symbol except a". Thus, the machine shown in Figure 5(a) scans its tape to the right until it finds a blank. We shall denote this most useful machine by $R_\sqcup$. (The reason why it is named as right scanning is that the label is R. If label is L, then it is named as scanning to the left.)

Figure 5

Another shorthand version of the same machine as in Figure 5(a) is shown in Figure 5(b). Here $a \neq \sqcup$ is read "any symbol a other than $\sqcup$". The advantage of this notation is that a may then be used elsewhere in the diagram as the name of a machine. To illustrate, Figure 6 depicts a machine that scans to the right until it finds a nonblank square, then copies the symbol in that square onto the square immediately to the left of where it was found.

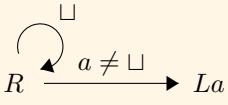

Figure 6

Note: *Check the example 4.1.7 - 4.1.10 from textbook. They are a little basic but important.*

2 Computing with Turing Machines

We adopt the following policy for presenting input to Turing machines:

- The input string which has *no blank symbols in it*, is written to the right of the leftmost symbol $\triangleright$, with a *blank to its left*, and *blanks to its right*;
- The head is positioned at the tape square containing the blank between the $\triangleright$ and the input;
- The machine starts operating in its initial state.

If $M = (K, \Sigma, \delta, s, H)$ is a Turing machine and $w \in (\Sigma - \{\sqcup, \triangleright\})^*$, then the **initial configuration of M on input w** is $(s, \triangleright\sqcup w)$. With this convention, we can now define how Turing machines are used as language recognizers.

Definition 2.1

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, such that $H = \{y, n\}$ consists of two distinguished halting states (y and n for "yes" and "no"). Any halting configuration whose state component is y is called an **accepting configuration**, while a halting configuration whose state component is n is called a **rejecting configuration**.

We say that M **accepts** an input $w \in (\Sigma - \{\sqcup, \triangleright\})^*$ if $(s, \triangleright\sqcup w)$ yields an accepting configuration; we say that M **rejects** w if $(s, \triangleright\sqcup w)$ yields a rejecting configuration.

Let $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$ be an alphabet, called the **input alphabet** of M; by fixing Σ_0 to be a subset of $\Sigma - \{\sqcup, \triangleright\}$, we allow our Turing machines to use extra symbols during their computation, besides those appearing in their inputs. We say that M **decides** a language $L \subseteq \Sigma_0^*$ if for any string $w \in \Sigma_0^*$ the following is true: "If $w \in L$ then M accepts w; and if $w \notin L$ then M rejects w".

Finally, call a language L **recursive** if there is a Turing machine that decides it.

That is, a Turing machine decides a language L if, when started with input w, it always halts, and does so in a halt state that is the correct response to the input: y if $w \in L$, n if $w \notin L$. Notice that no guarantees are given about what happens if the input to the machine contains blanks or the left end symbol.

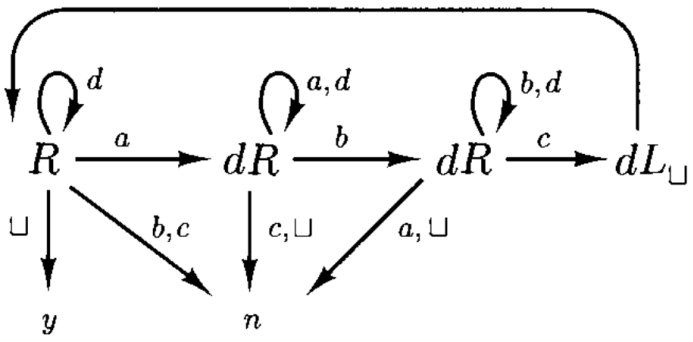

Figure 7

Example 2.1

Consider the language $L = \{a^n b^n c^n \mid n \geq 0\}$, which has heretofore evaded all types of language recognizers. The Turing machine whose diagram is shown in Figure 7 decides L. In this diagram we have also utilized two new basic machines, useful for deciding languages: Machine y makes the new state to be the accepting state y, while machine n moves the state to n.

The strategy employed by M is simple: On input $a^n b^n c^n$ it will operate in n stages. In each stage M starts from the left end of the string and moves to the right in search of an a.

- When it finds an a, it replaces it by a d and then looks further to the right for a b.
- When a b is found, it is replaced by a d, and the machine then looks for a c.
- When a c is found and is replaced by a d, then the stage is over, and the head returns to the left end of the input. Then the next stage begins.

