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 β. A string of symbols from the tape alphabet Γ 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.

DEFINITION 5.1.1 *A* **standard Turing machine (TM)** *is a six-tuple* $M = (\Gamma, \beta, Q, \delta, s, h)$ *where* Γ *is the* **tape alphabet** *not containing the* **blank symbol** β, Q *is the* **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, 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 strings in* L.

A function $f : \Gamma^* \mapsto \Gamma^* \cup \perp$, *where* $\perp$ *is a symbol that is not in* Γ, *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,* 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.*

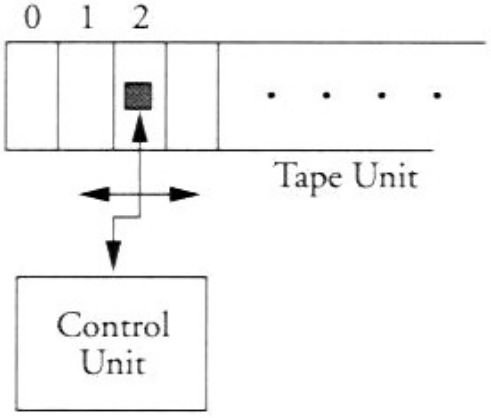

Figure 5.1 The control and tape units of the standard Turing machine.

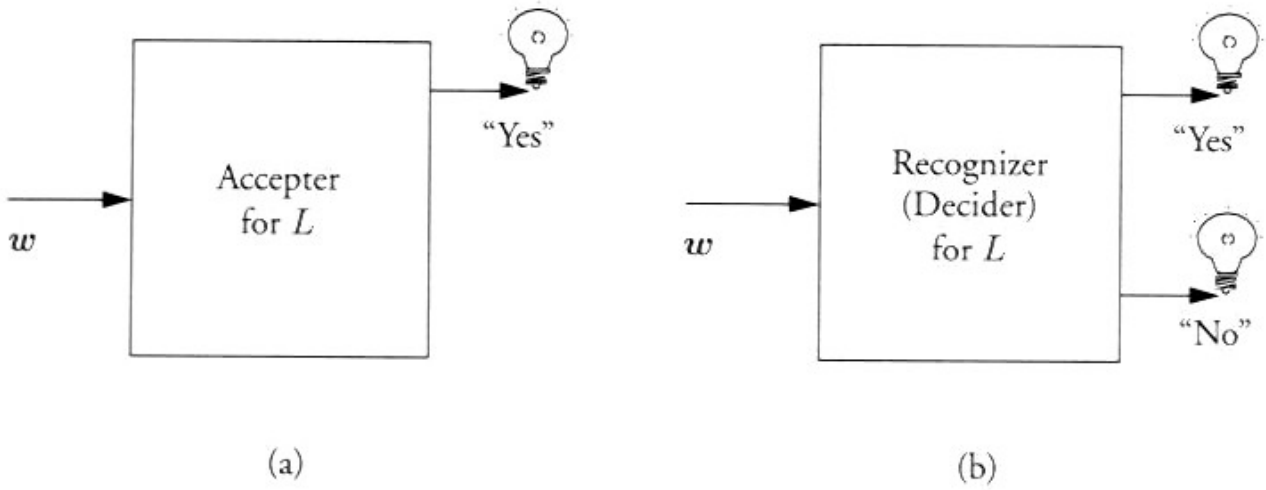

Figure 5.2 An accepter (a) for a language L is a Turing machine that can accept strings in a language L but may not halt on all inputs. A decider or recognizer (b) for a language L is a Turing machine that halts on all inputs and accepts strings in L.

