4 Turing Machines

4.1 THE DEFINITION OF A TURING MACHINE

We have seen in the last two chapters that neither finite automata nor pushdown automata can be regarded as truly general models for computers, since they are not capable of recognizing even such simple languages as $\{a^n b^n c^n : n \geq 0\}$. In this chapter we take up the study of devices that can recognize this and many more complicated languages. Although these devices, called **Turing machines** after their inventor Alan Turing (1912–1954), are more general than the automata previously studied, their basic appearance is similar to those automata. A Turing machine consists of a finite control, a tape, and a head that can be used for reading *or writing* on that tape. The formal definitions of Turing machines and their operation are in the same mathematical style as those used for finite and pushdown automata. So in order to gain the additional computational power and generality of function that Turing machines possess, we shall not move to an entirely new sort of model for a computer.

Nevertheless, Turing machines are not simply one more class of automata, to be replaced later on by a yet more powerful type. We shall see in this chapter that, as primitive as Turing machines seem to be, attempts to strengthen them do not have any effect. For example, we also study Turing machines with many tapes, or machines with fancier memory devices that can be read or written in a *random access* mode reminiscent of actual computers; significantly, these devices turn out to be no stronger in terms of computing power than basic Turing machines. We show results of this kind by simulation methods: We can convert any "augmented" machine into a standard Turing machine which functions in an analogous way. Thus any computation that can be carried out on the fancier type of machine can actually be carried out on a Turing machine of the standard variety. Furthermore, towards the end of this chapter we define a certain kind of

language generator, a substantial generalization of context-free grammars, which is also shown to be equivalent to the Turing machine. From a totally different perspective, we also pursue the question of when to regard a numerical function (such as $2^x + x^2$) as computable, and end up with a notion that is once more equivalent to Turing machines!

So the Turing machines seem to form a *stable* and *maximal* class of computational devices, in terms of the computations they can perform. In fact, in the next chapter we shall advance the widely accepted view that any reasonable way of formalizing the idea of an "algorithm" must be ultimately equivalent to the idea of a Turing machine.

But this is getting ahead of our story. 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.

Now let us look more closely at these machines. In essence, a Turing machine consists of a finite-state control unit and a tape (see Figure 4-1). 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 $\triangleright$

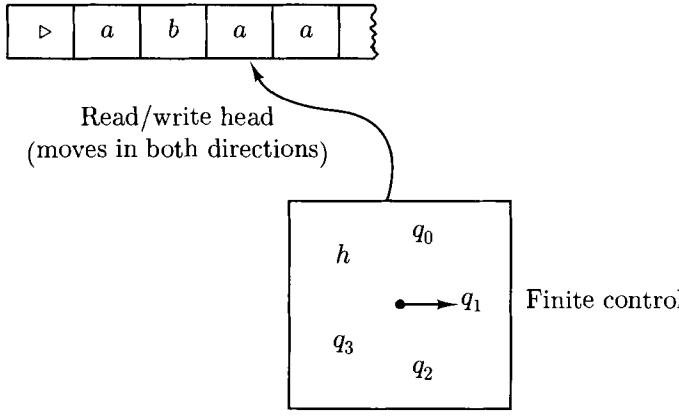


Figure 4-1

symbol. The rest of the tape initially contains **blank** symbols, denoted ⊔. 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 4.1.1: 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** ⊔ and the **left end symbol** ▷, but not containing the symbols ← and →;

$s \in K$ 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 ← or →, 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 the requirement (a) on δ: When it sees the left end of the tape ▷, it *must* move right. This way the leftmost ▷ is never erased, and M never falls off the left end of its tape. By (b), M never writes a ▷, and therefore ▷ 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 4.1.1: Consider the Turing machine $M = (K, \Sigma, \delta, s, \{h\})$, where

$$\begin{aligned} K &= \{q_0, q_1, h\}, \\ \Sigma &= \{a, \sqcup, \triangleright\}, \\ s &= q_0, \end{aligned}$$

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 $(K - H) \times \Sigma$.$\diamondsuit$

Example 4.1.2: Consider the Turing machine $M = (K, \Sigma, \delta, s, H)$, where

$$\begin{aligned} K &= \{q_0, h\}, \\ \Sigma &= \{a, \sqcup, \triangleright\}, \\ s &= q_0, \\ H &= \{h\}, \end{aligned}$$

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

and δ is given by this table.

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.*$\diamondsuit$

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 4.1.2: 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 4-2), 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**.

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 4-2, 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

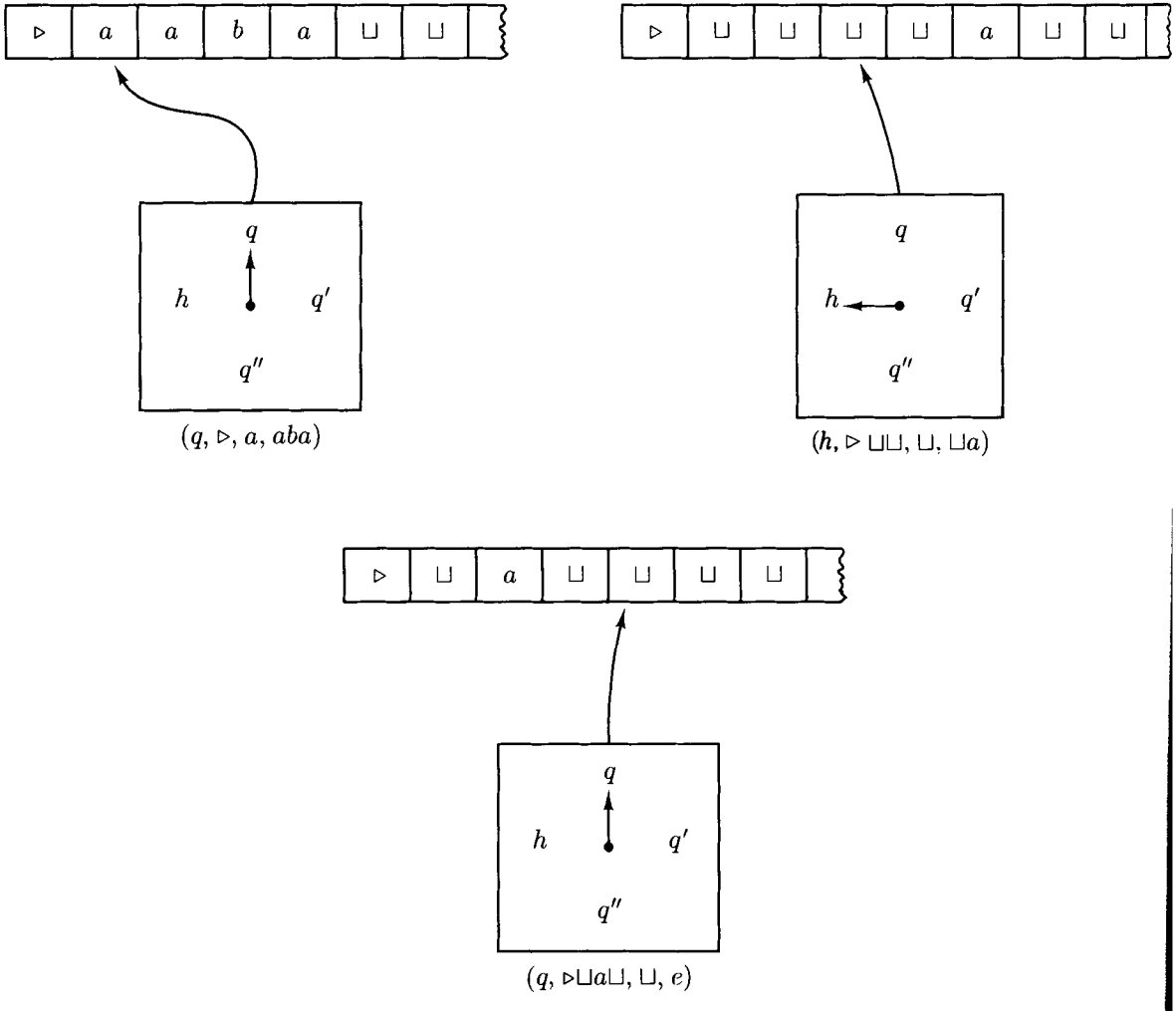


Figure 4-2

write the three configurations shown in Figure 4-2 as $(q, \triangleright \underline{a} aba)$, $(h, \triangleright \sqcup \sqcup \underline{\sqcup} \sqcup a)$, and $(q, \triangleright \sqcup a \sqcup \underline{\sqcup})$.

Definition 4.1.3: 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. $b = \leftarrow$, $w_1 = w_2 a_2$, and either
   (a) $u_2 = a_1 u_1$, if $a_1 \neq \sqcup$ or $u_1 \neq e$, or
   (b) $u_2 = e$, if $a_1 = \sqcup$ and $u_1 = e$; or
3. $b = \rightarrow$, $w_2 = w_1 a_1$, and either
   (a) $u_1 = a_2 u_2$, 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 4.1.3: 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): $(q_1, w\underline{a}) \vdash_M (q_2, wa\underline{\sqcup})$. $\diamondsuit$

Definition 4.1.4: 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 $C_0 \vdash_M^n C_n$.

Example 4.1.4: Consider the Turing machine M described in Example 4.1.1. If M is started in configuration $(q_1, \triangleright\sqcup aaaa)$, its computation would be represented

formally as follows.

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

The computation has ten steps.$\diamondsuit$

A Notation for Turing Machines

The Turing machines we have seen so far are extremely simple —at least when compared to our stated ambitions in this chapter— and their tabular form is already quite complex and hard to interpret. Obviously, we need a notation for Turing machines that is more graphic and transparent. For finite automata, we used in Chapter 2 a notation that involved states and arrows denoting transitions. We shall next adopt a similar notation for Turing machines. However, the things joined by arrows will in this case be *themselves* Turing machines. In other words, 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. So if M_1, M_2, and M_3 are Turing machines, the machine displayed in Figure 4-3 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.*

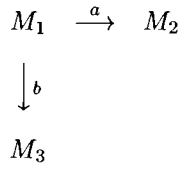


Figure 4-3

It is straightforward to give an explicit definition of the combined Turing machine from its constituents. Let us take the machine shown in Figure 4-3 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 4-3 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.

All the ingredients of our notation are now in place. We shall be building machines by combining the basic machines, and then we shall further combine the combined machines to obtain more complex machines, and so on. We know that, if we wished, we could come up with a quintuple form of every machine we thus describe, by starting from the quintuples of the basic machines and carrying out the explicit construction exemplified above.

Example 4.1.5: Figure 4-4(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.

(a) (b)

Figure 4-4

It will be convenient to represent this machine as in Figure 4-4(b). That is, an arrow labeled with several symbols is the same as several parallel arrows, one for each symbol. If an arrow is labeled by *all* symbols in the alphabet Σ of the machines, then the labels can be omitted. Thus, if we know that $\Sigma = \{a, b, \triangleright, \sqcup\}$, then we can display the machine above as

$$R \to R,$$

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.$\diamondsuit$

Example 4.1.6: 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 4-5(a) scans its tape to the right until it finds a blank. We shall denote this most useful machine by $R_{\sqcup}$.

(a) (b)

Figure 4-5

Another shorthand version of the same machine as in Figure 4-5(a) is shown in Figure 4-5(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 4-6 depicts a machine that scans to the right

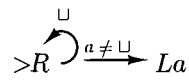


Figure 4-6

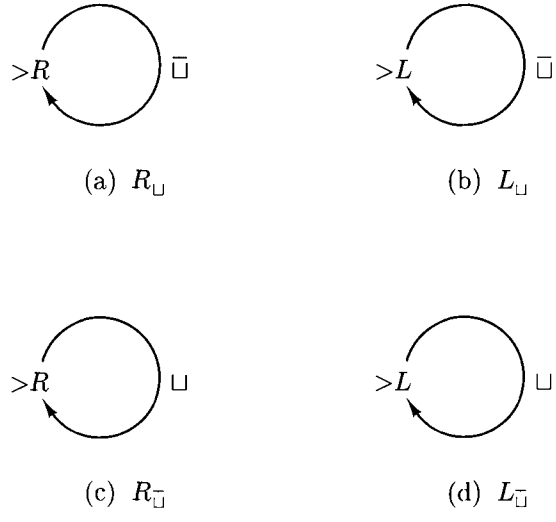


Figure 4-7

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.◇

Example 4.1.7: Machines to find marked or unmarked squares are illustrated in Figure 4-7. They are the following.

(a) $R_{\sqcup}$, which finds the first blank square to the right of the currently scanned square.
(b) $L_{\sqcup}$, which finds the first blank square to the left of the currently scanned square.
(c) $R_{\overline{\sqcup}}$, which finds the first *non*blank square to the right of the currently scanned square.
(d) $L_{\overline{\sqcup}}$, which finds the first nonblank square to the left of the currently scanned square.◇

Example 4.1.8: The *copying machine* C performs the following function: If C starts with input w, that is, if string w, containing only nonblank symbols but possibly empty, is put on an otherwise blank tape with one blank square to its

left, and the head is put on the blank square to the left of w, then the machine will eventually stop with $w \sqcup w$ on an otherwise blank tape. We say that C **transforms** $\sqcup w \underline{\sqcup}$ **into** $\sqcup w \sqcup w \underline{\sqcup}$.

A diagram for C is given in Figure 4-8.$\diamondsuit$

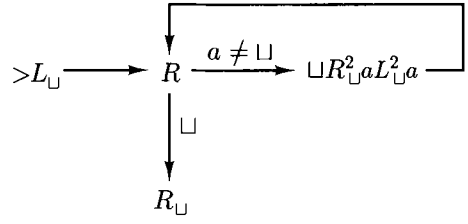


Figure 4-8

Example 4.1.9: The right-shifting machine $S_{\rightarrow}$, transforms $\sqcup w \underline{\sqcup}$, where w contains no blanks, into $\sqcup\sqcup\ w \underline{\sqcup}$. It is illustrated in Figure 4-9.$\diamondsuit$

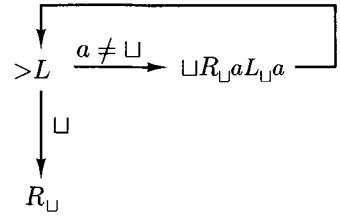


Figure 4-9

Example 4.1.10: Figure 4-10 is the machine defined in Example 4.1.1, which erases the a's in its tape.

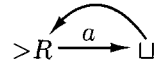


Figure 4-10

As a matter of fact, the fully developed transition table of this machine will differ from that of the machine given in Example 4.1.1 in ways that are subtle, inconsequential, and explored in Problem 4.1.8 —namely, the machine in Figure 4-10 will also contain certain extra states, which are final states of its constituents machines.$\diamondsuit$

Problems for Section 4.1

4.1.1. Let $M = (K, \Sigma, \delta, s, \{h\})$, where

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

and δ is given by the following table.