That is, at each stage the machine replaces an a, a b, and a c by d's. If at any point the machine senses that the string is not in $a^*b^*c^*$, or that there is an excess of a certain symbol (for example, if it sees a b or c while looking for an a), then it enters state n and rejects immediately.

If however it encounters the right end of the input while looking for an a, this means that all the input has been replaced by d's, and hence it was indeed of the form $a^n b^n c^n$, for some $n \geq 0$. The machine then accepts.

There is a subtle point in relation to Turing machines that decide languages: With the other language recognizers that we have seen so far (even the nondeterministic ones), one of two things could happen: Either

- the machine accepts the input, or
- the machine rejects the input.

A Turing machine, on the other hand, even if it has only two halt states y and n, always has the option of evading an answer ("yes" or "no"), by failing to halt. Given a Turing machine, it might or it might not decide a language (and there is no obvious way to tell whether it does). The far-reaching importance (and necessity) of this deficiency will become apparent later in this chapter, and in the next.

2.1 Recursive Functions

Since Turing machines can write on their tapes, they can provide more elaborate output than just a "yes" or a "no":

Definition 2.2

Let $M = (K, \Sigma, \delta, s, \{h\})$ be a Turing machine, let $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$ be an alphabet, and let $w \in \Sigma_0^*$. Suppose that M halts on input w, and that $(s, \triangleright\sqcup w) \vdash_M^* (h, \triangleright\sqcup y)$ for some $y \in \Sigma_0^*$. Then y is called the **output** of M on input w, and is denoted $M(w)$. Notice that $M(w)$ is defined *only if* M *halts on input* w, and in fact does so at a configuration of the form $(h, \triangleright\sqcup y)$ with $y \in \Sigma_0^*$.

Now let f be any function from Σ_0^* to Σ_0^*. We say that M **computes** function f if, for all $w \in \Sigma_0^*$, $M(w) = f(w)$. That is, for all $w \in \Sigma_0^*$, M eventually halts on input w, and when it does halt, its tape contains the string $\triangleright\sqcup f(w)$. A function f is called **recursive**, if there is a Turing machine M that computes f.

Example 2.2

The function $\kappa : \Sigma^* \mapsto \Sigma^*$ defined as $\kappa(w) = ww$ can be computed by the machine $CS_{\leftarrow}$, that is, the copying machine followed by the left-shifting machine.

Strings in $\{0,1\}^*$ can be used to represent the nonnegative integers in the familiar *binary notation*. Any string $w = a_1 a_2 \ldots a_n \in \{0,1\}^*$ represents the number

$$num(w) = a_1 \cdot 2^{n-1} + a_2 \cdot 2^{n-2} + \ldots + a_n.$$

And any natural number can be represented in a unique way by a string in $0 \cup 1(0 \cup 1)^*$ (that is to say, without redundant 0's in the beginning).

Accordingly, Turing machines computing functions from $\{0,1\}^*$ to $\{0,1\}^*$ can be thought of as computing functions from the natural numbers to the natural numbers. In fact, numerical functions with many arguments (such as addition and multiplication) can be computed by Turing machines computing functions from $\{0,1,;\}^*$ to $\{0,1\}^*$, where ";" is a symbol used to separate binary arguments.

Definition 2.3

Let $M = (K, \Sigma, \delta, s, \{h\})$ be a Turing machine such that $0, 1, ; \in \Sigma$, and let f be any function from $\mathbf{N^k}$ to $\mathbf{N}$ for some $k \geq 1$. We say that M **computes** function f if for all $w_1, \ldots, w_k \in 0 \cup 1(0 \cup 1)^*$ (that is, for any k strings that are binary encodings of integers), $\mathtt{num}(M(w_1; \ldots; w_k)) = f(num(w_1), ..., num(w_k))$.

That is, if M is started with the binary representations of the integers $n_1, ..., n_k$ as input, then it eventually halts, and when it does halt, its tape contains a string that represents number $f(n_1, ..., n_k)$ -the value of the function. A function $f : N^k \mapsto N$ is called **recursive** if there is a Turing machine M that computes f.

On our inability to tell whether a Turing machine decides a language, also applies to function computation. The price we must pay for the very broad range of functions that Turing machines can compute, is that we cannot tell whether a given Turing machine indeed computes such a function -that is to say, whether it halts on all inputs.