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 machine's task. If the machine's head is initially

q	a	$\delta(\sigma, q)$	
q_1	0	q_1	**R**
q_1	1	q_1	**R**
q_1	β	h	β

(a)

q	a	$\delta(\sigma, q)$	
q_1	0	q_2	β
q_1	1	q_3	β
q_1	β	h	β
q_2	0	q_4	**R**
q_2	1	q_4	**R**
q_2	β	q_4	**R**
q_3	0	q_5	**R**
q_3	1	q_5	**R**
q_3	β	q_5	**R**
q_4	0	q_2	0
q_4	1	q_3	0
q_4	β	h	0
q_5	0	q_2	1
q_5	1	q_3	1
q_5	β	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_4 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 β) 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

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 $L = \{a^n b^n c^n \mid n \geq 1\}$; 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 Γ. 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 Γ and Γ' be the tape alphabets of M_k and M_1, respectively. Let $|\Gamma'| = (2|\Gamma|)^k$ so that Γ' 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 Γ or a marked member of Γ. 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_1. In a

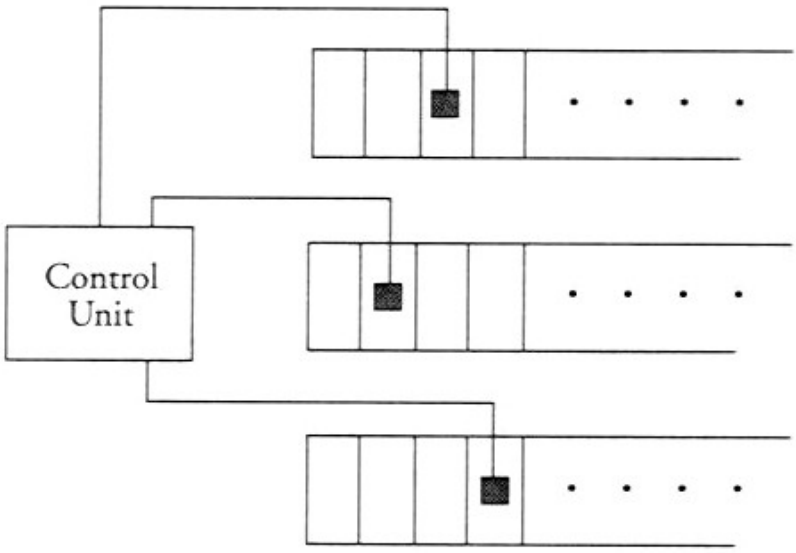

Figure 5.4 A three-tape Turing machine.

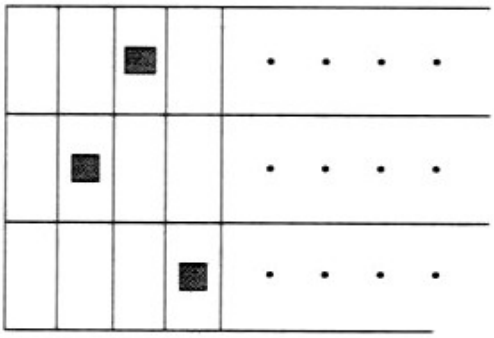

Figure 5.5 A single tape of a TM with a large tape alphabet that simulates a three-tape TM with a smaller tape alphabet.

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 (NTM)** *is a seven-tuple* $M = (\Sigma, \Gamma, \beta, Q, \delta, s, h)$ *where* Σ *is the* **choice input alphabet**, Γ *is the* **tape alphabet** *not containing the* **blank symbol** β, Q *is the* **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*