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

(a) Trace the computation of M starting from the configuration $(q_0, \triangleright \underline{a} abbba)$.
(b) Describe informally what M does when started in q_0 on any square of a tape.

4.1.2. Repeat Problem 4.1.1 for the machine $M = (K, \Sigma, \delta, s, \{h\})$, where

$$K = \{q_0, q_1, q_2, h\},$$
$$\Sigma = \{a, b, \sqcup, \triangleright\},$$
$$s = q_0,$$

and δ is given by the following table (the transitions on $\triangleright$ are $\delta(q, \triangleright) = (q, \triangleright)$, and are omitted).

q,	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, \leftarrow)$
q_0	b	$(q_0, \rightarrow)$
q_0	$\sqcup$	$(q_0, \rightarrow)$
q_1	a	$(q_1, \leftarrow)$
q_1	b	$(q_2, \rightarrow)$
q_1	$\sqcup$	$(q_1, \leftarrow)$
q_2	a	$(q_2, \rightarrow)$
q_2	b	$(q_2, \rightarrow)$
q_2	$\sqcup$	$(h, \sqcup)$

Start from the configuration $(q_0, \triangleright a\underline{b}b \sqcup bb \sqcup \sqcup \sqcup aba)$.

4.1.3. Repeat Problem 4.1.1 for the machine $M = (K, \Sigma, \delta, s, \{h\})$, where

$$\begin{aligned} K &= \{q_0, q_1, q_2, q_3, q_4, h\}, \\ \Sigma &= \{a, b, \sqcup, \triangleright\}, \\ s &= q_0, \end{aligned}$$

and δ is given by the following table.

q,	σ	$\delta(q, \sigma)$
q_0	a	$(q_2, \rightarrow)$
q_0	b	(q_3, a)
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	$(q_2, \rightarrow)$
q_1	b	$(q_2, \rightarrow)$
q_1	$\sqcup$	$(q_2, \rightarrow)$
q_1	$\triangleright$	$(q_1, \rightarrow)$
q_2	a	(q_1, b)
q_2	b	(q_3, a)
q_2	$\sqcup$	$(h, \sqcup)$
q_2	$\triangleright$	$(q_2, \rightarrow)$
q_3	a	$(q_4, \rightarrow)$
q_3	b	$(q_4, \rightarrow)$
q_3	$\sqcup$	$(q_4, \rightarrow)$
q_3	$\triangleright$	$(q_3, \rightarrow)$
q_4	a	$(q_2, \rightarrow)$
q_4	b	$(q_4, \rightarrow)$
q_4	$\sqcup$	$(h, \sqcup)$
q_4	$\triangleright$	$(q_4, \rightarrow)$

Start from the configuration $(q_0, \triangleright \underline{a}aabbbaa)$.

4.1.4. Let M be the Turing machine $(K, \Sigma, \delta, s, \{h\})$, where

$$\begin{aligned} K &= \{q_0, q_1, q_2, h\}, \\ \Sigma &= \{a, \sqcup, \triangleright\}, \\ s &= q_0, \end{aligned}$$

and δ is given by the following table.

Let $n \geq 0$. Describe carefully what M does when started in the configuration $(q_0, \triangleright \sqcup a^n \underline{a})$.

q,	σ	$\delta(q,\sigma)$
q_0	a	$(q_1, \leftarrow)$
q_0	$\sqcup$	$(q_0, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	$(q_2, \sqcup)$
q_1	$\sqcup$	$(h, \sqcup)$
q_1	$\triangleright$	$(q_1, \rightarrow)$
q_2	a	(q_2, a)
q_2	$\sqcup$	$(q_0, \leftarrow)$
q_2	$\triangleright$	$(q_2, \rightarrow)$

4.1.5. In the definition of a Turing machine, we allow rewriting a tape square without moving the head and moving the head left or right without rewriting the tape square. What would happen if we also allowed to leave the head stationary without rewriting the tape square?

4.1.6. (a) Which of the following could be configurations?

(i) $(q, \triangleright a \sqcup a\sqcup, \sqcup, \sqcup a)$
(ii) (q, abc, b, abc)
(iii) $(p, \triangleright abc, a, e)$
(iv) $(h, \triangleright, e, e)$
(v) $(q, \triangleright a \sqcup ab, b, \sqcup aa\sqcup)$
(vi) $(p, \triangleright a, ab, \sqcup a)$
(vii) $(q, \triangleright, e, \sqcup aa)$
(viii) $(h, \triangleright a, a, \sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup a)$

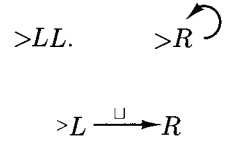
(b) Rewrite those of Parts (i) through (viii) that are configurations using the abbreviated notation.

(c) Rewrite these abbreviated configurations in full.

(i) $(q, \triangleright a\underline{b}cd)$
(ii) $(q, \triangleright\underline{a})$
(iii) $(p, \triangleright aa \sqcup \underline{\sqcup})$
(iv) $(h, \triangleright\underline{\sqcup}abc)$

4.1.7. Design and write out in full a Turing machine that scans to the right until it finds two consecutive a's and then halts. The alphabet of the Turing machine should be $\{a, b, \sqcup, \triangleright\}$.

4.1.8. Give the full details of the Turing machines illustrated.



4.1.9. Do the machines LR and RL always accomplish the same thing? Explain.

4.1.10. Explain what this machine does.

$$>R \xrightarrow{a \neq \sqcup} R \xrightarrow{b \neq \sqcup} R_{\sqcup} a R_{\sqcup} \mathrm{b}$$

4.1.11. Trace the operation of the Turing machine of Example 4.1.8 when started on $\triangleright \sqcup aabb$.

4.1.12. Trace the operation of the Turing machine of Example 4.1.9 on $\triangleright \sqcup aabb \sqcup$.

4.2 COMPUTING WITH TURING MACHINES

We introduced Turing machines with the promise that they outperform, as language acceptors, all other kinds of automata we introduced in previous chapters. So far, however, we have presented only the "mechanics" of Turing machines, without any indication of how they are to be used in order to perform computational tasks —to recognize languages, for example. It is as though a computer had been delivered to you without a keyboard, disk drive, or screen —that is, without the means for getting information into and out of it. It is time, therefore, to fix some conventions for the use of Turing machines.

First, we adopt the following policy for presenting input to Turing machines: The input string, with 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; and 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 4.2.1: 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.

Example 4.2.1: Consider the language $L = \{a^n b^n c^n : n \geq 0\}$, which has heretofore evaded all types of language recognizers. The Turing machine whose diagram is shown in Figure 4-11 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.

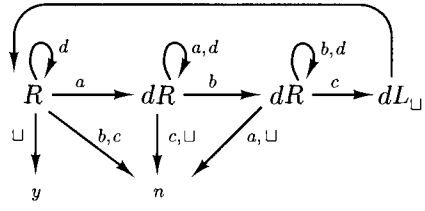


Figure 4-11

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.$\diamondsuit$

There is a subtle point in relation to Turing machines that decide languages: With the other language recognizers that we have seen so far in this book (even the nondeterministic ones), one of two things could happen: either the machine accepts the input, or it rejects it. 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.

Recursive Functions

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

Definition 4.2.2: 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 4.2.2: 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 (both were defined towards the end of the last section).$\diamondsuit$

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

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

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 4.2.3: 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 $w_1, \ldots, w_k \in 0 \cup 1\{0,1\}^*$ (that is, for any k strings that are binary encodings of integers), $\mathsf{num}(M(w_1; \ldots; w_k)) = f(\mathsf{num}(w_1), \ldots, \mathsf{num}(w_k))$. That is, if M is started with the binary representations of the integers $n_1, \ldots, n_k$ as input, then it eventually halts, and when it does halt, its tape contains a string that represents number $f(n_1, \ldots, n_k)$ —the value of the function. A function $f : \mathbf{N}^k \mapsto \mathbf{N}$ is called **recursive** if there is a Turing machine M that computes f.

In fact, the term *recursive* used to describe both functions and languages computed by Turing machines originates in the study of such numerical functions. It anticipates a result we shall prove towards the end of this chapter, namely that the numerical functions computable by Turing machines coincide with those that can be defined *recursively* from certain basic functions.

Example 4.2.3: We can design a machine that computes the *successor* function $\mathsf{succ}(n) = n + 1$ (Figure 4.12; S_R is the *right-shifting machine*, the rightward analog of the machine in Example 4.1.9). This machine first finds the right end of the input, and then goes to the left as long as it sees 1's, changing all of them to 0's. When it sees a 0, it changes it into a 1 and halts. If it sees a ⊔ while looking for a 0, this means that the input number has a binary representation that is all 1's (it is a power of two minus one), and so the machine again writes a 1 in the place of the ⊔ and halts, after shifting the whole string one position to the right. Strictly speaking, the machine shown does not compute $n+1$ because it fails to always halt with its head to the left of the result; but this can be fixed by adding a copy of $R_⊔$ (Figure 4-5).◇

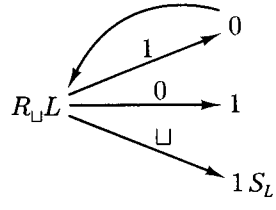


Figure 4-12

The last remark of the previous subsection, 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.

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 4.2.4: 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. Since any configuration that is not halting yields some other configuration (δ is a fully defined function), the machine must in this case 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 4.2.4: Let $L = \{w \in \{a, b\}^* : w \text{ contains at least one } a\}$. Then L is semidecided by the Turing machine shown in Figure 4-13.

Figure 4-13

This machine, when started in configuration $(q0, \triangleright\sqcup w)$ 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.$\diamondsuit$

"Going on forever into the blanks" is only one of the ways in which a Turing machine may fail to halt. For example, any machine with $\delta(q, a) = (q, a)$ will "loop forever" in place if it ever encounters an a in state q. Naturally, more complex looping behaviors can be designed, with the machine going indefinitely through a finite number of different configurations.

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 nonfinal 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.

We know from Example 4.2.1 that $\{a^n b^n c^n : n \geq 0\}$ is a recursive language. But is it recursively enumerable? The answer is easy: *Any recursive language is also recursively enumerable.* 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: $M = (K, \Sigma, \delta', s, \{y\})$, 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 $M(w) = n$.

We have proved the following important result:

Theorem 4.2.1: *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 (with our one-sided definition of semidecision that makes it virtually useless as a computational device) 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 4.2.2: *If L is a recursive language, then its complement $\overline{L}$ is also*

---

† We have already encountered the same difficulty with pushdown automata (recall Section 3.7). A pushdown automaton can in principle reject an input by manipulating forever its stack without reading any further input —in Section 3.7 we had to remove such behavior in order to obtain computationally useful pushdown automata for certain context-free languages.

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 rôles 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 $\overline{L}$. ■

Is the class of recursively enumerable languages also closed under complement? Again, we shall see in the next chapter that *the answer is negative.*

Problems for Section 4.2

4.2.1. Give a Turing machine (in our abbreviated notation) that computes the following function from strings in $\{a, b\}^*$ to strings in $\{a, b\}^*$: $f(w) = ww^R$.

4.2.2. Present Turing machines that decide the following languages over $\{a, b\}$:
(a) $\emptyset$
(b) $\{e\}$
(c) $\{a\}$
(d) $\{a\}^*$

4.2.3. Give a Turing machine that semidecides the language a^*ba^*b.

4.2.4. (a) Give an example of a Turing machine with one halting state that does not compute a function from strings to strings.
(b) Give an example of a Turing machine with two halting states, y and n, that does not decide a language.
(c) Can you give an example of a Turing machine with one halting state that does not *semidecide* a language?

4.3 EXTENSIONS OF THE TURING MACHINE

The examples of the previous section make it clear that Turing machines can perform fairly powerful computations, albeit slowly and clumsily. In order to better understand their surprising power, we shall consider the effect of extending the Turing machine model in various directions. We shall see that in each case

the additional features do not add to the classes of computable functions or decidable languages: the "new, improved models" of the Turing machine can in each instance be *simulated* by the standard model. Such results increase our confidence that the Turing machine is indeed the ultimate computational device, the end of our progression to more and more powerful automata. A side benefit of these results is that we shall feel free subsequently to use the additional features when designing Turing machines to solve particular problems, secure in the knowledge that our dependency on such features can, if necessary, be eliminated.

Multiple Tapes

One can think of Turing machines that have several tapes (see Figure 4-14). 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 $k = 1$.

Definition 4.3.1: 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 4.3.2: 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 $\delta(p, (a_1, \ldots, a_k)) = (b_1, \ldots, b_k)$, then in one move the machine

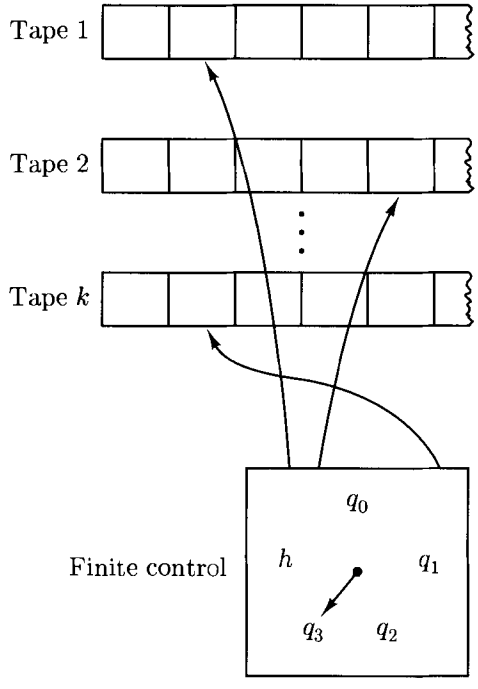


Figure 4-14

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 4.1.3. 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 $(p, (w_1'\underline{a_1'}u_1', \ldots, w_k'\underline{a_k'}u_k'))$.

Example 4.3.1: A k-tape Turing machine can be used for computing a function or deciding or semideciding 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, for example, the task of the copying machine C given in Example 4.1.8: to transform $\triangleright \sqcup w\underline{\sqcup}$ into $\triangleright \sqcup w \sqcup w\underline{\sqcup}$, 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 first tape is an a. And so on. (When representing multi-tape Turing machines, we refrain from using the shorthand M^2 for MM.) Using this convention, the 2-tape version of the copying machine might be illustrated as in Figure 4-15. We indicate the submachines performing Functions 1 through 3 above.$\diamondsuit$

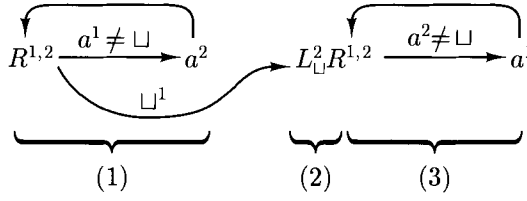


Figure 4-15

Example 4.3.2: We have seen (Example 4.2.3) that Turing machines can add 1 to any binary integer. It should come as no surprise that Turing machines

can also *add arbitrary binary numbers* (recall Problem 2.4.3, suggesting that even finite automata, in a certain sense, can). With two tapes this task can be accomplished by the machine depicted in Figure 4-16. Pairs of bits such as 01 on an arrow label are a shorthand for, in this case, $a^1 = 0, a^2 = 1$.