2.2 Recursively Enumerable Languages

If a Turing machine decides a language or computes a function, it can be reasonably thought of as an algorithm that performs correctly and reliably some computational task. We next introduce a third, subtler, way in which a Turing machine can define a language:

Definition 2.4

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, let $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$ be an alphabet, and let $L \subseteq \Sigma_0^*$ be a language. We say that M **semidecides** L if for any string $w \in \Sigma_0^*$ the following is true:

$w \in L$ if and only if M halts on input w.

A language L is **recursively enumerable** if and only if there is a Turing machine M that semidecides L.

Thus when M is presented with input $w \in L$, it is required to halt eventually. We *do not care precisely which halting configuration it reaches*, as long as it does eventually arrive at a halting configuration. If however $w \in \Sigma_0^* - L$, then M must never enter the halting state; that is, the machine must continue its computation indefinitely.

Extending the "functional" notation of Turing machines that we introduced in the previous subsection (which allows us to write equations such as $M(w) = v$), we shall write $M(w) = \nearrow$ if M fails to halt on input w. In this notation, we can restate the definition of semidecision of a language $L \subseteq \Sigma_0^*$ by Turing machine M as follows:

For all $w \in \Sigma_0^*$, $M(w) = \nearrow$ if and only if $w \notin L$.

Example 2.3

Let $L = \{w \in \{a,b\}^* \mid w \text{ contains at least one } a\}$. Then L is semidecided by the Turing machine shown in Figure 8.

Figure 8

This machine, when started in configuration $(q_0, \triangleright\sqcup w)$ for some $w \in \{a,b\}^*$, simply scans right until an a is encountered and then halts. If no a is found, the machine goes on forever into the blanks that follow its input, never halting.

So L is exactly the set of strings w in $\{a,b\}^*$ such that M halts on input w. Therefore M semidecides L, and thus L is recursively enumerable.

The definition of semidecision by Turing machines is a rather straightforward extension of the notion of acceptance for the deterministic finite automaton. There is a major difference, however. A finite automaton always halts when it has read all of its input -the question is whether it halts on a final or a non final state. In this sense it is a useful computational device, an *algorithm* from which we can reliably obtain answers as to whether an input belongs in the accepted language: We wait until all of the input has been read, and we then observe the state of the machine.

In contrast, a Turing machine that semidecides a language L cannot be usefully employed for telling whether a string w is in L, because, if $w \notin L$, then *we will never know when we have waited enough for an answer*. Turing machines that semidecide languages are no algorithms.

$\{a^n b^n c^n \mid n \geq 0\}$ is a recursive language. But is it recursively enumerable? $\{a^n b^n c^n \mid n \geq 0\}$ is also recursively enumerable. Actually, *Any recursive language is also 1'ecursively enumerable* as stated in theorem 2.1. All it takes in order to construct another Turing machine that semidecides, instead of decides, the language is to make the rejecting state n a nonhalting state, from which the machine is guaranteed to never halt.

Specifically, given any Turing machine $M = (K, \Sigma, \delta, s, \{y, n\})$ that decides L, we can define a machine M' that semidecides L as follows: $M = (K, \Sigma, \sigma', s, \{y\})$, where δ' is just δ augmented by the following transitions related to n -no longer a halting state: $\delta'(n, a) = (n, a)$ for all $a \in \Sigma$.

It is clear that if M indeed decides L, then M' semidecides L, because M' accepts the same inputs as M; furthermore, if M rejects an input w, then M' does not halt on w (it "loops forever" in state n). In other words, for all inputs w, $M'(w) = \nearrow$ if and only if $M(w) = n$.

Theorem 2.1

If a language is recursive, then it is recursively enumerable.

Naturally, the interesting (and difficult) question is the opposite:

> Can we always transform every Turing machine that semidecides a language into an actual algorithm for deciding the same language?

We shall see in the next chapter that the answer here is negative:

> There are recursively enumerable languages that are not recursive.

An important property of the class of recursive languages is that it is closed under complement:

Theorem 2.2

If L is a recursive language, then its complement $\overline{L}$ is also recursive.

