CHAPTER 5

# Computability

The Turing machine (TM) is believed to be the most general computational model that can be devised (the **Church-Turing thesis**). Despite many attempts, no computational model has yet been introduced that can perform computations impossible on a Turing machine. This is not a statement about efficiency; other machines, notably the RAM of Section 3.4, can do the same computations either more quickly or with less memory. Instead, it is a statement about the feasibility of computational tasks. If a task can be done on a Turing machine, it is considered feasible; if it cannot, it is considered infeasible. Thus, the TM is a litmus test for computational feasibility. As we show later, however, there are some well-defined tasks that cannot be done on a TM.

The chapter opens with a formal definition of the standard Turing machine and describes how the Turing machine can be used to compute functions and accept languages. We then examine multi-tape and nondeterministic TMs and show their equivalence to the standard model. The nondeterministic TM plays an important role in Chapter 8 in the classification of languages by their complexity. The equivalence of phrase-structure languages and the languages accepted by TMs is then established. The universal Turing machine is defined and used to explore limits on language acceptance by Turing machines. We show that some languages cannot be accepted by any Turing machine, while others can be accepted but not by Turing machines that halt on all inputs (the languages are unsolvable). This sets the stage for a proof that some problems, such as the Halting Problem, are unsolvable; that is, there is no Turing machine halting on all inputs that can decide for an arbitrary Turing machine $M$ and input string $\boldsymbol{w}$ whether or not $M$ will halt on $\boldsymbol{w}$. We close by defining the *partial recursive functions*, the most general functions computable by Turing machines.

## 5.1 The Standard Turing Machine Model

The standard Turing machine consists of a control unit, which is a finite-state machine, and a (single-ended) infinite-capacity tape unit. (See Fig. 5.1.) Each cell of the tape unit initially contains the blank symbol $\beta$. A string of symbols from the tape alphabet $\Gamma$ is written left-adjusted on the tape and the tape head is placed over the first cell. The control unit then reads the symbol under the head and makes a state transition the result of which is either to write a new symbol under the tape head or to move the head left (if possible) or right. (The TM described in Section 3.7 is slightly different; it always replaces the cell contents and always issues a move command, even if the effect in both cases is null. The equivalence between the standard TM and that described in Section 3.7 is easily established. See Problem 5.1.) A move left from the first cell leads to **abnormal termination**, a problem that can be avoided by having the Turing machine write a special **end-of-tape marker** in the first tape cell. This marker is a tape symbol not used elsewhere.

**DEFINITION 5.1.1** *A* **standard Turing machine (TM)** *is a six-tuple* $M = (\Gamma, \beta, Q, \delta, s, h)$ *where* $\Gamma$ *is the* **tape alphabet** *not containing the* **blank symbol** $\beta$, $Q$ *is the finite* **set of states**, $\delta : Q \times (\Gamma \cup \{\beta\}) \mapsto (Q \cup \{h\}) \times (\Gamma \cup \{\beta\} \cup \{\mathbf{L}, \mathbf{R}\})$ *is the* **next-state function**, $s$ *is the* **initial state**, *and* $h \notin Q$ *is the* **accepting halt state**. *A TM cannot exit from* $h$. *If* $M$ *is in state* $q$ *with letter* $a$ *under the tape head and* $\delta(q, a) = (q', \mathbf{C})$, *its control unit enters state* $q'$ *and writes* $a'$ *if* $\mathbf{C} = a' \in \Gamma \cup \{\beta\}$ *or moves the head left (if possible) or right if* $\mathbf{C}$ *is* **L** *or* **R**, *respectively.*

*The TM* $M$ **accepts the input string** $\boldsymbol{w} \in \Gamma^*$ *(it contains no blanks) if when started in state* $s$ *with* $\boldsymbol{w}$ *placed left-adjusted on its otherwise blank tape and the tape head at the leftmost tape cell, the last state entered by* $M$ *is* $h$. $M$ **accepts the language** $L(M)$ *consisting of all strings accepted by* $M$. *Languages accepted by Turing machines are called* **recursively enumerable**. *A language* $L$ *is* **decidable** *or* **recursive** *if there exists a TM* $M$ *that halts on every input string, whether in* $L$ *or not, and accepts exactly the strings in* $L$.

*A function* $f : \Gamma^* \mapsto \Gamma^* \cup \{\perp\}$, *where* $\perp$ *is a symbol that is not in* $\Gamma$, *is* **partial** *if for some* $\boldsymbol{w} \in \Gamma^*$, $f(\boldsymbol{w}) = \perp$ *(*$f$ *is* **not defined** *on* $\boldsymbol{w}$*). Otherwise,* $f$ *is* **total**.

*A TM* $M$ **computes a function** $f : \Gamma^* \mapsto \Gamma^* \cup \perp$ *for those* $\boldsymbol{w}$ *such that* $f(\boldsymbol{w})$ *is defined if when started in state* $s$ *with* $\boldsymbol{w}$ *placed left-adjusted on its otherwise blank tape and the tape head at the leftmost tape cell,* $M$ *enters the accepting halt state* $h$ *with* $f(\boldsymbol{w})$ *written left-adjusted on its otherwise blank tape. If a TM halts on all inputs, it implements an* **algorithm**. *A task defined by a total function* $f$ *is* **solvable** *if* $f$ *has an algorithm and* **unsolvable** *otherwise.*

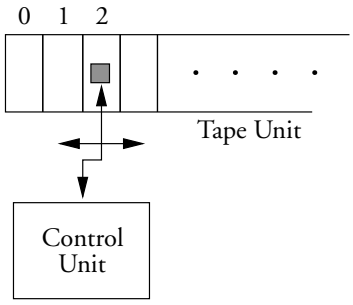


**Figure 5.1** The control and tape units of the standard Turing machine.

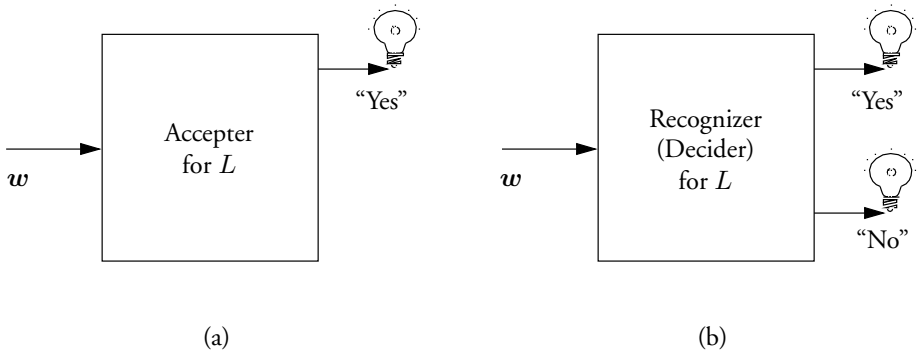


**Figure 5.2** An accepter (a) for a language $L$ is a Turing machine that can accept strings in a language $L$ but may not halt on all inputs. A decider or recognizer (b) for a language $L$ is a Turing machine that halts on all inputs and accepts strings in $L$.

The accepting halt state $h$ has been singled out to emphasize language acceptance. However, there is nothing to prevent a TM from having multiple halt states, states from which it does not exit. (A halt state can be realized by a state to which a TM returns on every input without moving the tape head or changing the value under the head.) On the other hand, on some inputs a TM may never halt. For example, it may endlessly move its tape head right one cell and write the symbol $a$.

Notice that we do not require a TM $M$ to halt on every input string for it to accept a language $L(M)$. It need only halt on those strings in the language. A language $L$ for which there is a TM $M$ accepting $L = L(M)$ that halts on all inputs is decidable. The distinction between accepting and recognizing (or deciding) a language $L$ is illustrated schematically in Fig. 5.2. An accepter is a TM that accepts strings in $L$ but may not halt on strings not in $L$. When the accepter determines that the string $\boldsymbol{w}$ is in the language $L$, it turns on the "Yes" light. If this light is not turned on, it may be that the string is not in $L$ or that the TM is just slow. On the other hand, a recognizer or decider is a TM that halts on all inputs and accepts strings in $L$. The "Yes" or "No" light is guaranteed to be turned on at some time.

The computing power of the TM is extended by allowing **partial computations**, computations on which the TM does not halt on every input. The computation of functions by Turing machines is discussed in Section 5.9.

### 5.1.1 Programming the Turing Machine

Programming a Turing machine means choosing a tape alphabet and designing its control unit, a finite-state machine. Since the FSM has been extensively studied elsewhere, we limit our discussion of programming of Turing machines to four examples, each of which illustrates a fundamental point about Turing machines. Although TMs are generally designed to perform unbounded computations, their control units have a bounded number of states. Thus, we must insure that as they move across their tapes they do not accumulate an unbounded amount of information.

A simple example of a TM is one that moves right until it encounters a blank, whereupon it halts. The TM of Fig. 5.3(a) performs this task. If the symbol under the head is 0 or 1,

| $q$ | $a$ | $\delta(\sigma, q)$ | |
|---|---|---|---|
| $q_1$ | 0 | $q_1$ | **R** |
| $q_1$ | 1 | $q_1$ | **R** |
| $q_1$ | $\beta$ | $h$ | $\beta$ |

(a)

| $q$ | $a$ | $\delta(\sigma, q)$ | |
|---|---|---|---|
| $q_1$ | 0 | $q_2$ | $\beta$ |
| $q_1$ | 1 | $q_3$ | $\beta$ |
| $q_1$ | $\beta$ | $h$ | $\beta$ |
| $q_2$ | 0 | $q_4$ | **R** |
| $q_2$ | 1 | $q_4$ | **R** |
| $q_2$ | $\beta$ | $q_4$ | **R** |
| $q_3$ | 0 | $q_5$ | **R** |
| $q_3$ | 1 | $q_5$ | **R** |
| $q_3$ | $\beta$ | $q_5$ | **R** |
| $q_4$ | 0 | $q_2$ | 0 |
| $q_4$ | 1 | $q_3$ | 0 |
| $q_4$ | $\beta$ | $h$ | 0 |
| $q_5$ | 0 | $q_2$ | 1 |
| $q_5$ | 1 | $q_3$ | 1 |
| $q_5$ | $\beta$ | $h$ | 1 |

(b)

**Figure 5.3** The transition functions of two Turing machines, one (a) that moves across the non-blank symbols on its tape and halts over the first blank symbol, and a second (b) that moves the input string right one position and inserts a blank to its left.

it moves right. If it is the blank symbol, it halts. This TM can be extended to replace the rightmost character in a string of non-blank characters with a blank. After finding the blank on the right of a non-blank string, it backs up one cell and replaces the character with a blank. Both TMs compute functions that map strings to strings.

A second example is a TM that replaces the first letter in its input string with a blank and shifts the remaining letters right one position. (See Fig. 5.3(b).) In its initial state $q_1$ this TM, which is assumed to be given a non-blank input string, records the symbol under the tape head by entering $q_2$ if the letter is 0 or $q_3$ if the letter is 1 and writing the blank symbol. In its current state it moves right and enters a corresponding state. (It enters $q_4$ if its current state is $q_2$ and $q_5$ if it is $q_3$.) In the new state it prints the letter originally in the cell to its left and enters either $q_2$ or $q_3$ depending on whether the current cell contains 0 or 1. This TM can be used to insert a special end-of-tape marker instead of a blank to the left of a string written initially on a tape. This idea can generalized to insert a symbol anyplace in another string.