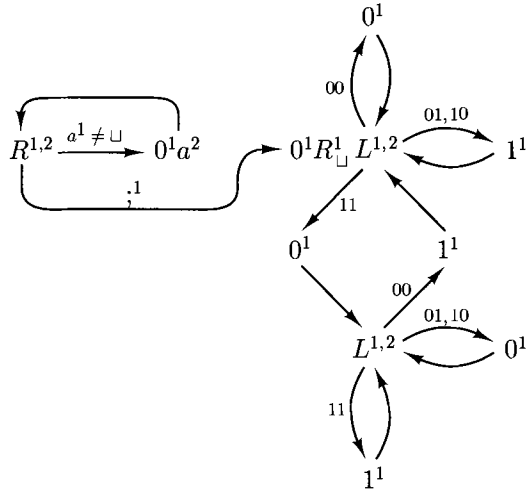


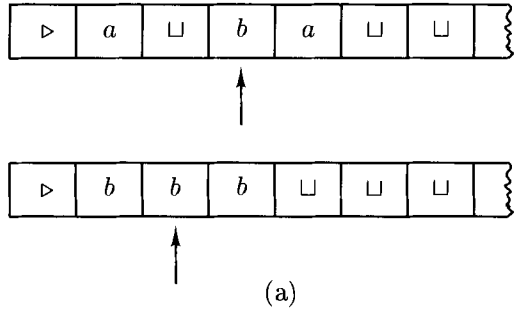
Figure 4-16

This machine first copies the first binary integer in its second tape, writing zeros in its place (and in the place of the ";" separating the two integers) in the first tape; this way the first tape contains the second integer, with zeros added in front. The machine then performs binary addition by the "school method," starting from the least significant bit of both integers, adding the corresponding bits, writing the result in the first tape, and "remembering the carry" in its state.◇

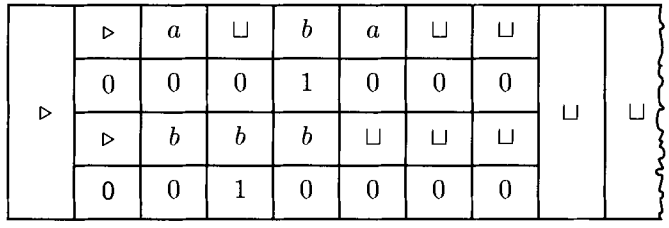
What is more, we can build a 3-tape Turing machine that *multiplies* two numbers; its design is left as an exercise (Problem 4.3.5).

Evidently, 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 4.3.1: *Let $M = (K, \Sigma, \delta, s, H)$ be a k-tape Turing machine for some $k \geq 1$. Then there is a standard Turing 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 $\mathcal{O}(t \cdot (|x| + t))$.*



(a)



(b)

Figure 4-17

Proof: The tape of M' must somehow contain all information in all tapes of M. A simple way of achieving this is by thinking that the tape of M' is divided into several *tracks* (see Figure 4-18(b)), with each "track" devoted to the simulation of a different tape of M. In particular, except for the leftmost square, which contains as usual the left end symbol ▷, and the infinite blank portion of the tape to the right, the single tape of M' is split horizontally into $2k$ tracks. The first, third, $\ldots$, $(2k-1)$st tracks of the tape of M' correspond to the first, second, $\ldots$, kth tapes of M. The second, fourth, $\ldots$, $2k$th tracks of the tape of

M' are used to record the positions of the heads on the first, second, ..., kth tapes of M in the following way: If the head on the ith tape of M is positioned over the nth tape square, then the $2i$th track of the tape of M' contains a 1 in the $(n+1)$st tape square and a 0 in all tape squares except the $(n+1)$st. For example, if $k = 2$, then the tapes and heads of M shown in Figure 4-18(a) would correspond to the tape of M' shown in Figure 4-18(b).

Of course, the division of the tape of M' into tracks is a purely conceptual device; formally, the effect is achieved by letting

$$\Sigma' = \Sigma \cup (\Sigma \times \{0,1\})^k.$$

That is, the alphabet of M' consists of the alphabet of M (this enables M' to receive the same inputs as M and deliver the same output), plus all $2k$-tuples of the form $(a_1, b_1, \ldots, a_k, b_k)$ with $a_1, \ldots, a_k \in \Sigma$ and $b_1, \ldots, b_k \in \{0,1\}$. The translation from this alphabet to the $2k$-track interpretation is simple: We read any such $2k$-tuple as saying that the first track of M' contains a_1, the second b_1, and so on up to the $2k$th track containing b_k. This in turn means that the corresponding symbol of the ith tape of M contains a_i, and that this symbol is scanned by the ith head if and only if $b_i = 1$ (recall Figure 4-17(b)).

When given an input $w \in \Sigma^*$, M' operates as follows.

(1) Shift the input one tape square to the right. Return to the square immediately to the right of the $\triangleright$, and write the symbol $(\triangleright, 0, \triangleright, 0, \ldots, \triangleright, 0)$ on it —this will represent the left end of the k tapes. Go one square to the right and write the symbol $(\sqcup, 1, \sqcup, 1, \ldots, \sqcup, 1)$ —this signifies that the first squares of all k tapes contain a $\sqcup$, and are all scanned by the heads. Proceed to the right. At each square, if a symbol $a \neq \sqcup$ is encountered, write in its position the symbol $(a, 0, \sqcup, 0, \ldots, \sqcup, 0)$. If a $\sqcup$ is encountered, the first phase is over. The tape contents of M' faithfully represent the initial configuration of M.

(2) Simulate the computation by M, until M would halt (*if* it would halt). To simulate one step of the computation of M, M' will have to perform the following sequence operations (we assume that it starts each step simulation with its head scanning the first "true blank," that is, the first square of its tape that has not yet been subdivided into tracks):

   (a) Scan left down the tape, gathering information about the symbols scanned by the k tape heads of M. After all scanned symbols have been identified (by the 1's in the corresponding even tracks), return to the leftmost true blank. No writing on the tape occurs during this part of the operation of M', but when the head has returned to the right end, the state of the finite control has changed to reflect the k-tuple of symbols from Σ, in the k tracks at the marked head positions.

   (b) Scan left and then right down the tape to update the tracks in accordance with the move of M that is to be simulated. On each pair of

tracks, this involves either moving the head position marker one square to the right or left, or rewriting the symbol from Σ.

(3) When M would halt, M' first converts its tape from tracks into single-symbol format, ignoring all tracks except for the first; it positions its head where M would have placed its first head, and finally it halts in the same state as M would have halted.

Many details have been omitted from this description. Phase 2, while by no means conceptually difficult, is rather messy to specify explicitly, and indeed there are several choices as to how the operations described might actually be carried out. One detail is perhaps worth describing. Occasionally, for some $n > |w|$, M may have to move one of its heads to the nth square of the corresponding tape *for the first time.* To simulate this, M' will have to extend the part of its tape that is divided into $2k$ tracks, and rewrite the first $\sqcup$ to the right as the $2k$-tuple $(\underline{\sqcup}, 0, \sqcup, 0, \ldots, \sqcup, 0) \in \Sigma'$.

It is clear that M' can simulate the behavior of M as indicated in the statement of the theorem. It remains to argue that the number of steps required by M' for simulating t steps of M on input x is $\mathcal{O}(t \cdot (|x| + t))$. Phase 1 of the simulation requires $\mathcal{O}(|x|)$ steps of M'. Then, for each step of M, M' must carry out the maneuver in Phase 2, (a) and (b). This requires M' to scan the $2k$-track part of its tape twice; that is, it requires a number of steps by M' that is proportional to the length of the $2k$-track part of the tape of M'. The question is, how long can this part of M''s tape be? It starts by being $|x| + 2$ long, and subsequently it increases in length by no more than one for each simulated step of M. Thus, if t steps of M are simulated on input x, the length of the $2k$-track part of the tape of M' is at most $|x| + 2 + t$, and hence each step of M can be simulated by $\mathcal{O}(|x| + t)$ steps of M', as was to be shown. ■

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.*

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, say. 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.

Multiple Heads

What if we allow a Turing 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 4.3.1.

The use of multiple heads, like multiple tapes, can sometimes drastically simplify the construction of a Turing machine. A 2-head version of the copying machine C in Example 4.1.8 could function in a way that is much more natural than the one-head version (or even the two-tape version, Example 4.3.1); see Problem 4.3.3.

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" (see the *tiling* problem in the next chapter). We leave it as an exercise (Problem 4.3.6) to define in detail the operation of such machines. 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 4.3.2: *Any language decided or semidecided, and any function computed by Turing machines with several tapes, heads, two-way infinite tapes, or multi-dimensional tapes, can be decided, semidecided, or computed, respectively, by a standard Turing machine.*

Problems for Section 4.3

4.3.1. Formally define:
(a) M semidecides L, where M is a two-way infinite tape Turing machine;
(b) M computes f, where M is a k-tape Turing machine and f is a function from strings to strings.

4.3.2. Formally define:
(a) a k-head Turing machine (with a single one-way infinite tape);
(b) a configuration of such a machine;
(c) the yields in one step relation between configurations of such a machine. (There is more than one correct set of definitions.)

4.3.3. Describe (in an extension of our notation for k-tape Turing machines) a 2-head Turing machine that compute the function $f(w) = ww$.

4.3.4. The stack of a pushdown automaton can be considered as a tape that can be written and erased only at the right end; in this sense a Turing machine is a generalization of the deterministic pushdown automaton. In this problem we consider a generalization in another direction, namely the *deterministic pushdown automaton with two stacks.*
(a) Define informally but carefully the operation of such a machine. Define what it means for such a machine to decide a language.
(b) Show that the class of languages decided by such machines is precisely the class of recursive languages.

4.3.5. Give a three-tape Turing machine which, when started with two binary integers separated by a ';' on its first tape, computes their product. (*Hint:* Use the adding machine of Example 4.3.2 as a "subroutine."

4.3.6. Formally define a Turing machine with a 2-dimensional tape, its configurations, and its computation. Define what it means for such a machine to decide a language L. Show that t steps of this machine, starting on an input of length n, can be simulated by a standard Turing machine in time that is *polynomial* in t and n.

4.4 RANDOM ACCESS TURING MACHINES

Despite the apparent power and versatility of the variants of the Turing machines we have discussed so far, they all have a quite limiting common feature: Their memory is *sequential;* that is, in order to access the information stored at some location, the machine must first access, one by one, all locations between the current and the desired one. In contrast, real computers have *random access memories*, each element of which can be accessed in a single step, if appropriately addressed. What would happen if we equipped our machines with such a random access capability, enabling them to access any desired tape square in a single step? To attain such a capability, we must also equip our machines with *registers*, capable of storing and manipulating the *addresses* of tape squares. In this subsection we define such an extension of the Turing machine; significantly, we see that it too is equivalent in power to the standard Turing machine, with only a polynomial loss in efficiency.

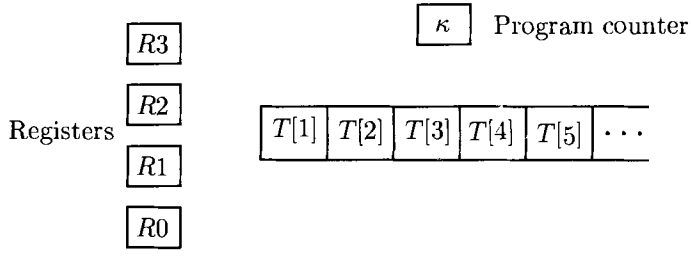


Figure 4-18

A *random access Turing machine* has a fixed number of *registers* and a one-way infinite tape (see Figure 4-18; we continue to call the machine's memory a "tape" for compatibility and comparison with the standard model, despite the fact that, as we shall see, it behaves much more like a random access memory chip). Each register and each tape square is capable of containing an arbitrary natural number. The machine acts on its tape squares and its registers as dictated by a fixed *program* —the analog of the transition function of ordinary Turing machines. The program of a random access Turing machine is a sequence of *instructions*, of a kind reminiscent of the instruction set of actual computers. The kinds of instructions allowed are enumerated in Figure 4-19.

Initially the register values are 0, the program counter is 1, and the tape contents encode the input string in a simple manner that will be specified shortly. Then the machine executes the first instruction of its program. This will change the contents of the registers or of the tape contents as indicated in Figure 4-19;

Instruction	Operand	Semantics
read	j	$R_0 := T[R_j]$
write	j	$T[R_j] := R_0$
store	j	$R_j := R_0$
load	j	$R_0 := R_j$
load	$= c$	$R_0 := c$
add	j	$R_0 := R_0 + R_j$
add	$= c$	$R_0 := R_0 + c$
sub	j	$R_0 := \max\{R_0 - R_j, 0\}$
sub	$= c$	$R_0 := \max\{R_0 - c, 0\}$
half		$R_0 := \lfloor \frac{R_0}{2} \rfloor$
jump	s	$\kappa := s$
jpos	s	if $R_0 > 0$ then $\kappa := s$
jzero	s	if $R_0 = 0$ then $\kappa := s$
halt		$\kappa := 0$

Notes: j stands for a register number, $0 \leq j < k$. $T[i]$ denotes the current contents of tape square i. R_j denotes the current contents of Register j. $s \leq p$ denotes any instruction number in the program. c is any natural number. All instructions change κ to $\kappa + 1$, unless explicitly stated otherwise.

Figure 4-19

also, the value of the *program counter* κ, an integer identifying the instruction to be executed next, will be computed as indicated in the figure. Notice the special role of Register 0: it is the *accumulator*, where all arithmetic and logical computation takes place. The κth instruction of the program will be executed next, and so on, until a **halt** instruction is executed —at this point the operation of the random access Turing machine ends.

We are now ready to define formally a random access Turing machine, its configurations, and its computation.

Definition 4.4.1: A **random access Turing machine** is a pair $M = (k, \Pi)$, where $k > 0$ is the number of **registers**, and $\Pi = (\pi_1, \pi_2, \ldots, \pi_p)$, the **program**, is a finite sequence of **instructions**, where each instruction π_i is of one of the types shown in Figure 4-19. We assume that the last instruction, π_p, is always a **halt** instruction (the program may contain other **halt** instructions as well).

A **configuration** of a random access Turing machine (k, Π) is a $k+2$-tuple $(\kappa, R_0, R_1, \ldots, R_{k-1}, T)$, where

$\kappa \in \mathbf{N}$ is the **program counter**, an integer between 0 and p. The configuration is a **halted configuration** if κ is zero.

For each j, $0 \leq j < k$, $R_j \in \mathbf{N}$ is the **current value of Register** j.