Proof. If L is decided by Turing machine $M = (K, \Sigma, \delta, s, \{y, n\})$, then L is decided by the Turing machine $M' = (K, \Sigma, \delta', s, \{y, n\})$ which is identical to M *except that it reverses the roles of the two special halting states* y *and* n. That is, δ' is defined as follows:

$$\delta'(q, a) = \begin{cases} n & \text{if } \delta(q, a) = y \\ y & \text{if } \delta(q, a) = n \\ \delta(q, a) & \text{otherwise} \end{cases}$$

It is clear that $M'(w) = y$ if and only if $M(w) = n$, and therefore M' decides $\overline{L}$. □

3 Extensions of Turing Machine

3.1 Multiple Tapes

One can think of Turing machines that have several tapes (see Figure 9). Each tape is connected to the finite control by means of a read/write head (one on each tape). The machine can in one step read the symbols scanned by all its heads and then, depending on those symbols and its current state, rewrite some of those scanned squares and move some of the heads to the left or right, in addition to changing state.

For any fixed integer $k \geq 1$, a k-tape Turing machine is a Turing machine equipped as above with k tapes and corresponding heads. Thus a "standard" Turing machine studied so far in this chapter is just a k-tape Turing machine, with $k = 1$.

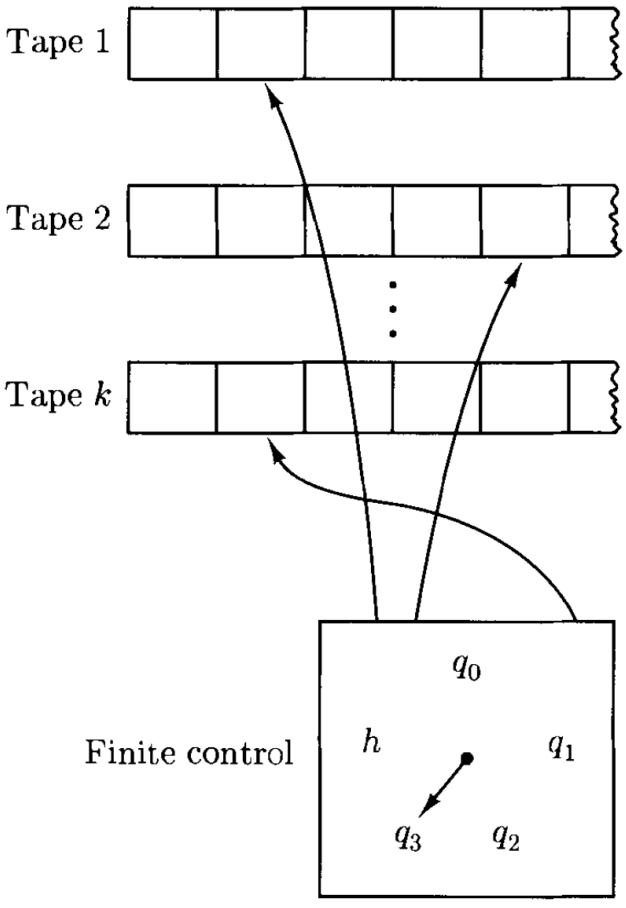

Figure 9

Definition 3.1

Let $k \geq 1$ be an integer. A **k-tape Turing machine** is a quintuple $(K, \Sigma, \delta, s, H)$, where K, Σ, s, and H are as in the definition of the ordinary Turing machine, and δ, the **transition function**, is a function from $(K - H) \times \Sigma^k$ to $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})^k$. That is, for each state q, and each k-tuple of tape symbols $(a_1, \ldots, a_k)$; $\delta(q, (a_1, \ldots, a_k)) = (p, (b_1, \ldots, b_k))$, where p is, as before, the new state, and b_j is, intuitively, the action taken by M at tape j. Naturally, we again insist that if $a_j = \triangleright$ for some $j \leq k$, then $b_j = \rightarrow$.

Computation takes place in all k tapes of a k-tape Turing machine. Accordingly, a *configuration* of such a machine must include information about all tapes:

Definition 3.2

Let $M = (K, \Sigma, \delta, s, H)$ be a k-tape Turing machine. A configuration of M is a member of

$$K \times (\triangleright\Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{e\}))^k$$

That is, a configuration identifies the *state*, the *tape contents*, and the *head position* in each of the k tapes.

If $(q, (w_1\underline{a_1}u_1, \ldots, w_k\underline{a_k}u_k))$ is a configuration of a k-tape Turing machine (where we have used the k-fold version of the abbreviated notation for configurations), and if $\delta(q, (a_1, \ldots, a_k)) = (p, (b_1, \ldots, b_k))$, then in one move the machine would move to configuration $(p, (w_1'\underline{a_1'}u_1', \ldots, w_k'\underline{a_k'}u_k'))$ (where -for $i = 1, \ldots, k$- $w_i'\underline{a_i'}u_i'$ is $w_i\underline{a_i}u_i$ modified by action b_i, precisely as in Definition 1.3).