A third example of a TM $M$ is one that accepts strings in the language $L = \{a^n b^n c^n \mid n \geq 1\}$. $M$ inserts an end-of-tape marker to the left of a string $\boldsymbol{w}$ placed on its tape and uses a computation denoted $C(x, y)$, in which it moves right across zero or more $x$'s followed by zero or more "pseudo-blanks" (a symbol other than $a$, $b$, $c$, or $\beta$) to an instance of $y$, entering a non-accepting halt state $f$ if some other pattern of letters is found. Starting in the first cell, if $M$ discovers that the next letter is not $a$, it exits to state $f$. If it is $a$, it replaces $a$ by a pseudo-blank. It then executes $C(a, b)$. $M$ then replaces $b$ by a pseudo-blank and executes $C(b, c)$, after which it replaces $c$ by a pseudo-blank and executes $C(c, \beta)$. It then returns to the beginning of the tape. If it arrives at the end-of-tape marker without encountering any

instances of $a$, $b$, or $c$, it terminates in the accepting halt state $h$. If not, then it moves right over pseudo-blanks until it finds an $a$, entering state $f$ if it finds some other letter. It then resumes the process executed on the first pass by invoking $C(a, b)$. This computation either enters the non-accepting halt state $f$ or on each pass it replaces one instance each of $a$, $b$, and $c$ with a pseudo-blank. Thus, $M$ accepts the language $L = \{a^n b^n c^n \,|\, n \geq 1\}$; that is, $L$ is decidable (recursive). Since $M$ makes one pass over the tape for each instance of $a$, it uses time $O(n^2)$ on a string of length $n$. Later we give examples of languages that are recursively enumerable but not recursive.

In Section 3.8 we reasoned that any RAM computation can be simulated by a Turing machine. We showed that any program written for the RAM can be executed on a Turing machine at the expense of an increase in the running time from $T$ steps on a RAM with $S$ bits of storage to a time $O(ST \log^2 S)$ on the Turing machine.

## 5.2 Extensions to the Standard Turing Machine Model

In this section we examine various extensions to the standard Turing machine model and establish their equivalence to the standard model. These extensions include the multi-tape, nondeterministic, and oracle Turing machines.

We first consider the **double-ended tape Turing machine**. Unlike the standard TM that has a tape bounded on one end, this is a TM whose single tape is double-ended. A TM of this kind can be simulated by a two-track one-tape TM by reading and writing data on the top track when working on cells to the right of the midpoint of the tape and reading and writing data on the bottom track when working with cells to its left. (See Problem 5.7.)

### 5.2.1 Multi-Tape Turing Machines

A **$k$-tape Turing machine** has a control unit and $k$ single-ended tapes of the kind shown in Fig. 5.1. Each tape has its own head and operates in the fashion indicated for the standard model. The FSM control unit accepts inputs from all tapes simultaneously, makes a state transition based on this data, and then supplies outputs to each tape in the form of either a letter to be written under its head or a head movement command. We assume that the tape alphabet of each tape is $\Gamma$. A three-tape TM is shown in Fig. 5.4. A $k$-tape TM $M_k$ can be simulated by a one-tape TM $M_1$, as we now show.

**THEOREM 5.2.1** *For each $k$-tape Turing machine $M_k$ there is a one-tape Turing machine $M_1$ such that a terminating $T$-step computation by $M_k$ can be simulated in $O(T^2)$ steps by $M_1$.*

**Proof** Let $\Gamma$ and $\Gamma'$ be the tape alphabets of $M_k$ and $M_1$, respectively. Let $|\Gamma'| = (2|\Gamma|)^k$ so that $\Gamma'$ has enough letters to allow the tape of $M_1$ to be subdivided into $k$ tracks, as suggested in Fig. 5.5. Each cell of a track contains $2|\Gamma|$ letters, a number large enough to allow each cell to contain either a member of $\Gamma$ or a marked member of $\Gamma$. The marked members retain their original identity but also contain the information that they have been marked. As suggested in Fig. 5.5 for a three-tape TM, $k$ heads can be simulated by one head by marking the positions of the $k$ heads on the tracks of $M_1$.

$M_1$ simulates $M_k$ in two passes. First it visits marked cells to collect the letters under the original tape heads, after which it makes a state transition akin to that made by $M_k$. In a second pass it visits the marked cells either to change their entries or to move the simulated

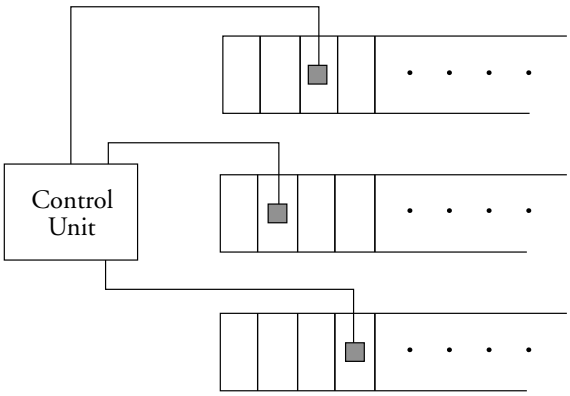


**Figure 5.4** A three-tape Turing machine.

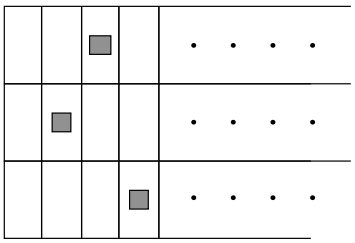

**Figure 5.5** A single tape of a TM with a large tape alphabet that simulates a three-tape TM with a smaller tape alphabet.

tape heads. If the $k$-tape TM executes $T$ steps, it uses at most $T + 1$ tape cells. Thus each pass requires $O(T)$ steps and the complete computation can be done in $O(T^2)$ steps. ■

Multi-tape machines in which the tapes are double-ended are equivalent to multi-tape single-ended Turing machines, as the reader can show.

## 5.2.2 Nondeterministic Turing Machines

The **nondeterministic standard Turing machine** (NDTM) is introduced in Section 3.7.1. We use a slightly altered definition that conforms to the definition of the standard Turing machine in Definition 5.1.1.

**DEFINITION 5.2.1** *A* **nondeterministic Turing machine (NDTM)** *is a seven-tuple* $M = (\Sigma, \Gamma, \beta, Q, \delta, s, h)$ *where* $\Sigma$ *is the* **choice input alphabet**, $\Gamma$ *is the* **tape alphabet** *not containing the* **blank symbol** $\beta$, $Q$ *is the finite* **set of states**, $\delta : Q \times \Sigma \times (\Gamma \cup \{\beta\}) \mapsto (Q \cup \{h\}) \times (\Gamma \cup \{\beta\} \cup \{\mathbf{L}, \mathbf{R}\}) \cup \{\perp\}$ *is the* **next-state function**, $s$ *is the* **initial state**, *and* $h \notin Q$ *is the* **accepting halt state**. *A TM cannot exit from* $h$. *If* $M$ *is in state* $q$ *with letter* $a$ *under the tape head and* $\delta(q, c, a) = (q', \mathbf{C})$, *its control unit enters state* $q'$ *and writes* $a'$ *if*

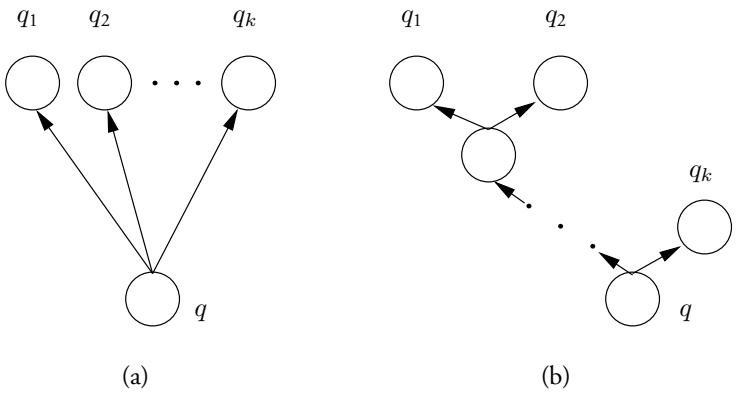


**Figure 5.6** The construction used to reduce the fan-out of a nondeterministic state.

$\mathbf{C} = a' \in \Gamma \cup \{\beta\}$, *or it moves the head left (if possible) or right if* $\mathbf{C}$ *is* $\mathbf{L}$ *or* $\mathbf{R}$, *respectively. If* $\delta(q, c, a) = \perp$, *there is no successor to the current state with choice input* $c$ *and tape symbol* $a$.

*An NDTM* $M$ *reads one character of its* **choice input string** $\boldsymbol{c} \in \Sigma^*$ *on each step. An NDTM* $M$ **accepts string** $\boldsymbol{w}$ *if there is some choice string* $\boldsymbol{c}$ *such that the last state entered by* $M$ *is* $h$ *when* $M$ *is started in state* $s$ *with* $\boldsymbol{w}$ *placed left-adjusted on its otherwise blank tape, and the tape head at the leftmost tape cell. An NDTM* $M$ **accepts the language** $L(M) \subseteq \Gamma^*$ *consisting of those strings* $\boldsymbol{w}$ *that it accepts. Thus, if* $\boldsymbol{w} \notin L(M)$, *there is no choice input for which* $M$ *accepts* $\boldsymbol{w}$.

If an NDTM has more than two nondeterministic choices for a particular state and letter under the tape head, we can design another NDTM that has at most two choices. As suggested in Fig. 5.6, for each state $q$ that has $k$ possible next states $q_1, \ldots, q_k$ for some input letter, we can add $k - 2$ intermediate states, each with two outgoing edges such that a) in each state the tape head doesn't move and no change is made in the letter under the head, but b) each state has the same $k$ possible successor states. It follows that the new machine computes the same function or accepts the same language as the original machine. Consequently, from this point on we assume that there are either one or two next states from each state of an NDTM for each tape symbol.

We now show that the range of computations that can be performed by deterministic and nondeterministic Turing machines is the same. However, this does not mean that with the identical resource bounds they compute the same set of functions.

**THEOREM 5.2.2** *Any language accepted by a nondeterministic standard TM can be accepted by a standard deterministic one.*

**Proof** The proof is by simulation. We simulate all possible computations of a nondeterministic standard TM $M_{\mathrm{ND}}$ on an input string $\boldsymbol{w}$ by a deterministic three-tape TM $M_{\mathrm{D}}$ and halt if we find a sequence of moves by $M_{\mathrm{ND}}$ that leads to an accepting halt state. Later this machine can be simulated by a one-tape TM. The three tapes of $M_{\mathrm{D}}$ are an input tape, a work tape, and **enumeration tape**. (See Fig. 5.7.) The input tape holds the input and is never modified. The work tape is used to simulate $M_{\mathrm{ND}}$. The enumeration tape contains choice sequences used by $M_{\mathrm{D}}$ to decide which move to make when simu-

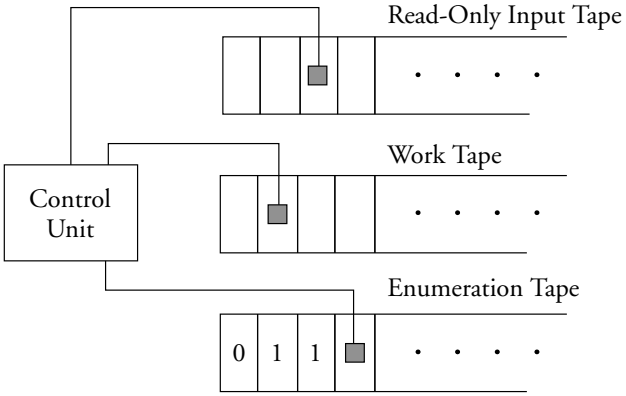


**Figure 5.7** A three-tape deterministic Turing machine that simulates a nondeterministic Turing machine.

lating $M_{\mathrm{ND}}$. These sequences are generated in lexicographical order, that is, in the order $0, 1, 00, 01, 10, 11, 000, 001, \ldots$. It is straightforward to design a deterministic TM that generates these sequences. (See Problem 5.2.)