T, the **tape contents**, is a finite set of pairs of positive integers —that is,

a finite subset of $(\mathbf{N} - \{0\}) \times (\mathbf{N} - \{0\})$— such that for all $i \geq 1$ there is at most one pair of the form $(i, m) \in T$.

Intuitively, $(i, m) \in T$ means that the ith tape square currently contains the integer $m > 0$. All tape squares not appearing as first components of a pair in T are assumed to contain 0.

Definition 4.4.1 (continued): Let $M = (k, \Pi)$ be a random access machine. We say that configuration $C = (\kappa, R_0, R_1, \ldots, R_{k-1}, T)$ of M **yields in one step** configuration $C' = (\kappa', R'_0, R'_1, \ldots, R'_{k-1}, T')$, denoted $C \vdash_M C'$, if, intuitively, the values of κ', the R'_j's and T' correctly reflect the application to κ, the R_j's, and T of the "semantics" (as in Figure 4-19) of the current instruction π_κ. We shall indicate the precise definition for a only a few of the fourteen kinds of instructions in Figure 4-19.

If π_κ is of the form read j, where $j < k$, then the execution of this instruction has the following effect: The value contained in Register 0 becomes equal to the value stored in tape square number R_j —the tape square "addressed" by Register j. That is, $R'_0 = T[R_j]$, where $T[R_j]$ is the unique value m such that $(R_j, m) \in T$, if such an m exists, and 0 otherwise. Also, $\kappa' = \kappa + 1$. All other components of the configuration C' are identical to those of C.

If π_κ is of the form add $= c$, where $c \geq 0$ is a fixed integer such as 5, then we have $R'_0 = R_0 + c$, and $\kappa' = \kappa + 1$, with all other components remaining the same.

If π_κ is of the form write j, where $j < k$, then we have $\kappa' = \kappa + 1$, T' is T with any pair of the form (R_j, m), if one exists, deleted, and, if $R_0 > 0$, the pair (R_j, R_0) added; all other components remain the same.

If π_κ is of the form jpos s, where $1 \leq s \leq p$, then we have $\kappa' = s$ if $R_0 > 0$, and $\kappa' = c + 1$ otherwise; all other components remain the same.

Similarly for the other kinds of instructions. The relation **yields**, $\vdash^*_M$, is the reflexive transitive closure of $\vdash_M$.

Example 4.4.1: The instruction set of our random access Turing machine (recall Figure 4-19) has no *multiplication* instruction mply. As it happens, if we allowed this instruction as a primitive, our random access Turing machine, although still equivalent to the standard Turing machine, would be much more time-consuming to simulate (see Problem 4.4.4).

The omission of the multiplication instruction is no great loss, however, because this instruction can be emulated by the program shown in Figure 4-20. If† Register 0 initially contains a natural number x and Register 1 initially

---

† The computation of a random access Turing machine starts with all registers 0.

contains y, then this random access Turing machine will halt, and Register 0 will contain the product $x \cdot y$. Multiplication is done by successive additions, where the instruction **half** is used to reveal the binary representation of y (actually, our instruction set contains this unusual instruction precisely for this use).

```
1.    store 2
2.    load 1
3.    jzero 19
4.    half
5.    store 3
6.    load 1
7.    sub 3
8.    sub 3
9.    jzero 13
10.   load 4
11.   add 2
12.   store 4
13.   load 2
14.   add 2
15.   store 2
16.   load 3
17.   store 1
19.   load 4
18.   jump 2
19.   load 4
20.   halt
```

Figure 4-20

Here is a typical sequence of configurations (since this machine does not interact with tape squares, the T part of these configurations is empty; there are $k = 5$ registers):

$(1;5,3,0,0,0;\emptyset) \vdash (2;5,3,5,0,0;\emptyset) \vdash (3;3,3,5,0,0;\emptyset) \vdash (4;3,3,5,0,0;\emptyset) \vdash$
$(5;1,3,5,0,0;\emptyset) \vdash (6;1,3,5,1,0;\emptyset) \vdash (7;3,3,5,1,0;\emptyset) \vdash (8;2,3,5,1,0;\emptyset) \vdash$
$(9;1,3,5,1,0;\emptyset) \vdash (10;1,3,5,1,0;\emptyset) \vdash (11;0,3,5,1,0;\emptyset) \vdash$
$(12;5,3,5,1,0;\emptyset) \vdash (13;5,3,5,1,5;\emptyset) \vdash (14;5,3,5,1,5;\emptyset) \vdash$
$(15;10,3,5,1,5;\emptyset) \vdash (16;10,3,10,1,5;\emptyset) \vdash (17;1,3,10,1,5;\emptyset) \vdash$
$(18;1,1,10,1,5;\emptyset) \vdash (2;1,1,10,1,5;\emptyset) \vdash^* (18;0,0,20,0,15;\emptyset) \vdash$
$(2;0,0,20,0,15;\emptyset) \vdash (3;0,0,20,0,15;\emptyset) \vdash (19;0,0,20,0,15;\emptyset) \vdash$
$(20;15,0,20,0,15;\emptyset)$]

However, since the present program is intended to be used as a part of other random access Turing machines, it makes sense to explore what would happen if it were started at an arbitrary configuration.

Let x and y be the nonnegative integers stored in Registers 0 and 1, respectively, at the beginning of the execution of this program. We claim that the machine eventually halts with the product $x \cdot y$ stored in Register 0 —as if it had executed the instruction "mply 1." The program proceeds in several *iterations*. An iteration is an execution of the sequence of instructions π_2 through π_{18}. At the kth iteration, $k \geq 1$, the following conditions hold:

(a) Register 2 contains $x2^k$,
(b) Register 3 contains $\lfloor y/2^k \rfloor$,
(c) Register 1 contains $\lfloor y/2^{k-1} \rfloor$,
(d) Register 4 contains the "partial result" $x \cdot (y \bmod 2^k)$.

The iteration seeks to maintain these "invariants." So, instructions π_2 through π_5 enforce Invariant (b), assuming (c) held in the previous iteration. Instructions π_6 through π_8 compute the kth least significant bit of y, and, if this bit is *not* zero, instructions π_9 through π_{12} add $x2^{k-1}$ to Register 4, as mandated by Invariant (d). Then Register 2 is doubled by instructions π_{13} through π_{15}, enforcing Invariant (a), and finally Register 3 is transferred to Register 1, enforcing (c). The iteration is then repeated. If at some point it is seen that $\lfloor y/2^{k-1} \rfloor = 0$, then the process terminates, and the final result is loaded from Register 4 to the accumulator.

We can abbreviate this program as "mply 1," that is, an instruction with semantics $R_0 := R_0 \cdot R_1$. We shall therefore feel free to use the instruction "mply j" or "mply = c" in our programs, knowing that we can simulate them by the above program. Naturally, the instruction numbers would be different, reflecting the program of which this mply instruction is a part. If a random access Turing machine uses this instruction, then it is implicitly assumed that, in addition to its registers explicitly mentioned, it must have three more registers that play the role of Registers 2, 3, and 4 in the above program.$\Diamond$

In fact, we can avoid the cumbersome appearance of random access Turing machine programs such as the one in the previous example by adopting some useful abbreviations. For example, denoting the value stored in Register 1 by R_1, in Register 2 by R_2, and so on, we can write

$$R_1 := R_2 + R_1 - 1$$

as an abbreviation of the sequence

```
1.    load 1
2.    add 2
3.    sub =1
4.    store 1
```

Once we adopt this, we could use better-looking names for the quantities stored at Registers 1 and 2, and express this sequence of instructions simply as

$$x := y + x - 1.$$

Here x and y are just names for the contents of Registers 1 and 2. We can even use abbreviations like

$$\text{while } x > 0 \text{ do } x := x - 3,$$

where x denotes the value of Register 1, instead of the sequence

1. load 1
2. jzero 6
3. sub =3
4. store 1
5. jump 1

Example 4.4.1 (continued): Here is a much more readable abbreviation of the mply program in Figure 4-20, where we are assuming that x and y are to be multiplied, and the result is w:

$w := 0$
while $y > 0$ do
  begin
  $z :=$ half(y)
  if $y - z - z \neq 0$ then $w := w + x$
  $x := x + x$
  $y := z$
  end
halt

The correspondence between the form above and the original program in Figure 4-20 is this: y stands for R_1, x for R_2, z for R_3, and w for R_4. Notice that we have also omitted for clarity the explicit instruction numbers; if goto instructions were necessary, we could label instructions by symbolic instruction labels like a and b wherever necessary.

Naturally, it is quite mechanical from an abbreviated program such as the above to arrive to an equivalent full-fledged random access Turing machine program such as the original one.$\Diamond$

Although we have explained the mechanics of random access Turing machines, we have not said how they receive their input and return their output. In order to facilitate comparisons with the standard Turing machine model, we shall assume that the input-output conventions of random access Turing machines are very much in the spirit of the input-output conventions for ordinary

Turing machines: The input is presented as a sequence of symbols in the tape. That is, although the tape of a random access Turing machine may contain arbitrary natural numbers, we assume that initially these numbers encode the symbols of some input string.

Definition 4.4.2: Let us fix an alphabet Σ —the alphabet from which our random access Turing machines will obtain their input— with $\sqcup \in \Sigma$ and $\triangleright \notin \Sigma$ ($\triangleright$ is not needed here, since a random access Turing machine is in no danger of falling off the left end of its tape). Also let $\mathbf{E}$ be a fixed bijection between Σ and $\{0, 1, \ldots, |\Sigma| - 1|\}$; this is how we encode the input and the output of random access Turing machines. We assume that $\mathbf{E}(\sqcup) = 0$. The **initial configuration** of a random access Turing machine $M = (k, \Pi)$ with input $w = a_1 a_2 \cdots a_n \in (\Sigma - \{\sqcup\})^*$ is $(\kappa, R_0, \ldots, R_{k-1}, T)$, where $\kappa = 1$, $R_j = 0$ for all j, and $T = \{(1, \mathbf{E}(a_1)), (2, \mathbf{E}(a_2)), \ldots, (n, \mathbf{E}(a_n))\}$.

We say that M **accepts** string $x \in \Sigma^*$ if the initial configuration with input x yields a halted configuration with $R_0 = 1$. We say it **rejects** x if the initial configuration with input x yields a halted configuration with $R_0 = 0$. In other words, once M halts, we read its verdict at Register 0; if this value is 1, the machine accepts, if it is 0, it rejects.

Let $\Sigma_0 \subseteq \Sigma - \{\sqcup\}$ be an alphabet, and let $L \subseteq \Sigma_0^*$ be a language. We say that M **decides** L if whenever $x \in L$, M accepts x, and whenever $x \notin L$ M rejects x. We say that M **semidecides** L if the following is true: $x \in L$ if and only if M on input x yields some halted configuration.

Finally, let $f : \Sigma_0^* \mapsto \Sigma_0^*$ be a function. We say that M **computes** f if, for all $x \in \Sigma_0^*$, the starting configuration of machine M with input x yields a halted configuration with these tape contents: $\{(1, \mathbf{E}(a_1)), (2, \mathbf{E}(a_2)), \ldots, (n, \mathbf{E}(a_n))\}$, where $f(x) = a_1 \cdots a_n$.

Example 4.4.2: We describe below a random access Turing machine program, in abbreviated form, deciding the language $\{a^n b^n c^n : n \geq 0\}$.

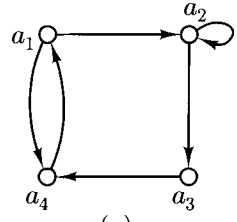
acount := *bcount* := *ccount* := 0, $n := 1$
while $T[n] = 1$ do: $n := n + 1$, *acount* := *acount* + 1
while $T[n] = 2$ do: $n := n + 1$, *bcount* := *bcount* + 1
while $T[n] = 3$ do $n := n + 1$, *acount* := *acount* + 1
if *acount* = *bcount* = *ccount* and $T[n] = 0$ then accept else reject

We are assuming here that $\mathbf{E}(a) = 1$, $\mathbf{E}(b) = 2$, $\mathbf{E}(c) = 3$, and we are using the variables *acount*, *bcount*, and *ccount* to stand for the number of a's, b's, and c's, respectively, found so far. We are also using the abbreviation accept for "load =1, halt" and reject for "load =0, halt." $\diamondsuit$

Example 4.4.3: For a more substantial example, we now describe a random access Turing machine that computes the *reflexive transitive closure* of a finite

binary relation (recall Section 1.6). We are given a directed graph $R \subseteq A \times A$, where $A = \{a_0, \ldots, a_{n-1}\}$, and we wish to compute R^*.

One important question immediately arises: How are we to *represent* a relation $R \subseteq A \times A$ as a string? R can be represented in terms of its *adjacency matrix* A_R, an $n \times n$ matrix with 0-1 entries such that the i,jth entry is 1 if and only if $(a_i, a_j) \in R$ (see Figure 4-21 for an example). In turn, the adjacency matrix can be represented as a string in $\{0,1\}^*$ of length n^2, by first arranging the first row of the matrix, then the second row, and so on. We denote the string representation of the adjacency matrix of a relation R as x_R. For example, if R is the binary relation shown in Figure 4-21(a), then A_R and x_R are as shown in Figure 4-21(b) and (c), respectively.



(a)

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

(b)

0 1 0 1 0 1 1 0 0 0 0 1 1 0 0 0

(c)

Figure 4-21: A graph, its adjacency matrix, and its string representation.

We must therefore design a random access Turing machine M that computes the function f defined as follows: For any relation R over some finite set $\{a_1, \ldots, a_n\}$, $f(x_R) = x_{R^*}$. Notice that we are not interested in how M responds to inputs that are not in $\{0,1\}^{n^2}$, that is, inputs that do not represent adjacency matrices of directed graphs.

The program of M is shown below; we are assuming that $\mathbf{E}(0) = 1$ and $\mathbf{E}(1) = 2$. As always, $\mathbf{E}(\sqcup) = 0$.