We say that configuration $(q, (w_1\underline{a_1}u_1, \ldots, w_k\underline{a_k}u_k))$ *yields in one step configuration* $(p, (w_1'\underline{a_1'}u_1', \ldots, w_k'\underline{a_k'}u_k'))$.

A k-tape Turing machine can be used for *computing a function* or *deciding or semi deciding a language* in any of the ways discussed above for standard Turing machines. We adopt the convention that the input string is placed on the first tape, in the same way as it would be presented to a standard Turing machine. The other tapes are initially blank, with the head on the leftmost blank square of each. At the end of a computation, *a k-tape Turing machine is to leave its output on its first tape; the contents of the other tapes are ignored.*

Multiple tapes often facilitate the construction of a Turing machine to perform a particular function. Consider the following example.

Example 3.1

Consider the transforming $\triangleright\sqcup w\sqcup$ into $\triangleright\sqcup w\sqcup w\sqcup$ where $w \in \{a, b\}^*$. A 2-tape Turing machine can accomplish this as follows.

1. Move the heads on both tapes to the right, copying each symbol on the first tape onto the second tape, until a blank is found on the first tape. The first square of the second tape should be left blank.
2. Move the head on the second tape to the left until a blank is found.
3. Again move the heads on both tapes to the right, this time copying symbols from the second tape onto the first tape. Halt when a blank is found on the second tape.

This sequence of actions can be pictured as follows.

At the beginning:	First tape	$\triangleright\underline{\sqcup}w$
	Second tape	$\triangleright\underline{\sqcup}$
After (1):	First tape	$\triangleright\sqcup w\underline{\sqcup}$
	Second tape	$\triangleright\sqcup w\underline{\sqcup}$
After (2):	First tape	$\triangleright\sqcup w\underline{\sqcup}$
	Second tape	$\triangleright\underline{\sqcup}w$
After (3):	First tape	$\triangleright\sqcup w\sqcup w\underline{\sqcup}$
	Second tape	$\triangleright\sqcup w\underline{\sqcup}$

Turing machines with more than one tape can be depicted in the same way that single-tape Turing machines were depicted in earlier sections. We simply attach as a superscript to the symbol denoting each machine the number of the tape on which it is to operate; all other tapes are unaffected. For example,

- $\sqcup^2$ writes a blank on the second tape,
- $L^1_{\sqcup}$ searches to the left for a blank on the first tape, and
- $R^{1,2}$ moves to the right the heads of both the first and the second tape.
- A label a^1 on an arrow denotes an action taken if the symbol scanned in the vfirst tape is an a.

Using this convention, the 2-tape version of the copying machine might be illustrated as in Figure 10 (example 3.1). We indicate the submachines performing Functions 1 through 3 above.

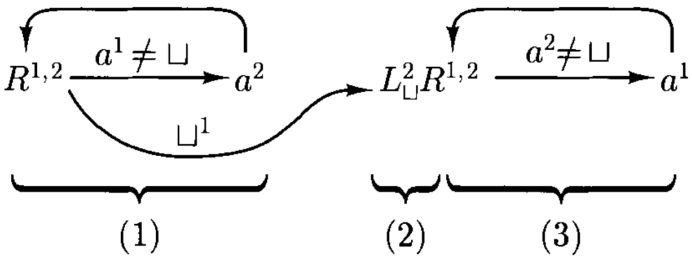

Figure 10

k-tape Turing machines are capable of quite complex computational tasks. We shall show next that any k-tape Turing machine can be simulated by a single-tape machine. By this we mean that, given any k-tape Turing machine, we can design a standard Turing machine that exhibits the same input-output behavior (decides or semidecides the same language, computes the same function).

Such simulations are important ingredients of our methodology in studying the power of computational devices in this and the next chapters. Typically, they amount to a method for mimicking a single step of the simulated machine by several steps of the simulating machine. Our first result of this sort, and its proof, is quite indicative of this line of reasoning.

Theorem 3.1

Let $M = (K, \Sigma, \delta, s, H)$ be a k-tape Turing machine for some $k \geq 1$. Then there is a standard Thring machine $M' = (K', \Sigma', \delta', s', H)$, where $\Sigma \subseteq \Sigma'$, and such that the following holds:

> For any input string $x \in \Sigma^*$, M on input x halts with output y on the first tape if and only if M' on input x halts at the same halting state, and with the same output y on its tape.