Breadth-first search is used. Since a string $\boldsymbol{w}$ is accepted by a nondeterministic TM if there is some choice input on which it is accepted, a deterministic TM $M_{\mathrm{D}}$ that accepts the input $\boldsymbol{w}$ accepted by $M_{\mathrm{ND}}$ can be constructed by erasing the work tape, copying the input sequence $\boldsymbol{w}$ to the work tape, placing the next choice input sequence in lexicographical order on the enumeration tape (initially this is the sequence 0), and then simulating $M_{\mathrm{ND}}$ on the work tape while reading one choice input from the enumeration tape on each step. If $M_{\mathrm{D}}$ runs out of choice inputs before reaching the halt state, the above procedure is restarted with the next choice input sequence. This method deterministically accepts the input string $\boldsymbol{w}$ if and only if there is some choice input to $M_{\mathrm{ND}}$ on which it is accepted. ■

Adding more than one tape to a nondeterministic Turing machine does not increase its computing power. To see this, it suffices to simulate a multi-tape nondeterministic Turing machine with a single-tape one, using a construction parallel to that of Theorem 5.2.1, and then invoke the above result. Applying these observations to language acceptance yields the following corollary.

**COROLLARY 5.2.1** *Any language accepted by a nondeterministic (multi-tape) Turing machine can be accepted by a deterministic standard Turing machine.*

We emphasize that this result does not mean that with identical resource bounds the deterministic and nondeterministic Turing machines compute the same set of functions.

### 5.2.3 Oracle Turing Machines

The **oracle Turing machine** (OTM) is a multi-tape TM or NDTM with a special **oracle tape** and an associated **oracle** function $h : \mathcal{B}^* \mapsto \mathcal{B}^*$, which need not be computable. (See Fig. 5.8.) After writing a string $\boldsymbol{z}$ on its oracle tape, the OTM signals to the oracle to replace $\boldsymbol{z}$ with the value $h(\boldsymbol{z})$ of the oracle function. During a computation the OTM may consult

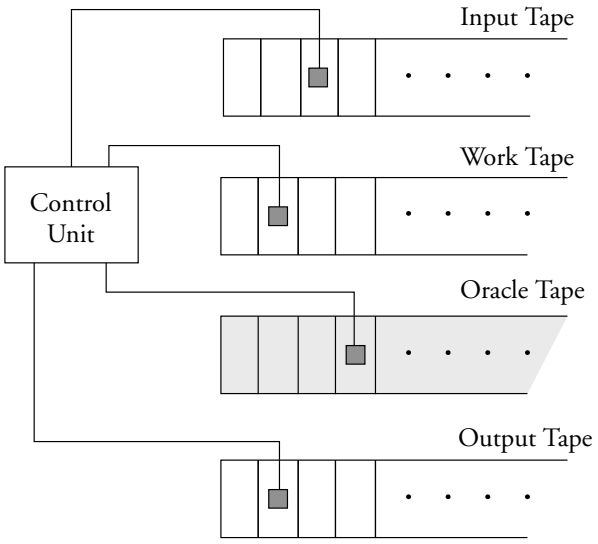


**Figure 5.8** The oracle Turing machine has an "oracle tape" on which it writes a string (a problem instance), after which an "oracle" returns an answer in one step.

the oracle as many times as it wishes. **Time** on an OTM is the number of steps taken, where one consultation of the oracle is counted as one step. **Space** is the number of cells used on the work tapes of an OTM not including the oracle tape. The OTM machine can be used to classify problems. (See Problem 8.15.)

### 5.2.4 Representing Restricted Models of Computation

Now that we have introduced a variety of Turing machine models, we ask how the finite-state machine and pushdown automaton fit into the picture.

The finite-state machine can be viewed as a Turing machine with two tapes, the first a read-only input tape and the second a write-only output tape. This TM reads consecutive symbols on its input tape, moving right after reading each symbol, and writes outputs on its output tape, moving right after writing each symbol. If this TM enters an accepting halt state, the input sequence read from the tape is accepted.

The pushdown automaton can be viewed as a Turing machine with two tapes, a read-only input tape and a pushdown tape. The pushdown tape is a standard tape that pushes a new symbol by moving its head right one cell and writing the new symbol into this previously blank cell. It pops the symbol at the top of the stack by copying the symbol, after which it replaces it with the blank symbol and moves its head left one cell.

The Turing machine can be simulated by two pushdown tapes. The movement of the head in one direction can be simulated by popping the top item of one stack and pushing it onto the other stack. To simulate the movement of the head in the opposite direction, interchange the names of the two stacks.

The nondeterministic equivalents of the finite-state machine and pushdown automaton are obtained by making their Turing machine control units nondeterministic.

## 5.3 Configuration Graphs

We now introduce configuration graphs, graphs that capture the state of Turing machines with potentially unlimited storage capacity. We begin by describing configuration graphs for one-tape Turing machines.

**DEFINITION 5.3.1** *The* **configuration of a standard Turing machine** $M$ *at any point in time is* $[x_1x_2\ldots\mathbf{p}x_j\ldots x_n]$, *where* $p$ *is the state of the control unit, the tape head is over the* $j$*th tape cell, and* $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ *is the string that contains all the non-blank symbols on the tape as well as the symbol under the head. Here the state* $p$ *is shown in boldface to the left of the symbol* $x_j$ *to indicate that the tape head is over the* $j$*th cell.* $x_n$ *and some of the symbols to its left may be blanks.*

To illustrate such configurations, consider a TM $M$ that is in state $p$ reading the third symbol on its tape, which contains $xyz$. This information is captured by the configuration $[xy\mathbf{p}z]$. If $M$ changes to state $q$ and moves its head right, then its new configuration is $[xyz\mathbf{q}\beta]$. In this case we add a blank $\beta$ to the right of the string $xyz$ to insure that the head resides over the string.

Because multi-tape TMs are important in classifying problems by their use of temporary work space, a definition for the configuration of a multi-tape TM is desirable. We now introduce a notation for this purpose that is somewhat more cumbersome than used for the standard TM. This notation uses an explicit binary number for the position of each tape head.

**DEFINITION 5.3.2** *The* **configuration of a** $k$**-tape Turing machine** $M$ *is* $(p, \boldsymbol{h_1}, \boldsymbol{h_2}, \ldots, \boldsymbol{h_k}, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$, *where* $\boldsymbol{h_r}$ *is the position of the head in binary on the* $r$*th tape,* $p$ *is the state of the control unit, and* $\boldsymbol{x}_r$ *is the string on the* $r$*th tape that includes all the non-blank symbols as well as the symbol under the head.*

We now define configuration graphs for deterministic TMs and NDTMs. Because we will apply configuration graphs to machines that halt on all inputs, we view them as acyclic.

**DEFINITION 5.3.3** *A* **configuration graph** $G(M_{\mathrm{ND}}, \boldsymbol{w})$ *associated with the NDTM* $M_{\mathrm{ND}}$ *is a directed graph whose vertices are configurations of* $M_{\mathrm{ND}}$. *(See Fig. 5.9.) There is a directed edge between two vertices if for some choice input vector* $\boldsymbol{c}$ $M_{\mathrm{ND}}$ *can move from the first configuration to*

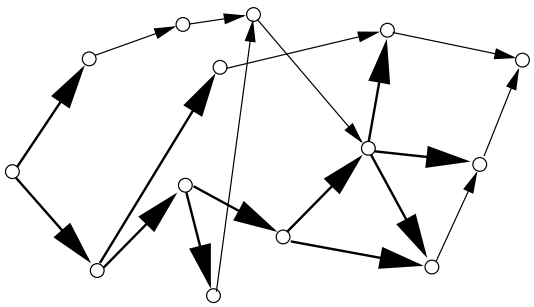

**Figure 5.9** The configuration graph $G(M_{\mathrm{ND}}, \boldsymbol{w})$ of a nondeterministic Turing machine $M_{\mathrm{ND}}$ on input $\boldsymbol{w}$ has one vertex for each configuration of $M_{\mathrm{ND}}$. The graph is acyclic. Heavy edges identify the nondeterministic choices associated with each configuration.

*the second in one step. There is one configuration corresponding to the initial state of the machine and one corresponding to the final state. (We assume without loss of generality that, after accepting an input string, $M_{\text{ND}}$ enters a cleanup phase during which it places a fixed string on each tape.)*

Configuration graphs are used in the next section to associate a phrase-structure language with a Turing machine. They are also used in many places in Chapter 8, especially in Section 8.5.3, where they are used to establish an important relationship between deterministic and nondeterministic space classes.

## 5.4 Phrase-Structure Languages and Turing Machines

We now demonstrate that the phrase-structure languages and the languages accepted by Turing machines are the same. We begin by showing that every recursively enumerable language is a phrase-structure language. For this purpose we use configurations of one-tape Turing machines. Then, for each phrase-structure language $L$ we describe the construction of a TM accepting $L$. We conclude that the languages accepted by TMs and described by phrase-structure grammars are the same.

With these conventions as background, if a standard TM halts in its accepting halt state, we can require that it halt with $\beta 1 \beta$ on its tape when it accepts the input string $\boldsymbol{w}$. Thus, the TM configuration when a TM halts and accepts its input string is $[\mathbf{h}\beta 1\beta]$. Its starting configuration is $[\mathbf{s}\beta w_1 w_2 \ldots w_n \beta]$, where $\boldsymbol{w} = w_1 w_2 \ldots w_n$.

**THEOREM 5.4.1** *Every recursively enumerable language is a phrase-structure language.*

**Proof** Let $M = (\Gamma, \beta, Q, \delta, s, h)$ be a deterministic TM and let $L(M)$ be the recursively enumerable language over the alphabet $\Gamma$ that it accepts. The goal is to show the existence of a phrase-structure grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ that can generate each string $\boldsymbol{w}$ of $L$, and no others. Since the TM accepting $L$ halts with $\beta 1 \beta$ on its tape when started with $\boldsymbol{w} \in L$, we design a grammar $G$ that produces the configurations of $M$ in reverse order. Starting with the final configuration $[\mathbf{h}\beta 1\beta]$, $G$ produces the starting configuration $[\mathbf{s}\beta w_1 w_2 \ldots w_n \beta]$, where $\boldsymbol{w} = w_1 w_2 \ldots w_n$, after which it strips off the characters $[\mathbf{s}\beta$ at the beginning and $\beta]$. The grammar $G$ defined below serves this purpose, as we show.

Let $\mathcal{N} = Q \cup \{\text{S}, \beta, [, ]\}$ and $\mathcal{T} = \Gamma$. The rules $\mathcal{R}$ of $G$ are defined as follows:

| | | | | |
|---|---|---|---|---|
| (a) | S | $\rightarrow$ | $[\mathbf{h}\beta 1\beta]$ | |
| (b) | $\beta]$ | $\rightarrow$ | $\beta\beta]$ | |
| (c) | $[\mathbf{s}\beta$ | $\rightarrow$ | $\epsilon$ | |
| (d) | $\beta\beta]$ | $\rightarrow$ | $\beta]$ | |
| (e) | $\beta]$ | $\rightarrow$ | $\epsilon$ | |
| (f) | $x\mathbf{q}$ | $\rightarrow$ | $\mathbf{p}x$ | for all $p \in Q$ and $x \in (\Gamma \cup \{\beta\})$ such that $\delta(p, x) = (q, \mathbf{R})$ |
| (g) | $\mathbf{q}zx$ | $\rightarrow$ | $z\mathbf{p}x$ | for all $p \in Q$ and $x, z \in (\Gamma \cup \{\beta\})$ such that $\delta(p, x) = (q, \mathbf{L})$ |
| (h) | $\mathbf{q}y$ | $\rightarrow$ | $\mathbf{p}x$ | for all $p \in Q$ and $x \in (\Gamma \cup \{\beta\})$ such that $\delta(p, x) = (q, y)$, $y \in (\Gamma \cup \{\beta\})$ |

These rules are designed to start with the transition $\text{S} \rightarrow [\mathbf{h}\beta 1\beta]$ (Rule (a)) and then rewrite $[\mathbf{h}\beta 1\beta]$ using other rules until the configuration $[\mathbf{s}\beta w_1 w_2 \ldots w_n \beta]$ is reached. At

this point Rule (c) is invoked to strip [$\mathbf{s}\beta$ from the beginning of the string, and Rule (e) strips $\beta$] from the end, thereby producing the string $w_1, w_2, \ldots, w_n$ that was written initially on $M$'s tape.