The first three instructions compute the number n of elements of the set A (one less than the smallest number whose square is the address of a blank in the input tape). From then on, the (i,j)th entry of the matrix, with $0 \leq i, j < n$, can be fetched as the $(i \cdot n + j)$-th symbol on the tape of M. Since this program is a straightforward implementation of the $\mathcal{O}(n^3)$ algorithm in Section 1.6, it is clear that it indeed computes the reflexive transitive closure of the relation represented by its input. Naturally, an unabbreviated random access Turing machine program can be mechanically derived from the program above.$\diamondsuit$

Evidently, the random access Turing machine is a remarkably powerful and

```
n := 1
while T[n · n] ≠ 0 do n := n + 1
n := n − 1
i := 0
while i < n do i := i + 1, T[i · n + i] := 2
i := j := k := 0
while j < n do j := j + 1,
  while i < n do i := i + 1,
    while k < n do k := k + 1,
      if T[i · n + j] = 2 and T[j · n + k] = 2 then T[i · n + k] := 2
halt
```

agile model. How does its power compare to that of the standard Turing machine? It is very easy to see, and not at all surprising, that *the random access Turing machine is at least as powerful as the standard Turing machine.* Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine; we can design a random access Turing machine M' that simulates M. M' has a register, call it n, that keeps track of the head position of M on its tape. Initially n points to the beginning of the input. Each state $q \in K$ is simulated by a sequence of instructions in the program of M'. For example, suppose that $\Sigma = \{\sqcup, a, b\}$, $\mathbf{E}(a) = 1$, $\mathbf{E}(b) = 2$, and let q be a state of M such that $\delta(q, \sqcup) = (p, \rightarrow)$, $\delta(q, a) = (p, \leftarrow)$, $\delta(q, b) = (r, \sqcup)$, and $\delta(q, \triangleright) = (s, \rightarrow)$. The sequence of instructions simulating state q is this:

```
q:  if T[n] = 0 then n := n + 1, goto p
    if T[n] = 1 then if n > 0 then n := n − 1, goto p
      else goto s
    if T[n] = 2 then T[n] := 0, goto r
```

The else clause in the third line (which should be present in any line simulating a $\leftarrow$ move) has the effect of the $\triangleright$ symbol, making sure that the head never falls off the left end of M''s tape. We have shown:

Theorem 4.4.1: *Any recursive or recursively enumerable language, and any recursive function, can be decided, semidecided, and computed, respectively, by a random access Turing machine.*

The remarkable direction is the opposite:

Theorem 4.4.2: *Any language decided or semidecided by a random access Turing machine, and any function computable by a random access Turing machine, can be decided, semidecided, and computed, respectively, by a standard Turing machine. Furthermore, if the machines halt on an input, then the number of steps taken by the standard Turing machine is bounded by a polynomial in the number of steps of the random access Turing machine on the same input.*

Proof: Let $M = (k, \Pi)$ be a random access Turing machine deciding or semideciding a language $L \subseteq \Sigma^*$ or computing a function from Σ^* to Σ^*. We will outline the design an ordinary Turing machine M' that simulates M. We shall describe M' as a $(k+3)$-tape machine, where k is the number of registers of M, which simulates M; we know from Theorem 4.3.1 that such a machine can in turn be simulated by the basic model.

Turing machine M' keeps track of the current configuration of the random access Turing machine M, and repeatedly computes the next configuration. The first tape is used only for reading the input of M, and possibly for reporting the output at the end, in the case where M computes a function. The second tape is used for keeping track of the T part of the configuration —the tape contents of M. The relation T is maintained as a sequence of strings of the form $(111, 10)$, a left parenthesis followed by the binary representation of an integer, followed by a comma, followed by another binary integer, followed by a right parenthesis. The intended meaning of the above string is that the seventh tape square of M contains the integer 2. The pairs of integers in the sequence representing T are not in any particular order, and may be separated by arbitrarily long sequences of blanks (this necessitates an *endmarker,* such as \$, at the end of the representation of T, to help M' decide when it has seen all such pairs). Each of the next k tapes of M' maintain the contents of a register of M, also in binary. The current value of the program counter κ of M is maintained in the state of M' in a manner to be explained below.

The simulation has three phases. During the first phase, M' receives on its first tape the input $x = a_1 a_2 \ldots a_n \in \Sigma^*$, and converts it to the string $(1, \mathbf{E}(a_1)) \ldots (n, \mathbf{E}(a_n))$ on the second tape. Thus M' can start the second phase, the simulation of M, from the initial configuration of M on input x.

During the second phase M' repeatedly simulates a step of M by several steps of its own. The precise nature of the step to be simulated depends heavily on the program counter κ of M. As we said before, κ is maintained in the state of M'. That is, the set of states of M' that are used during this phase are separated into p disjoint sets $K_1 \cup K_1 \cup \cdots \cup K_p$, where p is the number of instructions in the program Π of M. The set of states K_j "specializes" in simulating the instruction π_j of Π. The precise nature of this part of M' depends of course on the kind of the instruction π_j. We shall give three indicative examples of how this is done.

Suppose first that π_j is add 4, requiring that the contents of Register 4 be added to those of Register 0. Then M' will perform binary addition (recall Example 4.3.2) between its two tapes representing Registers 4 and 0, will leave the result in the tape for Register 0, and then move to the first state of K_{j+1} to start the simulation of the next instruction. If the instruction is, say, add = 33, then M' will start by writing the integer 33 in binary on the $(k+3)$rd tape (the one heretofore unassigned to parts of M); the binary representation

of the fixed integer 33 is "remembered" by the states of K_j. Then it will add 33 to the contents of Register 0, and finally it will erase the last tape and will move to K_{j+1}.

Suppose next that π_j is write 2, requiring that the contents of the accumulator be copied to the tape square pointed at by Register 2. Then M' will add at the right end of the second tape —the one where the contents of the tape of M are kept in the (x, y) format— the pair (x, y), where x is the contents of Register 2 and y those of Register 0; both x and y are copied from the corresponding tapes of M'. M' will then scan all other pairs (x', y') on the second tape, comparing each x', bit by bit, with the contents of Register i. If a match is found, the pair is erased, thus maintaining the integrity of the table. Then the state moves to K_{j+1}, and the next instruction is executed.

Suppose now that π_j is jpos 19, requiring that instruction 19 be executed next if Register 0 contains a positive integer. Turing machine M' simply scans its tape representing Register 0; if a 1 is found in the binary representation of the integer in it, M moves to K_{19}; otherwise it moves to K_{j+1}.

It is straightforward to simulate, in a very similar manner, all other kinds of instructions in the table of Figure 4-19. Eventually, M may reach a halt instruction. If this happens, M' enters its third phase, translating M's output to the Turing machine output conventions. If M is deciding a language, then M' would read the contents of Register 0. If they are 1, it will halt at state y, if they are 0 it will halt at state n. If M is semideciding a language, then M' simply halts at state h. Finally, if M is computing a function, then M must translate the contents of the tape of M to a string in Σ^*, inverting the bijection **E**, and then halt.

It is clear from the preceding discussion that a $k + 3$-tape Turing machine M' can be designed that performs the above tasks —and hence, by Theorem 4.3.1, a standard Turing machine can.

To prove the second part of the theorem, we shall establish that t steps of M on an input of size n can be simulated in $\mathcal{O}(t + n)^3$ time. Naturally, the constants in the $\mathcal{O}$ notation will, as usual, depend on the simulated machine M; for example they will depend on the largest constant (as in add = 314159) mentioned in the program of M.

The $\mathcal{O}(t + n)^3$ bound is based on the following three observations:

(a) At each step of M (including the addition and subtraction steps, see Problem 4.4.3) can be simulated in $\mathcal{O}(m)$ steps of M', where m is the total length of the nonblank parts of all tapes of M' —that is to say, the total length of the binary encodings of all integers in the current configuration of M.

(b) The parameter m defined above can at each step increase by at most $\mathcal{O}(r)$, where r is the length of the longest binary representation of any integer stored in the registers or tape squares of M. This is so because the increase comes either from an add instruction or from a store instruction, and in

both cases it is trivial to see that the increase can only be linear in r.

(c) Finally, it is easy to see that $r = \mathcal{O}(t)$; that is, the length of the largest integer represented by M can only increase by a constant at each step. The claimed bound follows by putting these three facts together. ■

Problems for Section 4.4

4.4.1. Give explicitly the full details of the random access Turing machine program of Example 4.4.2. Give the sequence of configurations of this machine on input *aabccc*.

4.4.2. Give (in our abbreviated notation) a random access Turing machine program that decides the language $\{wcw : w \in \{a, b\}^*\}$.

4.4.3. Show that, in the simulation in the proof of Theorem 4.4.2, each step can be simulated by $\mathcal{O}(m)$ steps of M', where m is the total length of M''s tapes. (You must establish that the 2-tape addition Turing machine in Example 4.3.2 operates in linear time.) Can you estimate the constant in $\mathcal{O}(m)$?

4.4.4. Suppose that our random access Turing machines had an explicit instruction mply. What goes wrong now in the second part of the proof of Theorem 4.4.2?

4.5 NONDETERMINISTIC TURING MACHINES

We have added to our Turing machines many seemingly powerful features -- multiple tapes and heads, even random access— with no appreciative increase in power. There is, however, an important and familiar feature that we have not tried yet: *nondeterminism.*

We have seen that when finite automata are allowed to act nondeterministically, no increase in computational power results (except that exponentially fewer states may be needed for the same task), but that nondeterministic pushdown automata are more powerful than deterministic ones. We can also imagine Turing machines that act nondeterministically: Such machines might have, on certain combinations of state and scanned symbol, more than one possible choice of behavior. 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.

When Turing machines are allowed to act nondeterministically, is there any increase in computational power? We must first define what it means for a nondeterministic Turing machine to compute something. 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 4.5.1: 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 4.5.2: 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 $v = f(w)$.

These definitions reflect the difficulties associated with computing by nondeterministic Turing machines. First notice that, for a nondeterministic machine to decide a language and compute a function, we require that *all* of its computations halt; we achieve this by postulating that there is no computation continuing after N steps, where N is an "upper bound" depending on the machine and the input (this is condition (a) above). Second, for M to decide a language, we only require that *at least* one of its possible computations end up accepting the input. Some, most, or all of the remaining computations could end up rejecting —the machine accepts by the force of this single accepting computation. This is a very unusual, asymmetric, and counterintuitive convention. For example, to

create a machine that decides the *complement* of the language, it is not enough to reverse the rôles of the y and n states (since the machine may have both accepting and rejecting computations for some inputs). As with nondeterministic finite automata, to show that the class of languages decided by nondeterministic Turing machines is closed under complement, we must go through an equivalent *deterministic* Turing machine —and our main result in this section (Theorem 4.5.1) states that an equivalent deterministic Turing machine must exist.

Finally, for a nondeterministic Turing machine to compute a *function*, we require that *all* possible computations agree on the outcome. If not, we would not be able to decide which one is the right value of the function (in the cases of deciding or semideciding a language, we resolve this uncertainty by postulating that the positive answer prevails).

Before showing that nondeterminism, like the features considered in the previous sections, can be eliminated from Turing machines, let us consider a classic example that demonstrates the power of nondeterminism in Turing machines as a conceptual device.

Example 4.5.1: A **composite number** is one that is the product of two natural numbers, each greater than one; for example, 4, 6, 8, 9, 10, and 12 are composite, but 1, 2, 3, 5, 7, and 11 are not. In other words, a composite number is a non-prime other than one or zero.

Let $C = \{100, 110, 1000, 1001, 1010, \ldots, 1011011, \ldots\}$ be the set of all binary representations of composite numbers. To design an "efficient" algorithm deciding C is an ancient, important, and difficult problem. To design such an algorithm it would seem necessary to come up with a clever way of discovering the *factors*, if any, of a number —a task that seems quite complex. Naturally, by searching exhaustively all numbers smaller than the given number (in fact, smaller than its *square root*) we would end up discovering its factors; the point is that *no more direct method is evident.*

However, if nondeterminism is available, we can design a machine to semidecide C rather simply, by guessing the factors, if there are any. This machine operates as follows, when given as input the binary representation of integer n:

(1) Nondeterministically choose two binary numbers p, q larger than one, bit by bit, and write their binary representation next to the input.
(2) Use the multiplication machine of Problem 4.3.5 (actually, the single-tape machine that simulates it) to replace the binary representations of p and q by that of their product.
(3) Check to see that the two integers, n and $p \cdot q$, are the same. This can be done easily by by comparing them bit by bit. Halt if the two integers are equal; otherwise continue forever in some fashion (recall that at present we are only interested in a machine that semidecides C).

This machine, on input 1000010 (the binary representation of 66) will have many rejecting (nonhalting) computations, corresponding to phase 1 above choosing pairs of binary integers, such as 101; 11101, that *fail* to multiply to 66. The point is that, since 66 is a composite number, there will be *at least one* computation of M that will end up accepting —and that is all we need. In fact, there will be more than one (corresponding to $2 \cdot 33 = 6 \cdot 11 = 11 \cdot 6 = 33 \cdot 2 = 66$). If the input were 1000011, however, no computation would end up accepting —because 67 is a prime number.

This machine can be modified to a nondeterministic machine that *decides* the language C. The deciding machine has the same basic structure, except that in Phase (1) the new machine never guesses an integer with more bits than n itself —obviously, such an integer cannot be a factor of n. And in Phase 3, after comparing the input and the product, the new machine would halt at state y if they are equal, and at state n otherwise. As a result, *all computations will eventually halt* after some finite number of steps.

The upper bound N required by Definition 4.5.2 is now easy to compute explicitly. Suppose that the given integer n has ℓ bits. Let N_1 be the maximum number of steps in any computation by the multiplying machine on any input of length $2\ell+1$ or less; this is a finite number, the maximum of finitely many natural numbers. Let N_2 the number of steps it takes to compare two strings of length at most 3ℓ each. Then any computation by M' will halt after $N_1 + N_2 + 3\ell + 6$ steps, certainly a finite number depending only on the machine and the input.$\diamondsuit$

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 4.5.1: *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: We shall describe the construction for the case in which M semidecides a language L; the constructions for the case of deciding a language or computing a function are very similar. So, 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.

But M may have an infinity of different computations starting from the same input; how can M explore them all? It does so by using a *dovetailing procedure* (recall the argument illustrated in Figure 1-8). The crucial observation is the following: Although for any configuration C of M there may be several configurations C' such that $C \vdash C'$, the *number* of such configurations C' is fixed and bounded in a way that depends only on M, not on C. Specifically, the number of quadruples $(q, a, p, b) \in \Delta$ that can be applicable at any point is finite; in fact, it cannot exceed $|K| \cdot (|\Sigma| + 2)$, since this is the maximum number of possible combinations (p, b) with $p \in K$ and $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$. Let us call r the maximum number of quadruples that can be applicable at any point; the number r can be determined by inspection of M. In fact, we shall assume that for each state-symbol combination (q, a), and for each integer $i \in \{1, 2, \ldots, r\}$, there is a well-defined ith applicable quadruple (q, a, p_i, b_i). If for some state-symbol combination (q, a) there are fewer than r relevant quadruples in Δ, then some may be repeated.

Since M is nondeterministic, it has no definite way to decide at each step how to choose among its r available "choices." But suppose that *we help it decide.* To be precise, let M_d, the *deterministic version of* M, be a device with the same set of states as M, but with two tapes. The first tape is the tape of M, initially containing the input w, while the second tape initially contains a string of n integers in the range $1, \ldots, r$, say $i_1 i_2 \ldots i_n$. M_d then operates as follows for n steps: In the first step, among the r possible next state-action combinations $(p_1, b_1), \ldots, (p_r, b_r)$ that are applicable to the initial configuration, M_d chooses the i_1th —that is, (p_{i_1}, b_{i_1}), the one suggested by the currently scanned symbol in the second tape, i_1. M_d also moves its second tape head to the right, so that it next scans i_2. In the next step, M_d takes the i_2th combination, then the i_3th, and so on. When M_d sees a blank on its second tape, meaning that it has run out of "hints," it halts.