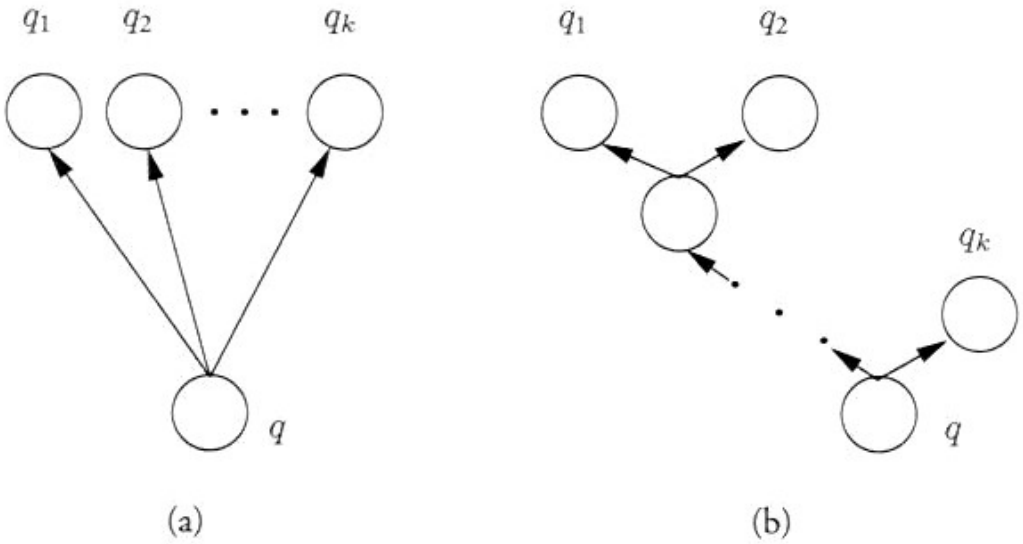

Figure 5.6 The construction used to reduce the fan-out of a nondeterministic state.

moves the head left (if possible) or right if **C** *is* **L** *or* **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 NTM M *reads one character of its* **choice input string** $\mathbf{c} \in \Sigma^*$ *on each step. An NTM* M **accepts string** $\boldsymbol{w}$ *if there is some choice string* $\mathbf{c}$ *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.*

An NTM 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_{ND} on an input string $\boldsymbol{w}$ by a deterministic three-tape TM M_{D} and halt if we find a sequence of moves by M_{ND} that leads to an accepting halt state. Later this machine can be simulated by a one-tape TM. The three tapes of M_{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_{ND}. The enumeration

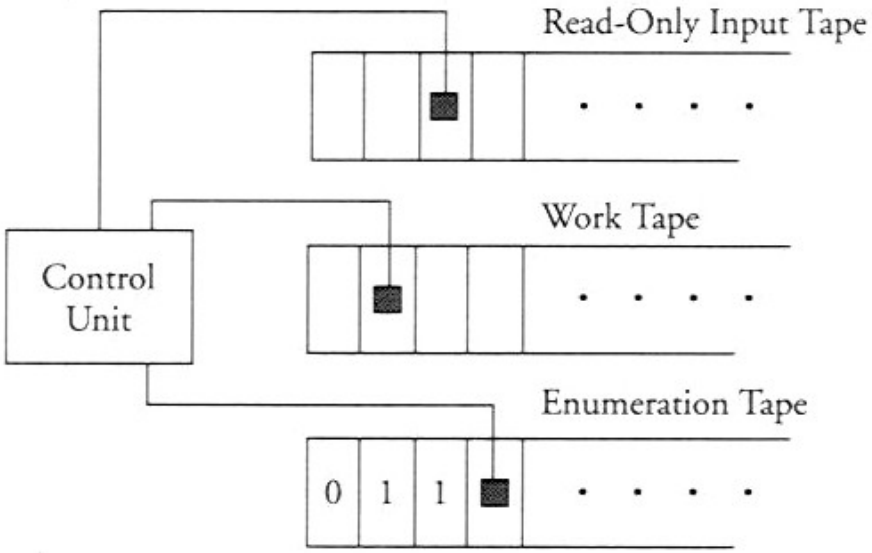

Figure 5.7 A three-tape deterministic Turing machine that simulates a nondeterministic Turing machine.

lating M_{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.)

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_{D} that accepts the input $\boldsymbol{w}$ accepted by M_{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_{ND} on the work tape while reading choice inputs from the enumeration tape. If M_{D} runs out of choice inputs before reaching the halt state, the above procedure is repeated; that is, the computation is restarted with the next choice input sequence. This breadth-first searching method deterministically accepts the input string $\boldsymbol{w}$ if and only if there is some choice input to M_{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 DTM 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 z on its oracle tape, the OTM signals to the oracle to replace