Rule (b) is used to add blank space at the right-hand end of the tape. Rules (f)–(h) mimic the transitions of $M$ in reverse order. Rule (f) says that if $M$ in state $p$ reading $x$ moves to state $q$ and moves its head right, then $M$'s configuration contained the substring $\mathbf{p}x$ before the move and $x\mathbf{q}$ after it. Thus, we map $x\mathbf{q}$ into $\mathbf{p}x$ with the rule $x\mathbf{q} \rightarrow \mathbf{p}x$. Similar reasoning is applied to Rule (g). If the transition $\delta(p, x) = (q, y)$, $y \in \Gamma \cup \{\beta\}$ is executed, $M$'s configuration contained the substring $\mathbf{p}x$ before the step and $\mathbf{q}y$ after it because the head does not move.

Clearly, every computation by a TM $M$ can be described by a sequence of configurations and the transitions between these configurations can be described by this grammar G. Thus, the strings accepted by $M$ can be generated by $G$. Conversely, if we are given a derivation in $G$, it produces a series of configurations characterizing computations by the TM $M$ in reverse order. Thus, the strings generated by $G$ are the strings accepted by $M$. ■

By showing that every phrase-structure language can be accepted by a Turing machine, we will have demonstrated the equivalence between the phrase-structure and recursively enumerable languages.

**THEOREM 5.4.2** *Every phrase-structure language is recursively enumerable.*

**Proof** Given a phrase-structure grammar $G$, we construct a nondeterministic two-tape TM $M$ with the property that $L(G) = L(M)$. Because every language accepted by a multi-tape TM is accepted by a one-tape TM and vice versa, we have the desired conclusion.

To decide whether or not to accept an input string placed on its first (input) tape, $M$ nondeterministically generates a terminal string on its second (work) tape using the rules of $G$. To do so, it puts $G$'s start symbol on its work tape and then nondeterministically expands it into a terminal string using the rules of $G$. After producing a terminal string, $M$ compares the input string with the string on its work tape. If they agree in every position, $M$ accepts the input string. If not, $M$ enters an infinite loop. To write the derived strings on its work tape, $M$ must either replace, delete, or insert characters in the string on its tape, tasks well suited to Turing machines.

Since it is possible for $M$ to generate every string in $L(G)$ on its work tape, it can accept every string in $L(G)$. On the other hand, every string accepted by $M$ is a string that it can generate using the rules of $G$. Thus, every string accepted by $M$ is in $L(G)$. It follows that $L(M) = L(G)$. ■

This last result gives meaning to the phrase "recursively enumerable": the languages accepted by Turing machines (the recursively enumerable languages) are languages whose strings can be enumerated by a Turing machine (a recursive device). Since an NDTM can be simulated by a DTM, all strings accepted by a TM can be generated deterministically in sequence.

## 5.5 Universal Turing Machines

A universal Turing machine is a Turing machine that can simulate the behavior of an arbitrary Turing machine, even the universal Turing machine itself. To give an explicit construction for such a machine, we show how to encode Turing machines as strings.

Without loss of generality we consider only deterministic Turing machines $M = (\Gamma, \beta, Q, \delta, s, h)$ that have a binary tape alphabet $\Gamma = \mathcal{B} = \{0, 1\}$. When $M$ is in state $p$ and the value under the head is $a$, the next-state function $\delta : Q \times (\Gamma \cup \{\beta\}) \mapsto (Q \cup \{h\}) \times (\Gamma \cup \{\beta\} \cup \{\mathbf{L}, \mathbf{R}\})$ takes $M$ to state $q$ and provides output $z$, where $\delta(p, a) = (q, z)$ and $z \in \Gamma \cup \{\beta\} \cup \{\mathbf{L}, \mathbf{R}\}$.

We now specify a convention for numbering states that simplifies the description of the next-state function $\delta$ of $M$.

**DEFINITION 5.5.1** *The* **canonical encoding** *of a Turing machine $M$, $\rho(M)$, is a string over the 10-letter alphabet $\Lambda = \{<, >, [, ], \#, 0, 1, \beta, \mathbf{R}, \mathbf{L}\}$ formed as follows:*

*(a) Let $Q = \{q_1, q_2, \ldots, q_k\}$ where $s = q_1$. Represent state $q_i$ in unary notation by the string $1^i$. The halt state $h$ is represented by the empty string.*

*(b) Let $(q, z)$ be the value of the next-state function when $M$ is in state $p$ reading $a$ under its tape head; that is, $\delta(p, a) = (q, z)$. Represent $(q, z)$ by the string $< z\#q >$ in which $q$ is represented in unary and $z \in \{0, 1, \beta, \mathbf{L}, \mathbf{R}\}$. If $q = h$, the value of the next-state function is $< z\# >$.*

*(c) For $p \in Q$, the three values $< z'\#q' >$, $< z''\#q'' >$, and $< z'''\#q''' >$ of $\delta(p, 0)$, $\delta(p, 1)$, and $\delta(p, \beta)$ are assembled as a triple $[< z'\#q' >< z''\#q'' >< z'''\#q''' >]$. The complete description of the next-state function $\delta$ is given as a sequence of such triples, one for each state $p \in Q$.*

To illustrate this definition, consider the two TMs whose next-state functions are shown in Fig. 5.3. The first moves across the non-blank initial string on its tape and halts over the first blank symbol. The second moves the input string right one position and inserts a blank to its left. The canonical encoding of the first TM is $[< \mathbf{R}\#1 >< \mathbf{R}\#1 >< \beta\# >]$ whereas that of the second is

$$
\begin{array}{lll}
[< \beta\#11 > & < \beta\#111 > & < \beta\# >] \\
[< \mathbf{R}\#1111 > & < \mathbf{R}\#1111 > & < \mathbf{R}\#1111 >] \\
[< \mathbf{R}\#11111 > & < \mathbf{R}\#11111 > & < \mathbf{R}\#11111 >] \\
[< 0\#11 > & < 0\#111 > & < 0\# >] \\
[< 1\#11 > & < 1\#111 > & < 1\# >]
\end{array}
$$

It follows that the canonical encodings of TMs are a subset of the strings defined by the regular expression $([(< \{0, 1, \beta, \mathbf{L}, \mathbf{R}\}\#1^* >)^3])^*$ which a TM can analyze to insure that for each state and tape letter there is a valid action.

A **universal Turing machine** (UTM) $U$ is a Turing machine that is capable of simulating an arbitrary Turing machine on an arbitrary input word $\boldsymbol{w}$. The construction of a UTM based on the simulation of the random-access machine is described in Section 3.8. Here we describe a direct construction of a UTM.

Let the UTM $U$ have a 20-letter alphabet $\widehat{\Lambda}$ containing the 10 symbols in $\Lambda$ plus another 10 symbols that are marked copies of the symbols in $\Lambda$. (The marked copies are used to simulate multiple tracks on a one-track TM.) That is, we define $\widehat{\Lambda}$ as follows:

$$
\widehat{\Lambda} = \{<, >, [, ], \#, 0, 1, \beta, \mathbf{R}, \mathbf{L}\} \cup \{\widehat{<}, \widehat{>}, \widehat{[}, \widehat{]}, \widehat{\#}, \widehat{0}, \widehat{1}, \widehat{\beta}, \widehat{\mathbf{R}}, \widehat{\mathbf{L}}\}
$$

To simulate the TM $M$ on the input string $\boldsymbol{w}$, we place $M$'s canonical encoding, $\rho(M)$, on the tape of the UTM $U$ preceded by $\beta$ and followed by $\boldsymbol{w}$, as suggested in Fig. 5.10. The

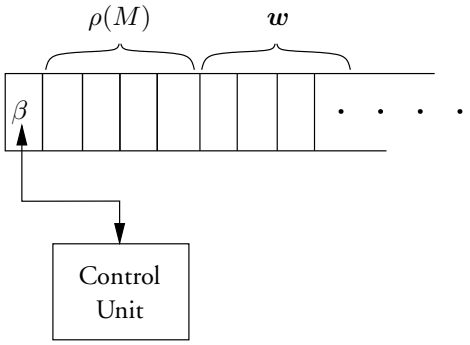


**Figure 5.10** The initial configuration of the tape of a universal TM that is prepared to simulate the TM $M$ on input $\boldsymbol{w}$. The left end-of-tape marker is the blank symbol $\beta$.

first letter of $\boldsymbol{w}$ follows the rightmost bracket, ], and is marked by replacing it with its marked equivalent, $\widehat{w}_1$. The current state $q$ of $M$ is identified by replacing the left bracket, [, in $q$'s triple by its marked equivalent, $\widehat{[}$. $U$ simulates $M$ by reading the marked input symbol $a$, the one that resides under $M$'s simulated head, and advancing its own head to the triple to the right of $\widehat{[}$ that corresponds to $a$. (Before it moves its head, it replaces $\widehat{[}$ with [.) That is, it advances its head to the first, second, or third triple associated with the current state depending on whether $a$ is 0, 1, or $\beta$. It then changes $<$ to $\widehat{<}$, moves to the symbol following $\widehat{<}$ and takes the required action on the simulated tape. If the action requires writing a symbol, it replaces $a$ with a new marked symbol. If it requires moving $M$'s head, the marking on $a$ is removed and the appropriate adjacent symbol is marked. $U$ returns to $\widehat{<}$ and removes the mark.

The UTM $U$ moves to the next state as follows. It moves its head three places to the right of $\widehat{<}$ after changing it to $<$, at which point it is to the right of $\#$, over the first digit representing the next state. If the symbol in this position is $>$, the next state is $h$, the halting state, and the UTM halts. If the symbol is 1, $U$ replaces it with $\widehat{1}$ and then moves its head left to the leftmost instance of [ (the leftmost tape cell contains $\beta$, an end-of tape marker). It marks [ and returns to $\widehat{1}$. It replaces $\widehat{1}$ with 1 and moves its head right one place. If $U$ finds the symbol 1, it marks it, moves left to $\widehat{[}$, restores it to [ and then moves right to the next instance of [ and marks it. It then moves right to $\widehat{1}$ and repeats this operation. However, if the UTM finds the symbol $>$, it has finished updating the current state so it moves right to the marked tape symbol, at which point it reads the symbol under $M$'s head and starts another transition cycle. The details of this construction are left to the reader. (See Problem 5.15.)

## 5.6 Encodings of Strings and Turing Machines

Given an alphabet $\mathcal{A}$ with an ordering of its letters, strings over this alphabet have an order known as the standard **lexicographical order**, which we now define. In this order, strings of length $n-1$ precede strings of length $n$. Thus, if $\mathcal{A} = \{0, 1, 2\}$, $201 < 0001$. Among the strings of length $n$, if $a$ and $b$ are in $\mathcal{A}$ and $a < b$, then all strings beginning with $a$ precede those beginning with $b$. For example, if $0 < 1 < 2$ in $\mathcal{A} = \{0, 1, 2\}$, then $022 < 200$. If two strings of length $n$ have the same prefix $\boldsymbol{u}$, the ordering between them is determined by the

order of the next letter. For example, for the alphabet $\mathcal{A}$ and the ordering given on its letters, $201021 < 201200$.

A simple algorithm produces the strings over an alphabet in lexicographical order. Strings of length 1 are produced by enumerating the letters from the alphabet in increasing order. Strings of length $n$ are enumerated by choosing the first letter from the alphabet in increasing order. The remaining $n-1$ letters are generated in lexicographical order by applying this algorithm recursively on strings of length $n-1$.