M_d is an important ingredient in our design of the deterministic Turing machine M' that simulates M. We shall describe M' as a 3-tape Turing machine; we know by Theorem 4.3.1 that M' can be converted into an equivalent single-tape Turing machine. The three tapes of M' are used as follows:

(1) The first tape is never changed; it always contains the original input w, so that each simulated computation of M can begin afresh 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 $1, 2, \ldots, r, 11, 12, \ldots, rr, 111, \ldots$. For $r = 2$, N is precisely the Turing machine that computes the binary successor function (Example 4.2.3); its generalization to $r > 2$ is rather straightforward.

M' is is the Turing machine given in Figure 4-22. By $C^{1\to 2}$ we mean a simple Turing machine that erases the second tape and copies the first tape on the second. B^3 is the machine that generates the lexicographically next string in the third tape. Finally, $M_d^{2,3}$ is the deterministic version of M, operating on tapes 2 and 3. This completes the description of M'.

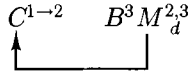


Figure 4-22

We claim that M' halts on an input w if and only if (some computation of) M does. Suppose that M' indeed halts on input w; by inspecting Figure 4-22, this means that M_d halts with its third tape head not scanning a blank. This implies that, for some string $i_1 i_2 \cdots i_n \in \{1, 2, \ldots, r\}^*$, M_d, when started with w on its first tape and $i_1 i_2 \cdots i_n$ on its second, halts before reaching the blank part of its second tape. This, however, means that there is a computation of M on input w that halts. Conversely, if there is a halting computation of M on input w, say with n steps, then M', after at most $r + r^2 + \cdots + r^n$ failed attempts, the string in $\{1, 2, \ldots, r\}^*$ corresponding to the choices of M's halting computation will be generated by B^3, and M_d will halt scanning the last symbol of this string. Thus M' will halt, and the proof is complete. ■

As we had expected, the simulation of a nondeterministic Turing machine by a deterministic one is not a step-by-step simulation, as were all other simulations we have seen in this chapter. 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 —whereas all other simulations described in this chapter are in fact *polynomial*. 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, explored in Chapters 6 and 7 of this book.

Problems for Section 4.5

5.1. Give (in abbreviated notation) nondeterministic Turing machines that accept these languages.

(a) $a^*abb^*baa^*$
(b) $\{ww^Ruu^R : w, u \in \{a, b\}^*\}$

4.5.2. Let $M = (K, \Sigma, \delta, s, \{h\})$ be the following nondeterministic Turing machine:

$$\begin{aligned} K &= \{q_0, q_1, h\}, \\ \Sigma &= \{a, \triangleright, \sqcup\}, \\ s &= q_0, \\ \Delta &= \{(q_0, \sqcup, q_1, a), (q_0, \sqcup, q_1, \sqcup), (q_1, \sqcup, q_1, \sqcup), (q_1, a, q_0, \rightarrow), (q_1, a, h, \rightarrow)\} \end{aligned}$$

Describe all possible computations of five steps or less by M starting from the configuration $(q_0, \triangleright\underline{\sqcup})$. Explain in words what M does when started from this configuration. What is the number r (in the proof of Theorem 4.5.1) for this machine?

4.5.3. Although nondeterministic Turing machines are not helpful in showing closure under complement of the recursive languages, they are very convenient for showing other closure properties. Use nondeterministic Turing machines to show that the class of recursive languages is closed under union, concatenation, and Kleene star. Repeat for the class of recursively enumerable languages.

4.6 GRAMMARS

In this chapter we have introduced several computational devices, namely the Turing machine and its many extensions, and we have demonstrated that *they are all equivalent in computational power.* All these various species of Turing machines can be reasonably called *automata*, like their weaker relatives —the finite automata and the pushdown automata— studied in previous chapters. Like those 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.

But in previous chapters we have seen that there is another important family of devices, very different in spirit from language acceptors, that can be used to define interesting classes of languages: *language generators*, such as regular expressions and context-free grammars. In fact, we have demonstrated that these two formalisms provide valuable *alternative characterizations* of the classes of languages defined by language acceptors. This chapter would not be complete without such a maneuver: 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.

Let us recall the essential features of a context-free grammar. It has an alphabet, V, which is divided into two parts, the set of terminal symbols, Σ, and the set of nonterminal symbols, $V - \Sigma$. It also has a finite set of rules, each of the form $A \to u$, where A is a nonterminal symbol and $u \in V^*$. A context-free grammar operates by starting from the start symbol S, a nonterminal, and repeatedly replacing the left-hand side of a rule by the corresponding right-hand side until no further such replacements can be made.

In a grammar all the same conventions apply, *except that the left-hand sides of rules need not consist of single nonterminals.* Instead, the left-hand side of a rule may consist of any string of terminals and nonterminals containing at least one nonterminal. A single step in a derivation entails removing the entire substring on the left-hand side of a rule and replacing it by the corresponding right-hand side. The final product is, as in context-free grammars, a string containing terminals only.

Definition 4.6.1: 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 $w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$.

Example 4.6.1: Any context-free grammar is a grammar; in fact, a context-free grammar is a grammar such that the left-hand side of each rule is a member of $V - \Sigma$, rather than $V^*(V - \Sigma)V^*$. Thus, in a grammar, a rule might have the form $uAv \to uwv$, which could be read "replace A by w *in the context of* u *and* v." Of course, the rules of a grammar may be of a form even more general than this; but it turns out that any language that can be generated by a grammar can be generated by one in which all rules are of this "context-dependent replacement" type (Problem 4.6.3).$\diamondsuit$

Example 4.6.2: The following grammar G generates the language $\{a^n b^n c^n : n \geq 1\}$. $G = (V, \Sigma, R, S)$, where

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

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.

It is rather obvious that any string of the form $a^n b^n c^n$ can be produced this way. Of course, many more strings that contain nonterminals can be produced; however, it is not hard to see that the only way to erase all nonterminals is to follow the procedure outlined above. Thus, the only strings in $\{a, b, c\}$ that can be generated by G are those in $\{a^n b^n c^n : n \geq 1\}$.$\Diamond$

Evidently, the class of languages generated by grammars contains certain non-context-free specimens. But precisely how large is this class of languages? More importantly, how does it relate to the other two important extensions of the context-free languages we have seen in this chapter, namely, the recursive and the recursively enumerable languages? As it happens, grammars play with respect to Turing machines precisely the same rôle that context-free grammars play in relation to pushdown automata, and regular expressions to finite automata:

Theorem 4.6.1: *A language is generated by a grammar if and only if it is recursively enumerable.*

Proof: *Only if.* Let $G = (V, \Sigma, R, S)$ be a grammar. We shall design a Turing machine M that semidecides the language generated by G. In fact, M will be *nondeterministic;* its conversion to a deterministic machine that semidecides the same language is guaranteed by Theorem 4.5.1.

M has three tapes. The first tape contains the input, call it w, and is never changed. In the second tape, M tries to reconstruct a derivation of w from S in the grammar G; M therefore starts by writing S on the second tape. Then M proceeds in steps, corresponding to the steps of the derivation being constructed. Each step starts with a nondeterministic transition, guessing one between $|R|+1$ possible states. Each of the first $|R|$ of these $|R|+1$ states is the beginning of a sequence of transitions that applies the corresponding rule to the current contents of the second tape. Suppose that the chosen rule is $u \to v$. M then scans its second tape from left to right, nondeterministically stopping at some symbol. It then checks that the next $|u|$ symbols match u, erases u, shifts the rest of the string appropriately to make just enough space for v, and writes v in u's place. If the check fails, M enters an unending computation —the present attempt at generating w has failed.

The $|R|+1$st choice of M entails checking whether the current string equals w, the input. If so, M halts and accepts: w can indeed be generated by G. And if the strings are found unequal, M again loops forever.

It is clear that the only possible halting computations of M are those that correspond to a derivation of w in G. Thus M accepts w if and only if $w \in L(G)$, and the *only if* direction has been proved.

If. Suppose now that $M = (K, \Sigma, \delta, s, \{h\})$ is a Turing machine. It will be convenient to assume that Σ and K are disjoint, and that neither contains the new endmarker symbol $\triangleleft$. We also assume that M, if it halts, it always does so in the configuration $(h, \triangleright\underline{\sqcup})$ —that is, after having erased its tape. Any Turing machine that semidecides a language can be transformed into an equivalent one that satisfies the above conditions. We shall construct a grammar $G = (V, \Sigma - \{\sqcup, \triangleright\}, R, S)$ that generates the language $L \subseteq (\Sigma - \{\sqcup, \triangleright\})^*$ semidecided by M.

The alphabet V consists of all symbols in Σ and all states in K, plus the start symbol S and the endmarker $\triangleleft$. Intuitively, the derivations of G will simulate *backward computations* of M. We shall simulate configuration $(q, \triangleright u\underline{a}w)$ by the string $\triangleright uaqw\triangleleft$ —that is, by the tape contents, with the current state inserted immediately after the currently scanned symbol, and with the endmarker $\triangleleft$ appended at the end of the string. The rules of G simulate *backwards moves of* M. That is, for each $q \in K$ and $a \in \Sigma$, G has these rules, depending on $\delta(q, a)$.

(1) If $\delta(q, a) = (p, b)$ for some $p \in K$ and $b \in \Sigma$, then G has a rule $bp \to aq$.
(2) If $\delta(q, a) = (p, \to)$ for some $p \in K$, then G has a rule $abp \to aqb$ for all $b \in \Sigma$, and also the rule $a \sqcup p\triangleleft \to aq\triangleleft$ (the last rule reverses the extension

of the tape to the right by a new blank).

(3) If $\delta(q, a) = (p, \leftarrow)$ for some $p \in K$, and $a \neq \sqcup$, then G has a rule $pa \rightarrow aq$.

(4) If $\delta(q, \sqcup) = (p, \leftarrow)$ for some $p \in K$, then G has a rule $pab \rightarrow aqb$ for all $b \in \Sigma$, and also the rule $p\triangleleft \rightarrow \sqcup q\triangleleft$ that reverses the erasing of extraneous blanks.

Finally, G contains certain transitions for the beginning of the computation (the end of the derivation) and the end of the computation (the beginning of the derivation). The rule

$$S \rightarrow \triangleright \sqcup h\triangleleft$$

forces the derivation to start exactly where an accepting computation would end. The other rules are $\triangleright \sqcup s \rightarrow e$, erasing the part of the final string to the left of the input, and $\triangleleft \rightarrow e$, erasing the endmarker and leaving the input string.

The following result makes precise our notion that G simulates backward computations of M:

Claim: *For any two configurations* $(q_1, u_1\underline{a_1}w_1)$ *and* $(q_2, u_2\underline{a_2}w_2)$ *of* M, *we have that* $(q_1, u_1\underline{a_1}w_1) \vdash_M (q_2, u_2\underline{a_2}w_2)$ *if and only if* $u_2a_2q_2w_2\triangleleft \Rightarrow_G u_1a_1q_1w_1\triangleleft$.

The proof of the claim is a straightforward case analysis on the nature of the move M, and is left as an exercise.

We now complete the proof of the theorem, by showing that, for all $w \in (\Sigma - \{\triangleright, \sqcup\})^*$, M halts on w if and only if $w \in L(G)$. $w \in L(G)$ if and only if

$$S \Rightarrow_G \triangleright \sqcup h\triangleleft \Rightarrow_G^* \triangleright \sqcup sw\triangleleft \Rightarrow_G w\triangleleft \Rightarrow_G w,$$

because $S \rightarrow \triangleright \sqcup h\triangleleft$ is the only rule that involves S, and the rules $\triangleright \sqcup s \rightarrow e$ and $\triangleleft \rightarrow e$ are the only rules that allow for the eventual erasing of the state and the endmarker $\triangleleft$. Now, by the claim, $\triangleright \sqcup h\triangleleft \Rightarrow_G^* \triangleright \sqcup sw\triangleleft$ if and only if $(s, \triangleright\underline{\sqcup}w) \vdash_M^* (h, \triangleright\underline{\sqcup})$, which happens if and only if M halts on w. This completes the proof of the Theorem. ■

Theorem 4.6.1 identifies grammars with an aspect of the Turing machines that we have deemed unrealistic —semidecision, with its one-sided definition that provides no information when the input is not in the language. This is consistent with what we know about grammars: If a string can be generated by the grammar, we can patiently search all possible derivations, starting from the shorter ones and proceeding to the longer ones, until we find the correct one. But if no derivation exists, this process will go on indefinitely, without giving us any useful information.

As it turns out, we can also identify grammars with the more useful modes of computation based on Turing machines.

Definition 4.6.2: Let $G = (V, \Sigma, R, S)$ be a grammar, and let $f : \Sigma^* \mapsto \Sigma^*$ be a function. We say that 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 called **grammatically computable** if and only if there is a grammar G that computes it.

We leave the proof of the following result —a modification of the proof of Theorem 4.6.1— as an exercise, see Problem 4.6.4:

Theorem 4.6.2: *A function $f : \Sigma^* \mapsto \Sigma^*$ is recursive if and only if it is grammatically computable.*

Problems for Section 4.6

4.6.1. (a) Give a derivation of the string *aaabbbccc* in the grammar of Example 4.6.2.

(b) Prove carefully that the grammar in Example 4.6.2 generates the language $L = \{a^n b^n c^n : n \geq 1\}$.

4.6.2. Find grammars that generate the following languages:

(a) $\{ww : w \in \{a, b\}^*\}$

(b) $\{a^{2^n} : n \geq 0\}$

(c) $\{a^{n^2} : n \geq 0\}$

4.6.3. Show that any grammar can be converted into an equivalent grammar with rules of the form $uAv \rightarrow uwv$, with $A \in V - \Sigma$, and $u, v, w \in V^*$.

4.6.4. Prove Theorem 4.6.2. (For the *only if* direction, given a grammar G, show how to construct a Turing machine which, on input w, outputs a string $u \in \Sigma^*$ such that $SwS \Rightarrow_G^* u$, if such a string u exists. For the *if* direction, use a simulation similar to the proof of Theorem 4.6.1, except in the opposite (forward) direction.)