Furthermore, if M halts on input x after t steps, then M' halts on input x after a number of steps which is $O(t \cdot (|x| + t))$.

!! *Check the proof of the Theorem 3.1 in the textbook.*

By using the conventions described for the input and output of a k-tape Turing machine, the following result is easily derived from the previous theorem.

Corollary

Any function that is computed or language that is decided or semidecided by a k-tape Turing machine is also computed, decided, or semidecided, respectively, by a standard Turing machine.

3.2 Two-way Infinite Tape

Suppose now that our machine has a tape that is infinite in both directions. All squares are initially blank, except for those containing the input; the head is initially to the left of the input. Also, our convention with the $\triangleright$ symbol would be unnecessary and meaningless for such machines.

It is not hard to see that, like multiple tapes, two-way infinite tapes do not add substantial power to Turing machines. A two-way infinite tape can be easily simulated by a 2-tape machine:

- one tape always contains the part of the tape to the right of the square containing the first input symbol, and
- the other contains the part of the tape to the left of this in reverse.

In turn, this 2-tape machine can be simulated by a standard Turing machine. In fact, the simulation need only take linear, instead of quadratic, time, since at each step only one of the tracks is active. Needless to say, machines with several two-way infinite tapes could also simulated in the same way.

3.3 Multiple Heads

What if we allow a Thring machine to have one tape, but several heads on it? In one step, the heads all sense the scanned symbols and move or write independently. (Some convention must be adopted about what happens when two heads that happen to be scanning the same tape square attempt to write different symbols. Perhaps the head with the lower number wins out. Also, let us assume that the heads cannot sense each other's presence in the same tape square, except perhaps indirectly, through unsuccessful writes.)

It is not hard to see that a simulation like the one we used for k-tape machines can be carried out for Turing machines with several heads on a tape. The basic idea is again to divide the tape into tracks, all but one of which are used solely to record the head positions. To simulate one computational step by the multiple-head machine, the tape must be scanned twice:

- Once to find the symbols at the head positions, and
- again to change those symbols or move the heads as appropriate.

The number of steps needed is again quadratic, as in Theorem 3.1. The use of multiple heads, like multiple tapes, can sometimes drastically simplify the construction of a Turing machine.

3.4 Two-Dimensional Tape

Another kind of generalization of the Turing machine would allow its "tape" to be an infinite two-dimensional grid. (One might even allow a space of higher dimension.) Such a device could be much more useful than standard Turing machines to solve problems such as "zigsaw puzzles". Once again, however, no fundamental increase in power results. Interestingly, the number of steps needed to simulate t steps of the two-dimensional Turing machine on input x by the ordinary Turing machine is again polynomial in t and $|x|$.

The above extensions on the Turing machine model can be combined: One can think of Turing machines *with several tapes, all or some of which are two-way infinite and have more than one head on them*, or are *even multidimensional*. Again, it is quite straightforward to see that the ultimate capabilities of the Turing machine remain the same.

We summarize our discussion of the several variants of Turing machines discussed so far as follows.

Theorem 3.2

Any language decided or semidecided, and any function computed by Turing machines with several tapes, heads, two-way infinite tapes, or multidimensional tapes, can be decided, semidecided, or computed, respectively, by a standard Turing machine.

4 Nondeterministic Turing Machines

Many seemingly powerful features, such as multiple tapes and heads, with no appreciative increase in power in Turing Machine. However, there is an important and familiar feature that was not tried yet: ***nondeterminism***.

Formally, a **nondeterministic Turing machine** is a quintuple $(K, \Sigma, \Delta, s, H)$, where K, Σ, s, and H are as for standard Turing machines, and Δ is a *subset* of $((K-H)\times\Sigma)\times(K\times(\Sigma\cup\{\leftarrow,\rightarrow\}))$, rather than a *function* from $((K-H)\times\Sigma)$ to $(K\times(\Sigma\cup\{\leftarrow,\rightarrow\}))$. Configurations and the relations $\vdash_M$ and $\vdash_M^*$ are defined in the natural way. But now $\vdash_M$ need not be single-valued: One configuration may yield several others in one step.

Since a nondeterministic machine could produce two different outputs or final states from the same input, we have to be careful about what is considered to be the end result of a computation by such a machine. Because of this, it is easiest to consider at first nondeterministic Turing that *semidecide* languages.