To prepare for later results, we observe that it is straightforward to test an arbitrary string over the alphabet $\Lambda$ given in Definition 5.5.1 to determine if it is a canonical description $\rho(M)$ of a Turing machine $M$. Each must be contained in $([(< \{0, 1, \beta, \mathbf{L}, \mathbf{R}\}\#1^* >)^3])^*$ and have a transition for each state and tape letter. If a putative encoding is not canonical, we associate with it the two-state **null TM** $T_{\text{null}}$ with next-state function satisfying $\delta(s, a) = (h, a)$ for all tape letters $a$. This encoding associates a Turing machine with each string over the alphabet $\Lambda$.

We now show how to identify the ***j*th Turing machine**, $M_j$. Given an order to the symbols in $\Lambda$, strings over this alphabet are generated in lexicographical order. We define the null TM to be the zeroth TM. Each string over $\Lambda$ that is not a canonical encoding is associated with this machine. The first TM is the one described by the lexicographically first string over $\Lambda$ that is a canonical encoding. The second TM is described by the second canonical encoding, etc. Not only does a TM determine which string is a canonical encoding, but when combined with an algorithm to generate strings in lexicographical order, this procedure also assigns a Turing machine to each string and allows the $j$th Turing machine to be found.

Observe that there is no loss in generality in assuming that the encodings of Turing machines are binary strings. We need only create a mapping from the letters in the alphabet $\Lambda$ to binary strings. Since it may be necessary to use marked letters, we can assume that the 20 strings in $\widehat{\Lambda}$ are available and are encoded into 5-bit binary strings. This allows us to view encodings of Turing machines as binary strings but to speak of the encodings in terms of the letters in the alphabet $\Lambda$.

## 5.7 Limits on Language Acceptance

A language $L$ that is **decidable** (also called **recursive**) has an algorithm, a Turing machine that halts on all inputs and accepts just those strings in $L$. A language for which there is a Turing machine that accepts just those strings in $L$, possibly not halting on strings not in $L$, is **recursively enumerable**. A language that is recursively enumerable but not decidable is **unsolvable**.

We begin by describing some decidable languages and then exhibit a language, $\mathcal{L}_1$, that is not recursively enumerable (no Turing machine exists to accepts strings in it) but whose complement, $\mathcal{L}_2$, is recursively enumerable but not decidable; that is, $\mathcal{L}_2$ is unsolvable. We use the language $\mathcal{L}_2$ to show that other languages, including the halting problem, are unsolvable.

### 5.7.1 Decidable Languages

Our first decidable problem is the language of pairs of regular expressions and strings such that the regular expression describes a language containing the corresponding string:

$$\mathcal{L}_{\text{RX}} = \{R, \boldsymbol{w} \mid \boldsymbol{w} \text{ is in the language described by the regular expression } R\}$$

**THEOREM 5.7.1** *The language $\mathcal{L}_{\mathrm{RX}}$ is decidable.*

**Proof** To decide on a string $R, \boldsymbol{w}$, use the method of Theorem 4.4.1 to construct a NFSM $M_1$ that accepts the language described by $R$. Then invoke the method of Theorem 4.2.1 to construct a DFSM $M_2$ accepting the same language as $M_1$. The string $\boldsymbol{w}$ is given to $M_2$, which accepts it if $R$ can generate it and rejects it otherwise. This procedure decides $\mathcal{L}_{\mathrm{RX}}$ because it halts on all strings $R, \boldsymbol{w}$, whether in $\mathcal{L}_{\mathrm{RX}}$ or not. ■

As a second example, we show that finite-state machines that recognize empty languages are decidable. Here an FSM encoded as Turing machine reads one input from the tape per step and makes a state transition, halting when it reaches the blank letter.

**THEOREM 5.7.2** *The language $L = \{\rho(M) \mid M$ is a DFSM and $L(M) = \emptyset\}$ is decidable.*

**Proof** $L(M)$ is not empty if there is some string $\boldsymbol{w}$ it can accept. To determine if there is such a string, we use a TM $M'$ that executes a breadth-first search on the graph of the DFSM $M$ that is provided as input to $M'$. $M'$ first marks the initial state of $M$ and then repeatedly marks any state that has not been marked previously and can be reached from a marked state until no additional states can be marked. This process terminates because $M$ has a finite number of states. Finally, $M'$ checks to see if there is a marked accepting state that can be reached from the initial state, rejecting the input $\rho(M)$ if so and accepting it if not. ■

The third language describes context-free grammars generating languages that are empty. Here we encode the definition of a context-free grammar $G$ as a string $\rho(G)$ over a small alphabet.

**THEOREM 5.7.3** *The language $L = \{\rho(G) \mid G$ is a CFG and $L(G) = \emptyset\}$ is decidable.*

**Proof** We design a TM $M'$ that, when given as input a description $\rho(G)$ of a CFG $G$, first marks all the terminals of the grammar and then scans all the rules of the grammar, marking non-terminal symbols that can be replaced by some marked symbols. (If there is a non-terminal $A$ that it is not marked and there is a rule $A \rightarrow BCD$ in which $B, C, D$ have already been marked, then the TM also marks $A$.) We repeat this procedure until no new non-terminals can be marked. This process terminates because the grammar $G$ has a finite number of non-terminals. If $S$ is not marked, we accept $\rho(G)$. Otherwise, we reject $\rho(G)$ because it is possible to generate a string of terminals from $S$. ■

## 5.7.2 A Language That Is Not Recursively Enumerable

Not unexpectedly, there are well-defined languages that are not recursively enumerable, as we show in this section. We also show that the complement of a decidable language is decidable. This allows us to exhibit a language that is recursively enumerable but undecidable.

Consider the language $\mathcal{L}_1$ defined below. It contains the $i$th binary input string if it is not accepted by the $i$th Turing machine.

$$\mathcal{L}_1 = \{\boldsymbol{w}_i \mid \boldsymbol{w}_i \text{ is not accepted by } M_i\}$$

**THEOREM 5.7.4** *The language $\mathcal{L}_1$ is not recursively enumerable; that is, no Turing machine exists that can accept all the strings in this language.*

| | $\rho(M_1)$ | $\rho(M_2)$ | | $\rho(M_k)$ | |
|---|---|---|---|---|---|
| $\mathbf{w}_1$ | reject | accept | ⋯ | reject | ⋯ |
| $\mathbf{w}_2$ | accept | accept | ⋯ | reject | ⋯ |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $\mathbf{w}_k$ | accept | reject | ⋯ | ? | ⋯ |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

**Figure 5.11** A table whose rows and columns are indexed by input strings and Turing machines, respectively. Here $\boldsymbol{w}_i$ is the $i$th input string and $\rho(M_j)$ is the encoding of the $j$th Turing machine. The entry in row $i$, column $j$ indicates whether or not $M_j$ accepts $\boldsymbol{w}_i$. The language $\mathcal{L}_1$ consists of input strings $\boldsymbol{w}_j$ for which the entry in the $j$th row and $j$th column is **reject**.

**Proof** We use proof by contradiction; that is, we assume the existence of a TM $M_k$ that accepts $\mathcal{L}_1$. If $\boldsymbol{w}_k$ is in $\mathcal{L}_1$, then $M_k$ accepts it, contradicting the definition of $\mathcal{L}_1$. This implies that $\boldsymbol{w}_k$ is not in $\mathcal{L}_1$. On the other hand, if $\boldsymbol{w}_k$ is not in $\mathcal{L}_1$, then it is not accepted by $M_k$. It follows from the definition of $\mathcal{L}_1$ that $\boldsymbol{w}_k$ is in $\mathcal{L}_1$. Thus, $\boldsymbol{w}_k$ is in $\mathcal{L}_1$ if and only if it is not in $\mathcal{L}_1$. We have a contradiction and no Turing machine accepts $\mathcal{L}_1$. ■

This proof uses **diagonalization**. (See Fig. 5.11.) In effect, we construct an infinite two-dimensional matrix whose rows are indexed by input words and whose columns are indexed by Turing machines. The entry in row $i$ and column $j$ of this matrix specifies whether or not input word $\boldsymbol{w}_i$ is accepted by $M_j$. The language $\mathcal{L}_1$ contains those words $\boldsymbol{w}_j$ that $M_j$ rejects, that is, it contains row indices (words) for which the word "reject" is found on the diagonal. If we assume that some TM, $M_k$, accepts $\mathcal{L}_1$, we have a problem because we cannot decide whether or not $\boldsymbol{w}_k$ is in $\mathcal{L}_1$. Diagonalization is effective in ruling out the possibility of solving a computational problem but has limited usefulness on problems of bounded size.

### 5.7.3 Recursively Enumerable but Not Decidable Languages

We show the existence of a language that is recursively enumerable but not decidable. Our approach is to show that the complement of a recursive language is recursive and then exhibit a recursively enumerable language $\mathcal{L}_2$ whose complement $\mathcal{L}_1$ is not recursively enumerable:

$$\mathcal{L}_2 = \{\boldsymbol{w}_i \,|\, \boldsymbol{w}_i \text{ is accepted by } M_i\}$$

**THEOREM 5.7.5** *The complement of a decidable language is decidable.*

**Proof** Let $L$ be a recursive language accepted by a Turing machine $M_1$ that halts on all input strings. Relabel the accepting halt state of $M_1$ as non-accepting and all non-accepting halt states as accepting. This produces a machine $M_2$ that enters an accepting halt state only when $M_1$ enters a non-accepting halt state and vice versa. We convert this non-standard machine to standard form (having one accepting halt state) by adding a new accepting halt

state and making a transition to it from all accepting halt states. This new machine halts on all inputs and accepts the complement of $L$. ■

**THEOREM 5.7.6** *The language $\mathcal{L}_2$ is recursively enumerable but not decidable.*

**Proof** To establish the desired result it suffices to exhibit a Turing machine $M$ that accepts each string in $\mathcal{L}_2$, because the complement $\overline{\mathcal{L}_2} = \mathcal{L}_1$, which is not recursively enumerable, as shown above.

Given a string $\boldsymbol{x}$ in $\mathcal{B}^*$, let $M$ enumerate the input strings over the alphabet $\mathcal{B}$ of $\mathcal{L}_2$ until it finds $\boldsymbol{x}$. Let $\boldsymbol{x}$ be the $i$th string where $i$ is recorded in binary on one of $M$'s tapes. The strings over the alphabet $\Lambda$ used for canonical encodings of Turing machines are enumerated and tested to determine whether or not they are canonical encodings, as described in Section 5.6. When the encoding $\rho(M_i)$ of the $i$th Turing machine is discovered, $M_i$ is simulated with a universal Turing machine on the input string $\boldsymbol{x}$. This universal machine will halt and accept the string $\boldsymbol{x}$ if it is in $\mathcal{L}_2$. Thus, $\mathcal{L}_2$ is recursively enumerable. ■

## 5.8 Reducibility and Unsolvability

In this section we show that there are many languages that are unsolvable (undecidable). In the previous section we showed that the language $\mathcal{L}_2$ is unsolvable. To show that a new problem is unsolvable we use reducibility: we assume an algorithm $A$ exists for a new language $L$ and then show that we can use $A$ to obtain an algorithm for a language previously shown to be unsolvable, thereby contradicting the assumption that algorithm $A$ exists.

We begin by introducing reducibility and then give examples of unsolvable languages. Many interesting languages are unsolvable.

### 5.8.1 Reducibility

A new language $\mathcal{L}_{\text{new}}$ can often be shown unsolvable by assuming it is solvable and then showing this implies that an older language $\mathcal{L}_{\text{old}}$ is solvable, where $\mathcal{L}_{\text{old}}$ has been previously shown to be unsolvable. Since this contradicts the facts, the new language cannot be solvable. This is one application of **reducibility**. The formal definition of reducibility is given below and illustrated by Fig. 5.12.