4.6.5. An oddity in the use of grammars to compute functions is that the order in which rules are applied is indeterminate. In the following alternative, due to A. A. Markov (1903–1979), this indeterminacy is avoided. A **Markov system** is a quadruple $G = (V, \Sigma, R, R_1)$, where V is an alphabet; $\Sigma \subseteq V$; R is a finite *sequence* (not set) of rules $(u_1 \rightarrow v_1, \ldots, u_k \rightarrow v_k)$, where $u_i, v_i \in V^*$; and R_1 is a set of rules from R. The relation $w \Rightarrow_G w'$ is defined as follows: If there is an i such that u_i is a substring of w, then let

i be the smallest such number, and let w_1 be the shortest string such that $w = w_1 u_i w_2$; then $w \Rightarrow_G w'$ provided that $w' = w_1 v_i w_2$. Thus if a rule is applicable, then there is at most one rule, and it is applicable in exactly one position. We say that G **computes** a function $f : \Sigma^* \mapsto \Sigma^*$ if for all $u \in \Sigma^*$

$$u = u_0 \Rightarrow_G u_1 \rightarrow_G \cdots \Rightarrow_G u_{n-1} \Rightarrow_G u_n = f(u),$$

and the first time that a rule from R_1 was used was the last one, $u_{n-1} \Rightarrow_G u_n$. Show that a function is computable by a Markov system if and only if it is recursive. (The proof is similar to that of Theorem 4.6.2.)

4.7 NUMERICAL FUNCTIONS

Let us now adopt a completely different point of view on computation, one that is not based on any explicit computational or information-processing formalism such as Turing machines or grammars, but instead focuses on what has to be computed: *functions from numbers to numbers.* For example, it is clear that the value of the function

$$f(m,n) = m \cdot n^2 + 3 \cdot m^{2 \cdot m + 17}$$

can be computed for any given values of m and n, because it is the composition of functions —addition, multiplication, and exponentiation, plus a few constants— that can be computed. And how do we know that exponentiation can be computed? Because it is *recursively defined* in terms of a simpler function (namely, multiplication) and values at smaller arguments. After all, m^n is 1 if $n = 0$, and otherwise it is $m \cdot m^{n-1}$. Multiplication itself can be defined recursively in terms of addition —and so on.

In principle, we should be able to start with functions from natural numbers to natural numbers that are so simple that they will be unequivocally considered computable (e.g., the identity function and the successor function $\mathsf{succ}(n) = n + 1$), and combine them slowly and patiently through combinators that are also very elementary and obviously computable —such as composition and recursive definition— and finally get a class of functions from numbers to numbers that are quite general and nontrivial. In this section we shall undertake this exercise. Significantly, the notion of computation thus defined will then be proved identical to the notions arrived at by the other approaches of this chapter —Turing machines, their variants, and grammars— that are so different in spirit, scope, and detail.

Definition 4.7.1: We start by defining certain extremely simple functions from $\mathbf{N}^k$ to $\mathbf{N}$, for various values of $k \geq 0$ (a 0-ary function is, of course, a constant, as it has nothing on which to depend). The **basic functions** are the following:

(a) For any $k \geq 0$, the **k-ary zero function** is defined as $\mathsf{zero}_k(n_1, \ldots, n_k) = 0$ for all $n_1, \ldots, n_k \in \mathbf{N}$.

(b) For any $k \geq j > 0$, the **jth k-ary identity function** is simply the function $\mathsf{id}_{k,j}(n_1, \ldots, n_k) = n_j$ for all $n_1, \ldots, n_k \in \mathbf{N}$.

(c) The **successor function** is defined as $\mathsf{succ}(n) = n + 1$ for all $n \in \mathbf{N}$.

Next we introduce two simple ways of combining functions to get slightly more complex functions.

(1) Let $k, \ell \geq 0$, let $g : \mathbf{N}^k \mapsto \mathbf{N}$ be a k-ary function, and let $h_1, \ldots, h_k$ be ℓ-ary functions. Then the **composition of g with $h_1, \ldots, h_k$** is the ℓ-ary function defined as

$$f(n_1, \ldots, n_\ell) = g(h_1(n_1, \ldots, n_\ell), \ldots, h_k(n_1, \ldots, n_\ell)).$$

(2) Let $k \geq 0$, let g be a k-ary function, and let h be a $(k+2)$-ary function. Then the **function defined recursively by g and h** is the $(k+1)$-ary function f defined as

$$\begin{aligned} f(n_1, \ldots, n_k, 0) &= g(n_1, \ldots, n_k), \\ f(n_1, \ldots, n_k, m+1) &= h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)) \end{aligned}$$

,

for all $n_1, \ldots, n_k, m \in \mathbf{N}$.

The **primitive recursive functions** are all basic functions, and all functions that can be obtained by them by any number of successive applications of composition and recursive definition.

Example 4.7.1: The function plus2, defined as $\mathsf{plus2}(n) = n + 2$ is primitive recursive, as it can be obtained from the basic function succ by composition with itself. In particular, let $k = \ell = 1$ in (1) of Definition 4.7.1, and let $g = h_1 = \mathsf{succ}$.

Similarly, the binary function plus, defined as $\mathsf{plus}(m, n) = m + n$ is primitive recursive, because it can be recursively defined from functions obtained by combining identity, zero, and successor functions. In particular, in Part 2 of Definition 4.7.1 set $k = 1$, take g to be the $\mathsf{id}_{1,1}$ function, and let h be the ternary function $h(m, n, p) = \mathsf{succ}(\mathsf{id}_{3,3}(m, n, p))$ —the composition of succ with $\mathsf{id}_{3,3}$. The resulting recursively defined function is precisely the plus function:

$$\begin{aligned} \mathsf{plus}(m, 0) &= m, \\ \mathsf{plus}(m, n+1) &= \mathsf{succ}(\mathsf{plus}(m, n)). \end{aligned}$$

Why stop? The function multiplication $\mathsf{mult}(m,n) = m \cdot n$ is defined recursively as

$$\mathsf{mult}(m,0) = \mathsf{zero}(m),$$
$$\mathsf{mult}(m,n+1) = \mathsf{plus}(m, \mathsf{mult}(m,n)),$$

and the function $\mathsf{exp}(m,n) = m^n$ is defined as

$$\mathsf{exp}(m,0) = \mathsf{succ}(\mathsf{zero}(m)),$$
$$\mathsf{exp}(m,n+1) = \mathsf{mult}(m, \mathsf{exp}(m,n)).$$

Hence all these functions are primitive recursive.

All *constant* functions of the form $f(n_1,\ldots,n_k) = 17$ are primitive recursive, since they can be obtained by composing an appropriate zero function with the succ function, in this example seventeen times. Also, the *sign* function $\mathsf{sgn}(n)$, which is zero if $n = 0$, and otherwise it is one, is also primitive recursive: $\mathsf{sgn}(0) = 0$, and $\mathsf{sgn}(n+1) = 1$.

For better readability, we shall henceforth use $m + n$ instead of $\mathsf{plus}(m,n)$, $m \cdot n$ instead of $\mathsf{mult}(m,n)$, and $m \uparrow n$ instead of $\mathsf{exp}(m,n)$. All numerical functions such as

$$m \cdot (n + m^2) + 178^m$$

are thus primitive recursive, since they are obtained from the ones above by successive compositions.

Since we are confined within the natural numbers, we cannot have true subtraction and division. However, we can define certain useful functions along these lines, such as $m \sim n = \max\{m-n, 0\}$, and the functions $\mathsf{div}(m,n)$ and $\mathsf{rem}(m,n)$ (the integer quotient and remainder of the division of m by n; assume that they are both 0 if $n = 0$). First define the *predecessor* function:

$$\mathsf{pred}(0) = 0,$$
$$\mathsf{pred}(n+1) = n,$$

from which we get our "nonnegative subtraction" function

$$m \sim 0 = m,$$
$$m \sim n+1 = \mathsf{pred}(m \sim n).$$

The quotient and remainder functions will be defined in a subsequent example.$\diamondsuit$

It is rather clear that we can calculate the value of any primitive recursive function for given values of its arguments. It is equally self-evident that we can calculate the validity of *assertions* about numbers such as

$$m \cdot n > m^2 + n + 7,$$

for any given values of m and n. It is convenient to define a **primitive recursive predicate** to be a primitive recursive function that only takes values 0 and 1. Intuitively, a primitive recursive predicate, such as greater-than(m, n), will capture a *relation* that may or may not hold between the values of m and n. If the relation holds, then the primitive recursive predicate will evaluate to 1, otherwise to 0.

Example 4.7.2: The function iszero, which is 1 if $n = 0$, and 0 if $n > 0$, is a primitive recursive predicate, defined recursively thus:

$$\begin{aligned} \text{iszero}(0) &= 1, \\ \text{iszero}(m+1) &= 0. \end{aligned}$$

Similarly, isone$(0) = 0$, and isone$(n+1) =$ iszero(n). The predicate positive(n) is the same as the already defined sgn(n). Also, greater-than-or-equal(m, n), written $m \geq n$, can be defined as iszero$(n \sim m)$. Its *negation*, less-than(m, n) is of course $1 \sim$ greater-than-or-equal(m, n). In general, the negation of any primitive recursive predicate is also a primitive recursive predicate. In fact, so are the *disjunction* and *conjunction* of two primitive recursive predicates: $p(m, n)$ or $q(m, n)$ is $1 \sim$ iszero$(p(m, n) + q(m, n))$, and $p(m, n)$ and $q(m, n)$ is $1 \sim$ iszero$(p(m, n) \cdot q(m, n))$. For example, equals(m, n) can be defined as the conjunction of greater-than-or-equal(m, n) and greater-than-or-equal(n, m).

Furthermore, if f and g are primitive recursive functions and p is a primitive recursive predicate, all three with the same arity k, then the **function defined by cases**

$$f(n_1, \ldots, n_k) = \begin{cases} g(n_1, \ldots, n_k), & \text{if } p(n_1, \ldots, n_k); \\ h(n_1, \ldots, n_k), & \text{otherwise} \end{cases}$$

is also primitive recursive, since it can be rewritten as:

$$f(n_1, \ldots, n_k) = p(n_1, \ldots, n_k) \cdot g(n_1, \ldots, n_k) + (1 \sim p(n_1, \ldots, n_k)) \cdot h(n_1, \ldots, n_k).$$

As we shall see, definition by cases is a very useful shorthand.$\diamondsuit$

Example 4.7.3: We can now define div and rem.

$$\begin{aligned} \text{rem}(0, n) &= 0, \\ \text{rem}(m+1, n) &= \begin{cases} 0 & \text{if equal}(\text{rem}(m, n), \text{pred}(n)); \\ \text{rem}(m, n) + 1 & \text{otherwise,} \end{cases} \end{aligned}$$

and

$$\mathsf{div}(0,n) = 0,$$
$$\mathsf{div}(m+1,n) = \begin{cases} \mathsf{div}(m+1,n)+1 & \text{if } \mathsf{equal}(\mathsf{rem}(m,n),\mathsf{pred}(n)); \\ \mathsf{div}(m,n) & \text{otherwise.} \end{cases}$$

Another interesting function that turns out to be primitive recursive is $\mathsf{digit}(m,n,p)$, *the m-th least significant digit of the base-p representation of n.* (As an illustration of the use of digit, the predicate $\mathsf{odd}(n)$, with the obvious meaning, can be written simply as $\mathsf{digit}(1,n,2)$.) It is easy to check that $\mathsf{digit}(m,n,p)$ can be defined as $\mathsf{div}(\mathsf{rem}(n,p\uparrow m),p\uparrow(m\sim 1))$.$\diamondsuit$

Example 4.7.4: If $f(n,m)$ is a primitive recursive function, then the *sum*

$$\mathsf{sum}_f(n,m) = f(n,0)+f(n,1)+\cdots+f(n,m)$$

is also primitive recursive, because it can be defined as $\mathsf{sum}_f(n,0) = 0$, and $\mathsf{sum}_f(n,m+1) = \mathsf{sum}_f(n,m)+f(n,m+1)$. We can also define this way the *unbounded conjunctions and disjunctions* of predicates. For example, if $p(n,m)$ is a predicate, the disjunction

$$p(n,0) \text{ or } p(n,1) \text{ or } p(n,2) \text{ or } \cdots \text{ or } p(n,m)$$

is just $\mathsf{sgn}(\mathsf{sum}_p(n,m))$.$\diamondsuit$

Evidently, starting from the extremely simple materials of Definition 4.7.1, we can show that several quite complex functions are primitive recursive. However, primitive recursive functions fail to capture all functions that we should reasonably consider computable. This is best established in terms of a *diagonalization* argument:

Example 4.7.5: The set of primitive recursive functions is *enumerable.* This is because each primitive recursive function can in principle be defined in terms of the basic functions, and therefore can be represented as a string in a finite alphabet; the alphabet should contain symbols for the identity, successor, and zero functions, for primitive recursion and composition, plus parentheses and the symbols 0 and 1 used to index in binary basic functions such as $\mathsf{id}_{17,11}$ (see Section 5.2 for another use of such indexing, this time to represent all Turing machines). We could then enumerate all strings in the alphabet and keep only the ones that are legal definitions of primitive recursive functions —in fact, we could choose to keep only the *unary* primitive recursive functions, those with only one argument.

Suppose then that we list all unary primitive recursive functions, as strings, in lexicographic order

$$f_0, f_1, f_2, f_3, \ldots$$

In principle, given any number $n \geq 0$, we could find the n-th unary primitive recursive function in this list, f_n, and then use its definition to compute the number $f_n(n) + 1$. Call this number $g(n)$. Clearly, $g(n)$ is a computable function —we just outlined how to compute it. Still, g is *not* a primitive recursive function. Because if it were, say $g = f_m$ for some $m \geq 0$, then we would have $f_m(m) = f_m(m) + 1$, which is absurd.

This is a diagonalization argument. It depends on our having a sequential listing of all primitive recursive functions; from that listing one can define a function which differs from all those in the list, and which, therefore, cannot itself be in the list. Compare this argument with the proof of Theorem 1.5.2, stating that $2^{\mathbf{N}}$ is uncountable. There we started with a purported listing of all the members of $2^{\mathbf{N}}$, and obtained a member of $2^{\mathbf{N}}$ not in the listing.$\Diamond$

Evidently, any way of defining functions so that they encompass everything we could reasonably call "computable" cannot be based only on simple operations such as composition and recursive definition, which produce functions that can always and reliably be recognized as such, and therefore enumerated. We have thus discovered an interesting truth about formalisms of computation: Any such formalism whose members (computational devices) are *self-evident* (that is, given a string we can decide easily whether it encodes a computational device in the formalism) must be either too weak (like finite-state automata and primitive recursive functions) or so general as to be useless in practice (like Turing machines that may or may not halt on an input). Any formalism that captures all computable functions, and just these, must include *functions that are not self-evident* (just as it is not self-evident whether a Turing machine halts on all inputs, and thus decides a language). Indeed, we define next a subtler operation on functions, corresponding to the familiar computational primitive of *unbounded iteration* —essentially the while loop. As we shall see, unbounded iteration does introduce the possibility that the result may *not* be a function.

Definition 4.7.2: Let g be a $(k+1)$-ary function, for some $k \geq 0$. The **minimalization** of g is the k-ary function f defined as follows:

$$f(n_1, \ldots, n_k) = \begin{cases} \text{the least } m \text{ such that } g(n_1, \ldots, n_k, m) = 1, \\ \quad \textit{if} \text{ such an } m \text{ exists:} \\ 0 \text{ otherwise.} \end{cases}$$

We shall denote the minimalization of g by $\mu\, m[g(n_1, \ldots, n_k, m) = 1]$.