Definition 4.1

Let $M = (K, \Sigma, \Delta, s, H)$ be a *nondeterministic* Turing machine. We say that M **accepts** an input $w \in (\Sigma - \{\triangleright, \sqcup\})^*$ if $(s,\ \triangleright\underline{\sqcup}w) \vdash_M (h,\ u\underline{a}v)$ for some $h \in H$ and $a \in \Sigma$; $u, v \in \Sigma^*$.

Notice that a nondeterministic machine accepts an input even though it may have many nonhalting computations on this input input (as long as at least one halting computation exists). We say that M **semidecides a language** $L \subseteq (\Sigma - \{\triangleright, \sqcup\})^*$ if the following holds for all $w \in (\Sigma - \{\triangleright, \sqcup\})^*$: $w \in L$ if and only if M accepts w.

It is a little more subtle to define what it means for a nondeterministic Turing machine to decide a language, or to compute a function.

Definition 4.2

Let $M = (K, \Sigma, \Delta, s, \{y, n\})$ be a nondeterministic Turing machine. We say that M **decides** a language $L \subseteq (\Sigma - \{\triangleright, \sqcup\})^*$ if the following two conditions hold for all $w \in (\Sigma - \{\triangleright, \sqcup\})^*$:

(a) There is a natural number N, depending on M and w, such that there is no configuration C satisfying $(s,\ \triangleright\underline{\sqcup}w) \vdash_M^N C$.

(b) $w \in L$ if and only if $(s,\ \triangleright\underline{\sqcup}w) \vdash_M^* (y,\ u\underline{a}v)$ for some $u, v \in \Sigma^*$, $a \in \Sigma$.

Finally, we say that M **computes** a function $f : (\Sigma - \{\triangleright, \sqcup\})^* \mapsto (\Sigma - \{\triangleright, \sqcup\})^*$ if the following two conditions hold for all $w \in (\Sigma - \{\triangleright, \sqcup\})^*$:

(a) There is an N, depending on M and w, such that there is no configuration C satisfying $(s,\ \triangleright\underline{\sqcup}w) \vdash_M^N C$.

(b) $(s,\ \triangleright\underline{\sqcup}w) \vdash_M^* (h,\ u\underline{a}v)$ if and only if $ua = \triangleright\sqcup$, and $v = f(w)$.

Nondeterminism would seem to be a very powerful feature that cannot be eliminated easily. Indeed, there appears to be no easy way to simulate a nondeterministic Turing machine by a deterministic one in a step-by-step manner, as we have done in all other cases of enhanced Turing machines that we have examined so far. However, the languages semidecided or decided by nondeterministic Turing machines are in fact no different from those semidecided or decided, respectively, by deterministic Turing machines.

Theorem 4.1

If a nondeterministic Turing machine M *semidecides* or *decides a language*, or *computes a function*, then there is a standard Turing machine M' *semideciding or deciding the same language*, or *computing the same function*.

Proof. Let $M = (K, \Sigma, \Delta, s, H)$ be a nondeterministic Turing machine semideciding a language L. Given an input w, M' will attempt to run systematically through all possible computations by M, searching for one that halts. When and if it discovers a halting computation, it too will halt. So M' will halt if and only if M halts, as required.

The proof uses a 3-tape deterministic Turing machine, M_d:

1. The first tape is never changed; it always contains the original input w, so that each simulated computation of M can begin a fresh with the same input.
2. The second and the third tapes are used to simulate the computations of M_d, the deterministic version of M, with all strings in $\{1, 2, \ldots, r\}^*$. The input is copied from the first tape onto the second before M' begins to simulate each new computation. Initially, the third tape contains e, the empty string (and therefore the simulation of M_d will not even start the first time around).
3. Between two simulations of M_d, M' uses a Turing machine N to generate the *lexicographically* next string in $\{1, 2, \ldots, r\}^*$. That is, N will generate from e the strings 1, 2, ..., r, 11, 12, ..., rr, 111, ... For $r = 2$, N is precisely the Turing machine that computes the binary successor function; its generalization to $r > 2$ is rather straightforward.

□

As we had expected, the simulation of a nondeterministic Turing machine by a deterministic one is not a step-by-step simulation. Instead, it goes through all possible computations of the nondeterministic Turing machine. As a result, it requires *exponentially many steps* in n to simulate a computation of n steps by the nondeterministic machine. Whether this long and indirect simulation is an intrinsic feature of nondeterminism, or an artifact of our poor understanding of it, is a deep and important open question.