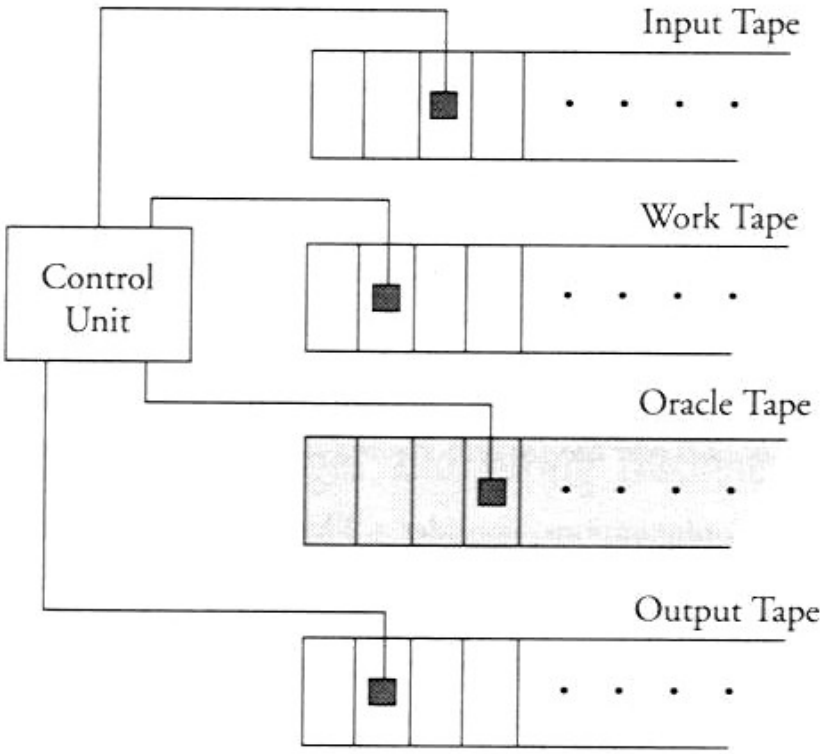

Figure 5.8 The oracle Turing machine has an "oracle tape" on which it writes a string (a problem instance), after which an "oracle" returns an answer in one step.

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

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 δ 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 δ 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 valid encodings of TMs can be described by the regular expression $([(< \{0, 1, \beta, \mathbf{L}, \mathbf{R}\}\#1^* >)^3])^*$. Consequently, a finite-state machine can determine in one pass over a string drawn from the alphabet Γ whether or not it is a valid encoding.

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 Λ plus another 10 symbols that are marked copies of the symbols in Λ. (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, $\rho(M)$,

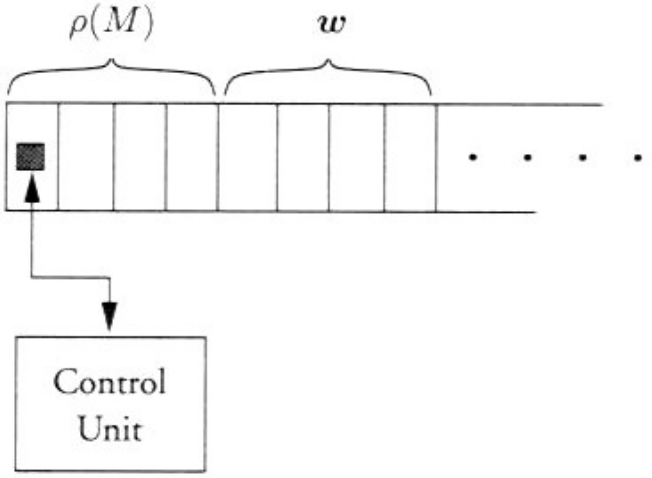

Figure 5.10 The initial configuration of the tape of a universal TM that is prepared to simulate the TM M on input w.

follows the rightmost bracket,], and is marked by replacing it with its marked equivalent, $\widehat{]}$. 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 β. It then moves to the symbol following $<$ 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.

The UTM U moves to the next state as follows. It moves its head right two places, 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 [. It marks this symbol and then 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

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 Λ given in Definition 5.5.1 to determine if it is a valid description $\rho(M)$ of a Turing machine M. This follows because the valid forms of $\rho(M)$ can be described by regular expressions and detected by a finite-state machine. If a putative canonical encoding does not correspond to a valid encoding, we associate with it the two-state **null TM** T_{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 Λ.

We now show how to identify the ***j*th Turing machine**, M_j. Given an order to the symbols in Λ, strings over this alphabet are generated in lexicographical order. We define the null TM to be the zeroth TM. Each string over Λ that is not a valid encoding is associated with this machine. The first TM is the one described by the lexicographically first string over Λ that is a valid encoding. The second TM is described by the second valid encoding, etc. Not only does a finite-state machine determine which string is a valid 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 jth 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 Λ 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 Λ.