Although the minimalization of a function g is always well-defined, there is no obvious method for computing it —even if we know how to compute g. The obvious method

$m := 0$;
while $g(n_1, \ldots, n_k, m) \neq 1$ **do** $m := m + 1$;
output m

is not an algorithm, because it may fail to terminate.

Let us then call a function g **minimalizable** if the above method always terminates. That is, a $(k+1)$-ary function g is *minimalizable* if it has the following property: For every $n_1, \ldots, n_k \in \mathbf{N}$, there is an $m \in \mathbf{N}$ such that $g(n_1, \ldots, n_k, m) = 1$.

Finally, call a function **μ-recursive** if it can be obtained from the basic functions by the operations of composition, recursive definition, *and minimalization of minimalizable functions.*

Note that now we cannot repeat the diagonalization argument of example 4.7.5 to show that there is a computable function that is not μ-recursive. The catch is that, given a purported definition of a μ-recursive function, *it is not clear at all whether indeed it defines a μ-recursive function* —that is, whether all applications of minimalization in this definition indeed acted upon minimalizable functions!

Example 4.7.6: We have defined an inverse of addition (the $\sim$ function), and an inverse of multiplication (the div function); but how about the inverse of exponentiation —the *logarithm?* Using minimalization,† we can define the *logarithm* function: $\mathsf{log}(m, n)$ is the smallest power to which we must raise $m+2$ to get an integer at least as big as $n+1$ (that is, $\mathsf{log}(m, n) = \lceil \log_{m+2}(n+1) \rceil$; we have used $m+2$ and $n+1$ as arguments to avoid the mathematical pitfalls in the definition of $\log_m n$ when $m \leq 1$ or $n = 0$). The function log is defined as follows:

$$\mathsf{log}(m, n) = \mu\, p[\mathsf{greater\text{-}than\text{-}or\text{-}equal}((m+2) \uparrow p, n+1)].$$

Note that this is a proper definition of a μ-recursive function, since the function $g(m, n, p) = \mathsf{greater\text{-}than\text{-}or\text{-}equal}((m+2) \uparrow p, n+1)$ is minimalizable: indeed, for any $m, n \geq 0$ there is $p \geq 0$ such that $(m+2)^p \geq n$ —because by raising an integer ≥ 2 to larger and larger powers we can obtain arbitrarily large integers.$\diamondsuit$

We can now prove the main result of this section:

---

† The logarithm function can be defined without the minimalization operation, see Problem 4.7.2. Our use of minimalization here is only for illustration and convenience.

Theorem 4.7.1: *A function $f : \mathbf{N}^k \mapsto \mathbf{N}$ is μ-recursive if and only if it is recursive (that is, computable by a Turing machine).*

Proof: *Only if:* Suppose that f is a μ-recursive function. Then it is defined from the basic functions by applications of composition, recursive definition, and minimalization on minimalizable functions. We shall show that f is Turing computable.

First, it is easy to see that the basic functions are recursive: We have seen this for the successor function (Example 4.2.3), and the remaining functions only involve erasing some or all of the inputs.

So, suppose that $f : \mathbf{N}^k \mapsto \mathbf{N}$ is the composition of the functions $g : \mathbf{N}^\ell \mapsto \mathbf{N}$ and $h_1, \ldots, h_\ell : \mathbf{N}^k \mapsto \mathbf{N}$, where, by induction we know how to compute g and the h_i's. Then we can compute f as follows (in this and the other cases we give programs in the style of random access Turing machine programs that compute these functions; it is fairly easy to see that the same effect can be achieved by standard Turing machines):

$m_1 := h_1(n_1, \ldots, n_k)$;
$m_2 := h_2(n_1, \ldots, n_k)$;
$\vdots$
$m_\ell := h_\ell(n_1, \ldots, n_k)$;
output $g(m_1, \ldots, m_\ell)$.

Similarly, if f is defined recursively from g and h (recall the definition), then $f(n_1, \ldots, n_k, m)$ can be computed by the following program:

$v := g(n_1, \ldots, n_k)$;
if $m = 0$ then output v
else for $i := 1, 2, \ldots, m$ do
  $v := h(n_1, \ldots, n_k, i-1, v)$;
output v.

Finally, suppose that f is defined as $\mu\, m[g(n_1, \ldots, n_k, m)]$, where g is minimalizable and computable. Then f can be computed by the program

$m := 0$;
while $g(n_1, \ldots, n_k, m) \neq 1$ do $m := m + 1$;
output m

Since we are assuming the g is minimalizable, the algorithm above will terminate and output a number.

We have therefore proved that all basic functions are recursive, and that the composition and the recursive definition of recursive functions, and the minimalization of minimalizable recursive functions, are recursive; we must conclude

that all μ-recursive functions are recursive. This completes the *only if* direction of the proof.

If. Suppose that a Turing machine $M = (K, \Sigma, \delta, s, \{h\})$ computes a function $f : \mathbf{N} \mapsto \mathbf{N}$ —we are assuming for simplicity of presentation that f is unary; the general case is an easy extension (see Problem 4.7.5). We shall show that f is μ-recursive. We shall patiently define certain μ-recursive functions pertaining to M and its operation until we have accumulated enough materials to define f itself.

Assume without loss of generality that K and Σ are disjoint. Let $b = |\Sigma| + |K|$, and let us fix a mapping $\mathbf{E}$ from $\Sigma \cup K$ to $\{0, 1, \ldots, b-1\}$, such that $\mathbf{E}(0) = 0$ and $\mathbf{E}(1) = 1$ —recall that, since M computes a numerical function, its alphabet must contain 0 and 1. Using this mapping, we shall represent configurations of M as integers in base-b. The configuration $(q, a_1 a_2 \ldots \underline{a_k} \ldots a_n)$, where the a_i's are symbols in Σ, will be represented as the base-b integer $a_1 a_2 \ldots a_k q a_{k+1} \ldots a_n$, that is, as the integer

$$\mathbf{E}(a_1)b^n + \mathbf{E}(a_2)b^{n-1} + \cdots + \mathbf{E}(a_k)b^{n-k+1} + \mathbf{E}(q)b^{n-k} + \mathbf{E}(a_{k+1})b^{n-k-1} + \cdots + \mathbf{E}(a_n)$$

We are now ready to embark on the definition of f as a μ-recursive function. Ultimately, f will be defined as

$$f(n) = \mathsf{num}(\mathsf{output}(\mathsf{last}(\mathsf{comp}(n)))).$$

num is a function that takes an integer whose base-b representation is a string of 0's and 1's and outputs the binary value of that string. output takes the integer representing in base b a halted configuration of the form $\triangleright \sqcup hw$, and omits the first three symbols $\triangleright \sqcup h$. comp(n) is the number whose representation in base b is the juxtaposition of the unique sequence of configurations that starts with $\triangleright \sqcup sw$, where w is the binary encoding of n, and ends with $\triangleright \sqcup hw'$, where w' is the binary encoding of $f(n)$; such a sequence exists, since we are assuming that M computes a function —namely, f. And last takes an integer representing the juxtaposition of configurations, and extracts the last configuration in the sequence (the part between the last $\triangleright$ and the end).

Of course, we have to define all these functions. We give most of the definitions below, leaving the rest as an exercise (Problem 4.7.4). Let us start with, well, last. We can define lastpos(n), the last (rightmost, least significant) position in the string encoded by n in base b where a $\triangleright$ occurs:

$$\mathsf{lastpos}(n) = \mu\, m[\mathsf{equal}(\mathsf{digit}(m, n, b), \mathbf{E}(\triangleright)) \text{ or } \mathsf{equal}(m, n)].$$

Notice that, in order to make the function within the brackets minimalizable, we allowed lastpos(n) to be n if no $\triangleright$ is found in n. Incidentally, this is another

superficial use of minimalization, as this function can be easily redefined without the use of minimalization. We can then define $\mathsf{last}(n)$ as $\mathsf{rem}(n, b \uparrow \mathsf{lastpos}(n))$. We could also define $\mathsf{rest}(n)$, the sequence that remains after the deletion of $\mathsf{last}(n)$, as $\mathsf{div}(n, b \uparrow \mathsf{lastpos}(n))$.

$\mathsf{output}(n)$ is simply $\mathsf{rem}(n, b \uparrow \mathsf{log}(b \sim 2, n \sim 1) \sim 2)$ —recall our convention that the arguments of log must be decreased by 2 and 1, respectively.

The function $\mathsf{num}(n)$ can be written as the sum

$$\mathsf{digit}(1, n, b) \cdot 2 + \mathsf{digit}(2, n, b) \cdot 2 \uparrow 2 + \cdots + \mathsf{digit}(\log(b \sim 2, n \sim 1), n, b) \cdot 2 \uparrow \log(b \sim 2, n \sim 1).$$

This is a μ-recursive function since both the summand and the bound $\log(b \sim 2, n \sim 1)$ are. Its inverse function, $\mathsf{bin}(n)$, which maps any integer to the string that is its binary encoding, encoded again in as a base-b integer, is very similar, with the rôles of 2 and b reversed.

The most interesting (and hard to define) function in the definition of $f(n)$ above is $\mathsf{comp}(n)$, which maps n to the sequence of configurations of M that carries out the computation of $f(n)$ —in fact, the base-b integer that encodes this sequence. At the highest level, it is just

$$\mathsf{comp}(n) = \mu\, m[\mathsf{iscomp}(m, n) \text{ and } \mathsf{halted}(\mathsf{last}(m))], \tag{1}$$

where $\mathsf{iscomp}(m, n)$ is a predicate stating that m is the sequence of configurations in a computation, not necessarily halted, starting from $\triangleright \underline{s}\mathbf{b}(n)$. (Incidentally, this is *the only place in this proof in which minimalization is truly, inherently needed.*) Notice that the function within the brackets in (1) is indeed minimalizable: Since M is assumed to compute f, such a sequence m of configurations will exist for all n. $\mathsf{halted}(n)$ is simply $\mathsf{equal}(\mathsf{digit}(\mathsf{log}(b - 2, n \sim 1) \sim 2, n, b), \mathbf{E}(h))$.

We leave the precise definition of iscomp as an exercise (Problem 4.7.4).

It follows that f is indeed a μ-recursive function, and the proof of the theorem has been completed. ■

Problems for Section 4.7

4.7.1. Let $f : \mathbf{N} \mapsto \mathbf{N}$ be a primitive recursive function, and define $F : \mathbf{N} \mapsto \mathbf{N}$ by

$$F(n) = f(f(f(\ldots f(n)\ldots))),$$

where there are n function compositions. Show that F is primitive recursive.

4.7.2. Show that the following functions are primitive recursive:

(a) $\mathsf{factorial}(n) = n!$.
(b) $\mathsf{gcd}(m, n)$, the greatest common divisor of m and n.
(c) $\mathsf{prime(n)}$, the predicate that is 1 if n is a prime number.
(d) $\mathsf{p(n)}$, the nth prime number, where $\mathsf{p(0)} = 2$, $\mathsf{p(1)} = 3$, and so on.
(e) The function log defined in the text.

4.7.3. Suppose that f is a μ-recursive bijection from $\mathbf{N}$ to $\mathbf{N}$. Show that its inverse, f^{-1}, is also μ-recursive.

4.7.4. Show that the function iscomp described in the proof of Theorem 4.7.1 is primitive recursive.

4.7.5. Which modifications must be made to the construction in the proof of the *if* directions of Theorem 4.7.1 if M computes a function $f : \mathbf{N}^k \mapsto \mathbf{N}$ with $k > 1$?

4.7.6. Develop a representation of primitive recursive functions as strings in an alphabet Σ of your choice (see the next chapter for such a representation of Turing machines). Formalize the argument in Example 4.7.5 that not all computable functions can be primitive recursive.

REFERENCES

Turing machines were first conceived by Alan M. Turing:

- A. M. Turing "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings, London Mathematical Society, 2,* 42 pp. 230-265, and no. 43, pp. 544–546, 1936.

Turing introduced this model in order to argue that all detailed sets of instructions that can be carried out by a human calculator can also be carried out by a suitably defined simple machine. For the record, Turing's original machine has one two-way infinite tape and one head (see Section 4.5). A similar model was independently conceived by Post; see

- E. L. Post "Finite Combinatory Processes. Formulation I," *Journal of Symbolic Logic, 1,* pp. 103–105, 1936.

The following books contain interesting introductions to Turing machines:

- M. L. Minsky *Computation: Finite and Infinite Machines,* Englewood Cliffs, N.J.: Prentice-Hall, 1967.
- F. C. Hennie *Introduction to Computability,* Reading, Mass.: Addison-Wesley, 1977.

The following are other advanced books on Turing machines and related concepts introduced in this and the three subsequent chapters:

- M. Davis, ed., *The Undecidable,* Hewlett, N.Y.: Raven Press, 1965. (This book contains many original articles on several aspects of the subject, including the papers of Turing and Post cited above.)
- M. Davis, ed., *Computability and Unsolvability* New York: McGraw-Hill, 1958.
- S. C. Kleene, *Introduction to Metamathematics,* Princeton, N.J.: D. Van Nostrand, 1952,
- W. S. Brainerd and L. H. Landweber, *Theory of Computation,* New York: John Wiley, 1974,

- M. Machtey and P. R. Young, *An Introduction to the General Theory of Algorithms,* New York: Elsevier North-Holland, 1978,
- H. Rogers, Jr., *The Theory of Recursive Functions and Effective Computability,* New York: McGraw-Hill, 1967,
- M. Sipser, *Introduction to the Theory of Computation,* Boston, Mass.: PWS Publishers, 1996,
- J. E. Hopcroft and J. D. Ullman *Introduction to Automata Theory, Languages, and Computation*, Reading, Mass.: Addison Wesley, 1979.
- C. H. Papadimitriou *Computational Complexity*, Reading, Mass.: Addison Wesley, 1994.
- H. Hermes, *Enumerability, Decidability, Computability,* New York: Springer Verlag, 1969 (translated from the German edition, 1965).

Our notion and notation for combining Turing machines (Section 4.3) was influenced by this last book.

Random access machines, similar in spirit to our "random access Turing machines" in Section 2.4, were studied in

- S. A. Cook and R. A. Reckhow "Time-bounded random-access machines," *Journal of Computer and Systems Sciences, 7,* 4, pp. 354–375, 1973.

Primitive and μ-recursive functions are due to Kleene

- S. C. Kleene "General recursive functions of natural numbers," *Mathematische Annalen, 112,* pp. 727–742, 1936,

and Markov Algorithms (Problem 2.6.5) are from

- A. A. Markov *Theory of Algorithms,* Trudy Math. Inst. V. A.Steklova, 1954. English translation: Israel Program for Scientific Translations, Jerusalem, 1961.