5 Grammars

Turing machines can be reasonably called *automata*. Like other automata, Turing machines and their extensions act basically as *language acceptors*, receiving an input, examining it, and expressing in various ways their approval or disapproval of it. Two important families of languages, the recursive and the recursively enumerable languages, have resulted.

Additionally, there is another important family of devices (other than language acceptors), very different in spirit from language acceptors, that can be used to define interest-

ing classes of languages: *language generators*, such as regular expressions and context-free grammars. In fact, It has been demonstrated that these two formalisms provide valuable *alternative characterizations* of the classes of languages defined by language acceptors.

We shall now introduce a new kind of language generator that is a generalization of the the context-free grammar, called the **grammar** (or **unrestricted grammar**, to contrast it with the context-free grammars) and show that the class of languages generated by such grammars is precisely the class of recursively enumerable ones.

Definition 5.1

A **grammar** (or **unrestricted grammar**, or a **rewriting system**) is a quadruple $G = (V, \Sigma, R, S)$, where

- V is an alphabet;
- $\Sigma \subseteq V$ is the set of **terminal** symbols, and $V - \Sigma$ is called the set of **nonterminal** symbols;
- $S \in V - \Sigma$ is the **start** symbol; and
- R, the set of **rules**, is a finite subset of $(V^*(V - \Sigma)V^*) \times V^*$.

We write $u \to v$ if $(u,\ v) \in R$; we write $u \Rightarrow_G v$ if and only if, for some $w_1, w_2 \in V^*$ and some rule $u' \to v' \in R$, $u = w_1 u' w_2$ and $v = w_1 v' w_2$.

As usual, $\Rightarrow_G^*$ is the reflexive, transitive closure of $\Rightarrow_G$. A string $w \in \Sigma^*$ is generated by G if and only if $S \Rightarrow_G^* w$; and $L(G)$, the **language generated by** G is the set of all strings in Σ^* generated by G.

We also use other terminology introduced originally for context-free grammars; for example, a **derivation** is a sequence of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$.

Grammars play with respect to Turing machines precisely the same role that context-free grammars play in relation to pushdown automata, and regular expressions to finite automata:

Theorem 5.1

A language is generated by a grammar if and only if it is recursively enumerable.

Proof. ***Read proof from the textbook, pp. 230-231.*** □

Example 5.1

The following grammar G generates the language $\{a^n b^n c^n \mid n \geq 1\}$. $G = (V,\ \Sigma,\ R,\ S)$, where

$$
\begin{aligned}
V &= \{S,\ a,\ b,\ c,\ A,\ B,\ C,\ T_a,\ T_b,\ T_c\},\\
\Sigma &= \{a,\ b,\ c\}, \text{and}\\
R &= \{S \to ABCS,\\
&\qquad S \to T_c,\\
&\qquad CA \to AC,\\
&\qquad BA \to AB,\\
&\qquad CB \to BC,\\
&\qquad CT_c \to T_c c,\\
&\qquad CT_c \to T_b C,\\
&\qquad BT_b \to T_b b,\\
&\qquad Bn \to T_a b,\\
&\qquad AT_a \to T_a a,\\
&\qquad T_a \to e\}
\end{aligned}
$$

- The first three rules generate a string of the form $(ABC)^n T_c$.
- Then the next three rules allow the A's, B's, and C's in the string to "sort out" themselves correctly, so that the string becomes $A^n B^n C^n T_c$.
- Finally, the remaining rules allow the T_c to "migrate" to the left, transforming all C's to c's, and then becoming T_b. In turn, T_b migrates to the left, transforming all B's into b's and becoming T_a, and finally T_a transforms all A's into a's and then is erased.

Definition 5.2

Let $G = (V, \Sigma, R, S)$ be a grammar, and let $f : \Sigma^* \mapsto \Sigma^*$ be a function. G **computes** f if for all $w, v \in \Sigma^*$, the following is true,

$$SwS \Rightarrow_G^* v \text{ if and only if } v = f(w)$$

That is, the string consisting of the input w, with a starting symbol of G on each side, yields exactly one string in Σ^*: the correct value of $f(w)$.

A function $f : \Sigma^* \mapsto \Sigma^*$ is **grammatically computable** if and only if there is a grammar G that computes it.

Theorem 5.2

A function $f : \Sigma^* \mapsto \Sigma^*$ is recursive if and only if it is grammatically computable.