**DEFINITION 5.8.1** *The language $L_1$ is* **reducible** *to the language $L_2$ if there is an algorithm computing a total function $f : \mathcal{C}^* \mapsto \mathcal{D}^*$ that translates each string $\boldsymbol{w}$ over the alphabet $\mathcal{C}$ of $L_1$ into a string $\boldsymbol{z} = f(\boldsymbol{w})$ over the alphabet $\mathcal{D}$ of $L_2$ such that $\boldsymbol{w} \in L_1$ if and only if $\boldsymbol{z} \in L_2$.*

In this definition, testing for membership of a string $\boldsymbol{w}$ in $L_1$ is reduced to testing for membership of a string $\boldsymbol{z}$ in $L_2$, where the latter problem is presumably a previously solved problem. It is important to note that the latter problem is no easier than the former, even though the use of the word "reduce" suggests that it is. Rather, reducibility establishes a link between two problems with the expectation that the properties of one can be used to deduce properties of the other. For example, reducibility is used to identify **NP**-complete problems. (See Sections 3.9.3 and 8.7.)

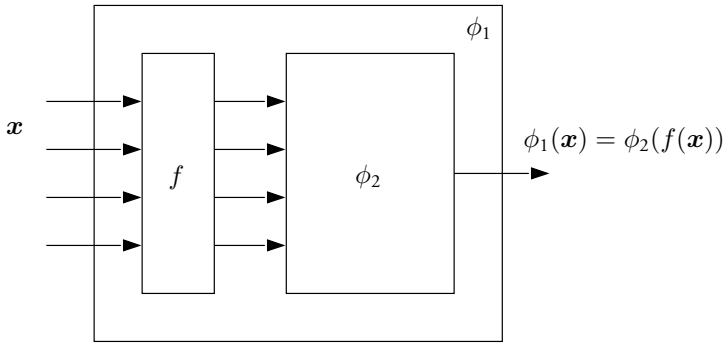


**Figure 5.12** The characteristic function $\phi_i$ of $L_i$, $i = 1, 2$ has value 1 on strings in $L_i$ and 0 otherwise. Because the language $L_1$ is reducible to the language $L_2$, there is a function $f$ such that for all $\boldsymbol{x}$, $\phi_1(\boldsymbol{x}) = \phi_2(f(\boldsymbol{x}))$.

Reducibility is a fundamental idea that is formally introduced in Section 2.4 and used throughout this book. Reductions of the type defined above are known as **many-to-one reductions**. (See Section 8.7 for more on this subject.)

The following lemma is a tool to show that problems are unsolvable. We use the same mechanism in Chapter 8 to classify languages by their use of time, space and other computational resources.

**LEMMA 5.8.1** *Let $L_1$ be reducible to $L_2$. If $L_2$ is decidable, then $L_1$ is decidable. If $L_1$ is unsolvable and $L_2$ is recursively enumerable, $L_2$ is also unsolvable.*

**Proof** Let $T$ be a Turing machine implementing the algorithm that translates strings over the alphabet of $L_1$ to strings over the alphabet of $L_2$. If $L_2$ is decidable, there is a halting Turing machine $M_2$ that accepts it. A multi-tape Turing machine $M_1$ that decides $L_1$ can be constructed as follows: On input string $\boldsymbol{w}$, $M_1$ invokes $T$ to generate the string $\boldsymbol{z}$, which it then passes to $M_2$. If $M_2$ accepts $\boldsymbol{z}$, $M_1$ accepts $\boldsymbol{w}$. If $M_2$ rejects it, so does $M_1$. Thus, $M_1$ decides $L_1$.

Suppose now that $L_1$ is unsolvable. Assuming that $L_2$ is decidable, from the above construction, $L_1$ is decidable, contradicting this assumption. Thus, $L_2$ cannot be decidable. ■

The power of this lemma will be apparent in the next section.

## 5.8.2 Unsolvable Problems

In this section we examine six representative unsolvable problems. They range from the classical halting problem to Rice's theorem.

We begin by considering the **halting problem** for Turing machines. The problem is to determine for an arbitrary TM $M$ and an arbitrary input string $\boldsymbol{x}$ whether $M$ with input $\boldsymbol{x}$ halts or not. We characterize this problem by the language $\mathcal{L}_H$ shown below. We show it is unsolvable, that is, $\mathcal{L}_H$ is recursively enumerable but not decidable. No Turing machine exists to decide this language.

$$\mathcal{L}_H = \{\rho(M), \boldsymbol{w} \mid M \text{ halts on input } \boldsymbol{w}\}$$

**THEOREM 5.8.1** *The language $\mathcal{L}_H$ is recursively enumerable but not decidable.*

**Proof** To show that $\mathcal{L}_H$ is recursively enumerable, pass the encoding $\rho(M)$ of the TM $M$ and the input string $\boldsymbol{w}$ to the universal Turing machine $U$ of Section 5.5. This machine simulates $M$ and halts on the input $\boldsymbol{w}$ if and only if $M$ halts on $\boldsymbol{w}$. Thus, $\mathcal{L}_H$ is recursively enumerable.

To show that $\mathcal{L}_H$ is undecidable, we assume that $\mathcal{L}_H$ is decidable by a Turing machine $M_H$ and show a contradiction. Using $M_H$ we construct a Turing machine $M^*$ that decides the language $\mathcal{L}^* = \{\rho(M), \boldsymbol{w} \mid \boldsymbol{w} \text{ is not accepted by } M\}$. $M^*$ simulates $M_H$ on $\rho(M), \boldsymbol{w}$ to determine whether $M$ halts or not on $\boldsymbol{w}$. If $M_H$ says that $M$ does not halt, $M^*$ accepts $\boldsymbol{w}$. If $M_H$ says that $M$ does halt, $M^*$ simulates $M$ on input string $\boldsymbol{w}$ and rejects $\boldsymbol{w}$ if $M$ accepts it and accepts $\boldsymbol{w}$ if $M$ rejects it. Thus, if $\mathcal{L}_H$ is decidable, so is $\mathcal{L}^*$.

The procedures described in Section 5.6 can be used to design a Turing machine $M^\star$ that determines for which integer $i$ the input string $\boldsymbol{w}$ is lexicographically the $i$th string, $\boldsymbol{w}_i$, and also produce the description $\rho(M_i)$ of the $i$th Turing machine $M_i$.

To decide $\mathcal{L}_1$ we use $M^\star$ to translate an input string $\boldsymbol{w} = \boldsymbol{w}_i$ to the string $\rho(M_i), \boldsymbol{w}_i$. Given the presumed existence of $M^*$, we can decide $\mathcal{L}_1$ by deciding $\mathcal{L}^*$. However, by Theorem 5.7.4, $\mathcal{L}_1$ is not decidable (it is not even recursively enumerable). Thus, $\mathcal{L}^*$ is not decidable which implies that $\mathcal{L}_H$ is also not decidable. ■

The second unsolvable problem we consider is the **empty tape acceptance problem**: given a Turing machine $M$, we ask if we can tell whether it accepts the empty string. We reduce the halting problem to it. (See Fig. 5.13.)

$$\mathcal{L}_{\text{ET}} = \{\rho(M) \mid L(M) \text{ contains the empty string}\}$$

**THEOREM 5.8.2** *The language $\mathcal{L}_{\text{ET}}$ is not decidable.*

**Proof** To show that $\mathcal{L}_{\text{ET}}$ is not decidable, we assume that it is and derive a contradiction. The contradiction is produced by assuming the existence of a TM $M_{\text{ET}}$ that decides $\mathcal{L}_{\text{ET}}$ and then showing that this implies the existence of a TM $M_H$ that decides $\mathcal{L}_{\text{H}}$.

Given an encoding $\rho(M)$ for an arbitrary TM $M$ and an arbitrary input $\boldsymbol{w}$, the TM $M_H$ constructs a TM $T(M, \boldsymbol{w})$ that writes $\boldsymbol{w}$ on the tape when the tape is empty and simulates $M$ on $\boldsymbol{w}$, halting if $M$ halts. Thus, $T(M, \boldsymbol{w})$ accepts the empty tape if $M$ halts on $\boldsymbol{w}$. $M_H$ decides $\mathcal{L}_{\text{H}}$ by constructing an encoding of $T(M, \boldsymbol{w})$ and passing it to $M_{\text{ET}}$. (See Fig. 5.13.) The language accepted by $T(M, \boldsymbol{w})$ includes the empty string if and only

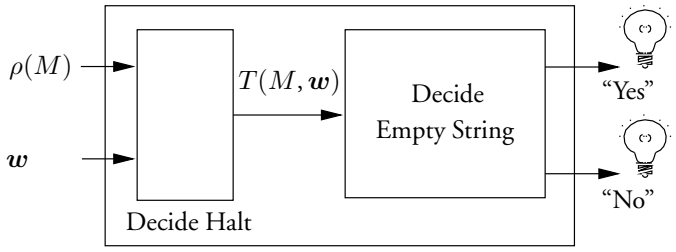


**Figure 5.13** Schematic representation of the reduction from $\mathcal{L}_H$ to $\mathcal{L}_{\text{ET}}$.

if $M$ halts on $\boldsymbol{w}$. Thus, $M_H$ decides the halting problem, which as shown earlier cannot be decided. ■

The third unsolvable problem we consider is the **empty set acceptance problem**: Given a Turing machine, we ask if we can tell if the language it accepts is empty. We reduce the halting problem to this language.

$$\mathcal{L}_{\rm EL} = \{\rho(M) \mid L(M) = \emptyset\}$$

**THEOREM 5.8.3** *The language $\mathcal{L}_{\rm EL}$ is not decidable.*

**Proof** We reduce $\mathcal{L}_H$ to $\mathcal{L}_{\rm EL}$, assume that $\mathcal{L}_{\rm EL}$ is decidable by a TM $M_{\rm EL}$, and then show that a TM $M_H$ exists that decides $\mathcal{L}_H$, thereby establishing a contradiction.

Given an encoding $\rho(M)$ for an arbitrary TM $M$ and an arbitrary input $\boldsymbol{w}$, the TM $M_H$ constructs a TM $T(M, \boldsymbol{w})$ that accepts the string placed on its tape if it is $\boldsymbol{w}$ and $M$ halts on it; otherwise it enters an infinite loop. $M_H$ can implement $T(M, \boldsymbol{w})$ by entering an infinite loop if its input string is not $\boldsymbol{w}$ and otherwise simulating $M$ on $\boldsymbol{w}$ with a universal Turing machine.

It follows that $L(T(M, \boldsymbol{w}))$ is empty if $M$ does not halt on $\boldsymbol{w}$ and contains $\boldsymbol{w}$ if it does halt. Under the assumption that $M_{\rm EL}$ decides $\mathcal{L}_{\rm EL}$, $M_H$ can decide $\mathcal{L}_{\rm H}$ by constructing $T(M, \boldsymbol{w})$ and passing it to $M_{\rm EL}$, which accepts $\rho(T(M, \boldsymbol{w}))$ if $M$ does not halt on $\boldsymbol{w}$ and rejects it if $M$ does halt. Thus, $M_H$ decides $\mathcal{L}_H$, a contradiction. ■

The fourth problem we consider is the **regular machine recognition problem**. In this case we ask if a Turing machine exists that can decide from the description of an arbitrary Turing machine $M$ whether the language accepted by $M$ is regular or not:

$$\mathcal{L}_R = \{\rho(M) \mid L(M) \text{ is regular}\}$$

**THEOREM 5.8.4** *The language $\mathcal{L}_R$ is not decidable.*

**Proof** We assume that a TM $M_R$ exists to decide $\mathcal{L}_R$ and show that this implies the existence of a TM $M_H$ that decides $\mathcal{L}_H$, a contradiction. Thus, $M_R$ cannot exist.

Given an encoding $\rho(M)$ for an arbitrary TM $M$ and an arbitrary input $\boldsymbol{w}$, the TM $M_H$ constructs a TM $T(M, \boldsymbol{w})$ that scans its tape. If it finds a string in $\{0^n1^n \mid n \geq 0\}$, it accepts it; if not, $T(M, \boldsymbol{w})$ erases the tape and simulates $M$ on $\boldsymbol{w}$, halting only if $M$ halts on $\boldsymbol{w}$. Thus, $T(M, \boldsymbol{w})$ accepts all strings in $\mathcal{B}^*$ if $M$ halts on $\boldsymbol{w}$ but accepts only strings in $\{0^n1^n \mid n \geq 0\}$ otherwise. Thus, $T(M, \boldsymbol{w})$ accepts the regular language $\mathcal{B}^*$ if $M$ halts on $\boldsymbol{w}$ and accepts the context-free language $\{0^n1^n \mid n \geq 0\}$ otherwise. Thus, $M_H$ can be implemented by constructing $T(M, \boldsymbol{w})$ and passing it to $M_R$, which is presumed to decide $\mathcal{L}_R$. ■

The fifth problem generalizes the above result and is known as **Rice's theorem**. It says that no algorithm exists to determine from the description of a TM whether or not the language it accepts falls into any proper subset of the recursively enumerable languages.

Let **RE** be the set of recursively enumerable languages over $\mathcal{B}$. For each set $\mathcal{C}$ that is a proper subset of **RE**, define the following language:

$$\mathcal{L}_{\mathcal{C}} = \{\rho(M) \mid L(M) \in \mathcal{C}\}$$

Rice's theorem says that, for all $\mathcal{C}$ such that $\mathcal{C} \neq \emptyset$ and $\mathcal{C} \subset$ **RE**, the language $\mathcal{L}_{\mathcal{C}}$ defined above is undecidable.

**THEOREM 5.8.5 (Rice)** *Let $\mathcal{C} \subset$ **RE**, $\mathcal{C} \neq \emptyset$. The language $\mathcal{L}_\mathcal{C}$ is not decidable.*

**Proof** To prove that $\mathcal{L}_\mathcal{C}$ is not decidable, we assume that it is decidable by the TM $M_\mathcal{C}$ and show that this implies the existence of a TM $M_H$ that decides $\mathcal{L}_H$, which has been shown previously not to exist. Thus, $M_\mathcal{C}$ cannot exist.

We consider two cases, the first in which $\mathcal{B}^*$ is in not $\mathcal{C}$ and the second in which it is in $\mathcal{C}$. In the first case, let $L$ be a language in $\mathcal{C}$. In the second, let $L$ be a language in **RE** $- \mathcal{C}$. Since $\mathcal{C}$ is a proper subset of **RE** and not empty, there is always a language $L$ such that one of $L$ and $\mathcal{B}^*$ is in $\mathcal{C}$ and the other is in its complement **RE** $- \mathcal{C}$.

Given an encoding $\rho(M)$ for an arbitrary TM $M$ and an arbitrary input $\boldsymbol{w}$, the TM $M_H$ constructs a (four-tape) TM $T(M, \boldsymbol{w})$ that simulates two machines in parallel (by alternatively simulating one step of each machine). The first, $M_0$, uses a phrase-structure grammar for $L$ to see if $T(M, \boldsymbol{w})$'s input string $\boldsymbol{x}$ is in $L$; it holds $\boldsymbol{x}$ on one tape, holds the current choice inputs for the NDTM $M_L$ of Theorem 5.4.2 on a second, and uses a third tape for the deterministic simulation of $M_L$. (See the comments following Theorem 5.4.2.) $T(M, \boldsymbol{w})$ halts if $M_0$ generates $\boldsymbol{x}$. The second TM writes $\boldsymbol{w}$ on the fourth tape and simulates $M$ on it. $T(M, \boldsymbol{w})$ halts if $M$ halts on $\boldsymbol{w}$. Thus, $T(M, \boldsymbol{w})$ accepts the regular language $\mathcal{B}^*$ if $M$ halts on $\boldsymbol{w}$ and accepts $L$ otherwise. Thus, $M_H$ can be implemented by constructing $T(M, \boldsymbol{w})$ and passing it to $M_\mathcal{C}$, which is presumed to decide $\mathcal{L}_\mathcal{C}$. ■

Our last problem is the **self-terminating machine problem**. The question addressed is whether a Turing machine $M$ given a description $\rho(M)$ of itself as input will halt or not. The problem is defined by the following language. We give a direct proof that it is undecidable; that is, we do not reduce some other problem to it.

$$\mathcal{L}_{\mathrm{ST}} = \{\rho(M) \mid M \text{ is self-terminating}\}$$

**THEOREM 5.8.6** *The language $\mathcal{L}_{\mathrm{ST}}$ is recursively enumerable but not decidable.*

**Proof** To show that $\mathcal{L}_{\mathrm{ST}}$ is recursively enumerable we exhibit a TM $T$ that accepts strings in $\mathcal{L}_{\mathrm{ST}}$. $T$ makes a copy of its input string $\rho(M)$ and simulates $M$ on $\rho(M)$ by passing $(\rho(M), \rho(M))$ to a universal TM that halts and accepts $\rho(M)$ if it is in $\mathcal{L}_{\mathrm{ST}}$.

To show that $\mathcal{L}_{\mathrm{ST}}$ is not decidable, we assume that it is and arrive at a contradiction. Let $M_{\mathrm{ST}}$ decide $\mathcal{L}_{\mathrm{ST}}$. We design a TM $M^*$ that does the following: $M^*$ simulates $M_{\mathrm{ST}}$ on the input string $\boldsymbol{w}$. If $M_{\mathrm{ST}}$ halts and accepts $\boldsymbol{w}$, $M^*$ enters an infinite loop. If $M_{\mathrm{ST}}$ halts and rejects $\boldsymbol{w}$, $M^*$ accepts $\boldsymbol{w}$. ($M_{\mathrm{ST}}$ halts on all inputs.)

The new machine $M^*$ is either self-terminating or it is not. If $M^*$ is self-terminating, then on input $\rho(M^*)$, which is an encoding of itself, $M^*$ enters an infinite loop because $M_{\mathrm{ST}}$ detects that it is self-terminating. Thus, $M^*$ is not self-terminating. On the other hand, if $M^*$ is not self-terminating, on input $\rho(M^*)$ it halts and accepts $\rho(M^*)$ because $M_{\mathrm{ST}}$ detects that it is not self-terminating and enters the rejecting halt state. But this contradicts the assumption that $M^*$ is not self-terminating. Since we arrive at a contradiction in both cases, the assumption that $\mathcal{L}_{\mathrm{ST}}$ is decidable must be false. ■

## 5.9 Functions Computed by Turing Machines

In this section we introduce the partial recursive functions, a family of functions in which each function is constructed from three basic function types, zero, successor, and projection,

and three operations on functions, composition, primitive recursion, and minimalization. Although we do not have the space to show this, the functions computed by Turing machines are exactly the partial recursive functions. In this section, we show one half of this result, namely, that every partial recursive function can be encoded as a RAM program (see Section 3.4.3) that can be executed by Turing machines.

We begin with the primitive recursive functions then describe the partial recursive functions. We then show that partial recursive functions can be realized by RAM programs.

## 5.9.1 Primitive Recursive Functions

Let $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ be the set of non-negative integers. The partial recursive functions, $f : \mathbb{N}^n \mapsto \mathbb{N}^m$, map $n$-tuples of integers over $\mathbb{N}$ to $m$-tuples of integers in $\mathbb{N}$ for arbitrary $n$ and $m$. Partial recursive functions may be partial functions. They are constructed from three base function types, the **successor function** $S : \mathbb{N} \mapsto \mathbb{N}$, where $S(x) = x + 1$, the **predecessor function** $P : \mathbb{N} \mapsto \mathbb{N}$, where $P(x)$ returns either 0 if $x = 0$ or the integer one less than $x$, and the **projection functions** $U_j^n : \mathbb{N}^n \mapsto \mathbb{N}$, $1 \leq j \leq n$, where $U_j^n(x_1, x_2, \ldots, x_n) = x_j$. These basic functions are combined using a finite number of applications of function composition, primitive recursion, and minimalization.

**Function composition** is studied in Chapters 2 and 6. A function $f : \mathbb{N}^n \mapsto \mathbb{N}$ of $n$ arguments is defined by the composition of a function $g : \mathbb{N}^m \mapsto \mathbb{N}$ of $m$ arguments with $m$ functions $f_1 : \mathbb{N}^n \mapsto \mathbb{N}$, $f_2 : \mathbb{N}^n \mapsto \mathbb{N}$, $\ldots$, $f_m : \mathbb{N}^n \mapsto \mathbb{N}$, each of $n$ arguments, as follows:

$$f(x_1, x_2, \ldots, x_n) = g(f_1(x_1, x_2, \ldots, x_n), \ldots, f_m(x_1, x_2, \ldots, x_n))$$

A function $f : \mathbb{N}^{n+1} \mapsto \mathbb{N}$ of $n + 1$ arguments is defined by **primitive recursion** from a function $g : \mathbb{N}^n \mapsto \mathbb{N}$ of $n$ arguments and a function $h : \mathbb{N}^{n+2} \mapsto \mathbb{N}$ on $n + 2$ arguments if and only if for all values of $x_1, x_2, \ldots, x_n$ and $y$ in $\mathbb{N}$:

$$f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n)$$
$$f(x_1, x_2, \ldots, x_n, y + 1) = h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y))$$

In the above definition if $n = 0$, we adopt the convention that the value of $f$ is a constant. Thus, $f(x_1, x_2, \ldots, x_n, k)$ is defined recursively in terms of $h$ and itself with $k$ replaced by $k - 1$ unless $k = 0$.

**DEFINITION 5.9.1** *The class of* **primitive recursive functions** *is the smallest class of functions that contains the base functions and is closed under composition and primitive recursion.*

Many functions of interest are primitive recursive. Among these is the **zero function** $Z : \mathbb{N} \mapsto \mathbb{N}$, where $Z(x) = 0$. It is defined by primitive recursion by $Z(0) = 0$ and

$$Z(x + 1) = U_2^2(x, Z(x))$$

Other important primitive recursive functions are addition, subtraction, multiplication, and division, as we now show. Let $f_{\text{add}} : \mathbb{N}^2 \mapsto \mathbb{N}$, $f_{\text{sub}} : \mathbb{N}^2 \mapsto \mathbb{N}$, $f_{\text{mult}} : \mathbb{N}^2 \mapsto \mathbb{N}$, and $f_{\text{div}} : \mathbb{N}^2 \mapsto \mathbb{N}$ denote integer addition, subtraction, multiplication, and division.

For the **integer addition function** $f_{\text{add}}$ introduce the function $h_1 : \mathbb{N}^3 \mapsto \mathbb{N}$ on three arguments, where $h_1$ is defined below in terms of the successor and projection functions:

$$h_1(x_1, x_2, x_3) = S(U_3^3(x_1, x_2, x_3))$$

Then, $h_1(x_1, x_2, x_3) = x_3 + 1$. Now define $f_{\text{add}}(x, y)$ using primitive recursion, as follows:

$$f_{\text{add}}(x, 0) = U_1^1(x)$$
$$f_{\text{add}}(x, y + 1) = h_1(x, y, f_{\text{add}}(x, y))$$

The role of $h$ is to carry the values of $x$ and $y$ from one recursive invocation to another. To determine the value of $f_{\text{add}}(x, y)$ from this definition, if $y = 0$, $f_{\text{add}}(x, y) = x$. If $y > 0$, $f_{\text{add}}(x, y) = h_1(x, y - 1, f_{\text{add}}(x, y - 1))$. This in turn causes other recursive invocations of $f_{\text{add}}$. The infix notation $+$ is used for $f_{\text{add}}$; that is, $f_{\text{add}}(x, y) = x + y$.

Because the primitive recursive functions are defined over the non-negative integers, the subtraction function $f_{\text{sub}}(x, y)$ must return the value 0 if $y$ is larger than $x$, an operation called **proper subtraction**. (Its infix notation is $\dot{-}$ and we write $f_{\text{sub}}(x, y) = x \dot{-} y$.) It is defined as follows:

$$f_{\text{sub}}(x, 0) = U_1^1(x)$$
$$f_{\text{sub}}(x, y + 1) = U_3^3(x, y, P(f_{\text{sub}}(x, y)))$$

The value of $f_{\text{sub}}(x, y)$ is $x$ if $y = 0$ and is the predecessor of $f_{\text{sub}}(x, y - 1)$ otherwise.

The **integer multiplication function**, $f_{\text{mult}}$, is defined in terms of the function $h_2 : \mathbb{N}^3 \mapsto \mathbb{N}$:

$$h_2(x_1, x_2, x_3) = f_{\text{add}}(U_1^3(x_1, x_2, x_3), U_3^3(x_1, x_2, x_3))$$

Using primitive recursion, we have

$$f_{\text{mult}}(x, 0) = Z(x)$$
$$f_{\text{mult}}(x, y + 1) = h_2(x, y, f_{\text{mult}}(x, y))$$

The value of $f_{\text{mult}}(x, y)$ is zero if $y = 0$ and otherwise is the result of adding $x$ to itself $y$ times. To see this, note that the value of $h_2$ is the sum of its first and third arguments, $x$ and $f_{\text{mult}}(x, y)$. On each invocation of primitive recursion the value of $y$ is decremented by 1 until the value 0 is reached. The definition of the division function is left as Problem 5.26.

Define the function $f_{\text{sign}} : \mathbb{N} \mapsto \mathbb{N}$ so that $f_{\text{sign}}(0) = 0$ and $f_{\text{sign}}(x + 1) = 1$. To show that $f_{\text{sign}}$ is primitive recursive it suffices to invoke the projection operator formally. A function with value 0 or 1 is called a **predicate**.

## 5.9.2 Partial Recursive Functions

The partial recursive functions are obtained by extending the primitive recursive functions to include minimalization. **Minimalization** defines a function $f : \mathbb{N}^n \mapsto \mathbb{N}$ in terms of a second function $g : \mathbb{N}^{n+1} \mapsto \mathbb{N}$ by letting $f(\boldsymbol{x})$ be the smallest integer $y \in \mathbb{N}$ such that $g(\boldsymbol{x}, y) = 0$ and $g(\boldsymbol{x}, z)$ is defined for all $z \leq y$, $z \in \mathbb{N}$. Note that if $g(\boldsymbol{x}, z)$ is not defined for all $z \leq y$, then $f(\boldsymbol{x})$ is not defined. Thus, minimalization can result in partial functions.

**DEFINITION 5.9.2** *The set of* **partial recursive functions** *is the smallest set of functions containing the base functions that is closed under composition, primitive recursion, and minimalization.*

A partial recursive function that is defined for all points in its domain is called a **recursive function**.

### 5.9.3 Partial Recursive Functions are RAM-Computable

There is a nice correspondence between RAM programs and partial recursive functions. The straight-line programs result from applying composition to the base functions. Adding primitive recursion corresponds to adding for-loops whereas adding minimilization corresponds to adding while loops.

It is not difficult to see that every partial recursive function can be described by a program in the RAM assembly language of Section 3.4.3. For example, to compute the zero function, $Z(x)$, it suffices for a RAM program to clear register $R_1$. To compute the successor function, $S(x)$, it suffices to increment register $R_1$. Similarly, to compute the projection function $U_j^n$, one need only load register $R_1$ with the contents of register $R_j$. Function composition it is straightforward: one need only insure that the functions $f_j$, $1 \leq j \leq m$, deposit their values in registers that are accessed by $g$. Similar constructions are possible for primitive recursion and minimalization. (See Problems 5.29, 5.30, and 5.31.)

## Problems

#### THE STANDARD TURING MACHINE MODEL

5.1 Show that the standard Turing machine model of Section 5.1 and the model of Section 3.7 are equivalent in that one can simulate the other.

#### PROGRAMMING THE TURING MACHINE

5.2 Describe a Turing machine that generates the binary strings in lexicographical order. The first few strings in this ordering are 0, 1, 00, 01, 10, 11, 000, 001, ....

5.3 Describe a Turing machine recognizing $\{x^i y^j x^k \mid i, j, k \geq 1 \text{ and } k = i \cdot j\}$.

5.4 Describe a Turing machine that computes the function whose value on input $a^i b^j$ is $c^k$, where $k = i \cdot j$.

5.5 Describe a Turing machine that accepts the string $(\boldsymbol{u}, \boldsymbol{v})$ if $\boldsymbol{u}$ is a substring of $\boldsymbol{v}$.

5.6 The **element distinctness language**, $L_{\text{ed}}$, consists of binary strings no two of which are the same; that is, $L_{\text{ed}} = \{2\boldsymbol{w}_1 2 \ldots 2\boldsymbol{w}_k 2 \mid \boldsymbol{w}_i \in \mathcal{B}^* \text{ and } \boldsymbol{w}_i \neq \boldsymbol{w}_j, \text{for } i \neq j\}$. Describe a Turing machine that accepts this language.

#### EXTENSIONS TO THE STANDARD TURING MACHINE MODEL

5.7 Given a Turing machine with a double-ended tape, show how it can be simulated by one with a single-ended tape.

5.8 Show equivalence between the standard Turing machine and the one-tape **double-headed** Turing machine with two heads that can move independently on its one tape.

5.9 Show that a pushdown automaton with two pushdown tapes is equivalent to a Turing machine.

5.10 Figure 5.14 shows a representation of a Turing machine with a two-dimensional tape whose head can move one step vertically or horizontally. Give a complete definition of a two-dimensional TM and sketch a proof that it can be simulated by a standard TM.

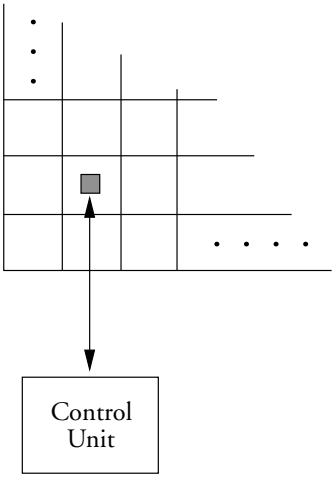


**Figure 5.14** A schematic representation of a two-dimensional Turing machine.

5.11 By analogy with the construction given in Section 3.9.7, show that every deterministic $T$-step multi-tape Turing machine computation can be simulated on a two-tape Turing machine in $O(T \log T)$ steps.

#### PHRASE-STRUCTURE LANGUAGES AND TURING MACHINES

5.12 Give a detailed design of a Turing machine recognizing $\{a^n b^n c^n \mid n \geq 1\}$.

5.13 Use the method of Theorem 5.4.1 to construct a phrase-structure grammar generating $\{a^n b^n c^n \mid n \geq 1\}$.

5.14 Design a Turing machine recognizing the language $\{0^{2^i} \mid i \geq 1\}$.

#### UNIVERSAL TURING MACHINES

5.15 Using the description of Section 5.5, give a complete description of a universal Turing machine.

5.16 Construct a universal TM that has only two non-accepting states.

#### DECIDABLE PROBLEMS

5.17 Show that the following languages are decidable:

a) $L = \{\rho(M), \boldsymbol{w} \mid M$ is a DFSM that accepts the input string $\boldsymbol{w}\}$

b) $L = \{\rho(M) \mid M$ is a DFSM and $L(M)$ is infinite$\}$

5.18 The **symmetric difference** between sets $A$ and $B$ is defined by $(A - B) \cup (B - A)$, where $A - B = A \cap \overline{B}$. Use the symmetric difference to show that the following language is decidable:

$\mathcal{L}_{\text{EQ\_FSM}} = \{\rho(M_1), \rho(M_2) \mid M_1$ and $M_2$ are FSMs recognizing the same language$\}$

5.19 Show that the following language is decidable:

$$L = \{\rho(G), \boldsymbol{w} \mid \rho(G) \text{ encodes a CFG } G \text{ that generates } \boldsymbol{w}\}$$

**Hint:** How long is a derivation of $\boldsymbol{w}$ if $G$ is in Chomsky normal form?

5.20 Show that the following language is decidable:

$$L = \{\rho(G) \mid \rho(G) \text{ encodes a CFG } G \text{ for which } L(G) \neq \emptyset\}$$

5.21 Let $L_1, L_2 \in \mathbf{P}$ where $\mathbf{P}$ is the class of polynomial-time problems (see Definition 3.7.2). Show that the following statements hold:

a) $L_1 \cup L_2 \in \mathbf{P}$

b) $L_1 L_2 \in \mathbf{P}$, where $L_1 L_2$ is the concatenation of $L_1$ and $L_2$

c) $\overline{L_1} \in \mathbf{P}$

5.22 Let $L_1 \in \mathbf{P}$. Show that $L_1^* \in \mathbf{P}$.

**Hint:** Try using dynamic programming, the algorithmic concept illustrated by the parsing algorithm of Theorem 4.11.2.

### UNSOLVABLE PROBLEMS

5.23 Show that the problem of determining whether an arbitrary TM starting with a blank tape will ever halt is unsolvable.

5.24 Show that the following language is undecidable:

$$L_{\text{EQ}} = \{\rho(M_1), \rho(M_2) \mid L(M_1) = L(M_2)\}$$

5.25 Determine which of the following problems are solvable and unsolvable. Defend your conclusions.

a) $\{\rho(M), \boldsymbol{w}, p \mid M \text{ reaches state } p \text{ on input } \boldsymbol{w} \text{ from its initial state}\}$

b) $\{\rho(M), \boldsymbol{p} \mid \text{there is a configuration } [u_1 \ldots u_m \boldsymbol{q} v_1 \ldots v_n] \text{ yielding a configuration containing state } \boldsymbol{p}\}$

c) $\{\rho(M), a \mid M \text{ writes character } a \text{ when started on the empty tape}\}$

d) $\{\rho(M) \mid M \text{ writes a non-blank character when started on the empty tape}\}$

e) $\{\rho(M), \boldsymbol{w} \mid \text{on input } \boldsymbol{w}\ M \text{ moves its head to the left}\}$

### FUNCTIONS COMPUTED BY TURING MACHINES

5.26 Define the integer division function $f_{\text{div}} : \mathbb{N}^2 \mapsto \mathbb{N}$ using primitive recursion.

5.27 Show that the function $f_{\text{remain}} : \mathbb{N}^2 \mapsto \mathbb{N}$ that provides the remainder of $x$ after division by $y$ is a primitive recursive function.

5.28 Show that the factorial function $x!$ is primitive recursive.

5.29 Write a RAM program (see Section 3.4.3) to realize the composition operation.

5.30 Write a RAM program (see Section 3.4.3) to realize the primitive recursion operation.

5.31 Write a RAM program (see Section 3.4.3) to realize the minimalization operation.

## Chapter Notes

Alan Turing introduced the Turing machine, gave an example of a universal machine and demonstrated the unsolvability of the halting problem in [337]. A similar model was independently developed by Post [254]. Chomsky [69] demonstrated the equivalence of phrase-structure languages. Rice's theorem is presented in [279].

Church gave a formal model of computation in [72]. The equivalence between the partial recursive functions and the Turing computable functions was shown by Kleene [167].

For a more extensive introduction to Turing machines, see the books by Hopcroft and Ullman [140] and Lewis and Papadimitriou [199].