CHAPTER

7

TURING MACHINES

In this chapter we introduce the Turing machine, a simple mathematical model of a computer. Despite its simplicity, the Turing machine models the computing capability of a general-purpose computer. The Turing machine is studied both for the class of languages it defines (called the recursively enumerable sets) and the class of integer functions it computes (called the partial recursive functions). A variety of other models of computation are introduced and shown to be equivalent to the Turing machine in computing power.

7.1 INTRODUCTION

The intuitive notion of an algorithm or effective procedure has arisen several times. In Chapter 3 we exhibited an effective procedure to determine if the set accepted by a finite automation was empty, finite, or infinite. One might naively assume that for any class of languages with finite descriptions, there exists an effective procedure for answering such questions. However, this is not the case. For example, there is no algorithm to tell whether the complement of a CFL is empty (although we can tell whether the CFL itself is empty). Note that we are not asking for a procedure that answers the question for a specific context-free language, but rather a single procedure that will correctly answer the question for all CFL's. It is clear that if we need only determine whether one specific CFL has an empty complement, then an algorithm to answer the question exists. That is, there is one algorithm that says "yes" and another that says "no," independent of their inputs. One of these must be correct. Of course, which of the two algorithms answers the question correctly may not be obvious.

At the turn of the century, the mathematician David Hilbert set out on a program to find an algorithm for determining the truth or falsity of any mathematical proposition. In particular, he was looking for a procedure to determine if an arbitrary formula in the first-order predicate calculus, applied to integers, was true. Since the first-order predicate calculus is powerful enough to express the statement that the language generated by a context-free grammar is Σ^*, had Hilbert been successful, our problem of deciding whether the complement of a CFL is empty would be solved. However, in 1931, Kurt Gödel published his famous incompleteness theorem, which proved that no such effective procedure could exist. He constructed a formula in the predicate calculus applied to integers, whose very definition stated that it could neither be proved nor disproved within this logical system. The formalization of this argument and the subsequent clarification and formalization of our intuitive notion of an effective procedure is one of the great intellectual achievements of this century.

Once the notion of an effective procedure was formalized, it was shown that there was no effective procedure for computing many specific functions. Actually the existence of such functions is easily seen from a counting argument. Consider the class of functions mapping the nonnegative integers onto $\{0, 1\}$. These functions can be put into one-to-one correspondence with the reals. However, if we assume that effective procedures have finite descriptions, then the class of all effective procedures can be put into one-to-one correspondence with the integers. Since there is no one-to-one correspondence between the integers and the reals, there must exist functions with no corresponding effective procedures to compute them. There are simply too many functions, a noncountable number, and only a countable number of procedures. Thus the existence of noncomputable functions is not surprising. What is surprising is that some problems and functions with genuine significance in mathematics, computer science, and other disciplines are noncomputable.

Today the Turing machine has become the accepted formalization of an effective procedure. Clearly one cannot prove that the Turing machine model is equivalent to our intuitive notion of a computer, but there are compelling arguments for this equivalence, which has become known as Church's hypothesis. In particular, the Turing machine is equivalent in computing power to the digital computer as we know it today and also to all the most general mathematical notions of computation.

7.2 THE TURING MACHINE MODEL

A formal model for an effective procedure should possess certain properties. First, each procedure should be finitely describable. Second, the procedure should consist of discrete steps, each of which can be carried out mechanically. Such a model was introduced by Alan Turing in 1936. We present a variant of it here.

The basic model, illustrated in Fig. 7.1, has a finite control, an input tape that is divided into cells, and a tape head that scans one cell of the tape at a time. The tape has a leftmost cell but is infinite to the right. Each cell of the tape may hold exactly one of a finite number of tape symbols. Initially, the n leftmost cells, for some finite $n \geq 0$, hold the input, which is a string of symbols chosen from a subset of the tape symbols called the input symbols. The remaining infinity of cells each hold the blank, which is a special tape symbol that is not an input symbol.

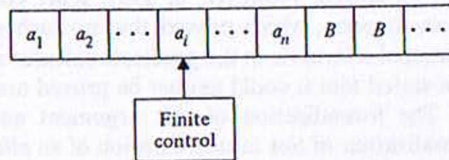

Fig. 7.1 Basic Turing machine.

In one move the Turing machine, depending upon the symbol scanned by the tape head and the state of the finite control,

1) changes state,
2) prints a symbol on the tape cell scanned, replacing what was written there, and
3) moves its head left or right one cell.

Note that the difference between a Turing machine and a two-way finite automaton lies in the former's ability to change symbols on its tape.

Formally, a Turing machine (TM) is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

where

Q is the finite set of *states*,
Γ is the finite set of allowable *tape symbols*,
B, a symbol of Γ, is the *blank*,
Σ, a subset of Γ not including B, is the set of *input symbols*,
δ is the *next move function*, a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ (δ may, however, be undefined for some arguments),
q_0 in Q is the *start state*,
$F \subseteq Q$ is the set of *final states*.

We denote an *instantaneous description* (ID) of the Turing machine M by $\alpha_1 q \alpha_2$. Here q, the current state of M, is in Q; $\alpha_1\alpha_2$ is the string in Γ^* that is the contents of the tape up to the rightmost nonblank symbol or the symbol to the left of the head, whichever is rightmost. (Observe that the blank B may occur in $\alpha_1\alpha_2$.)

We assume that Q and Γ are disjoint to avoid confusion. Finally, the tape head is assumed to be scanning the leftmost symbol of α_2, or if $\alpha_2 = \epsilon$, the head is scanning a blank.

We define a *move* of M as follows. Let $X_1 X_2 \cdots X_{i-1} q X_i \cdots X_n$ be an ID. Suppose $\delta(q, X_i) = (p, Y, L)$, where if $i - 1 = n$, then X_i is taken to be B. If $i = 1$, then there is no next ID, as the tape head is not allowed to fall off the left end of the tape. If $i > 1$, then we write

$$X_1 X_2 \cdots X_{i-1} q X_i \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n. \quad (7.1)$$

However, if any suffix of $X_{i-1} Y X_{i+1} \cdots X_n$ is completely blank, that suffix is deleted in (7.1).

Alternatively, suppose $\delta(q, X_i) = (p, Y, R)$. Then we write:

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n. \quad (7.2)$$

Note that in the case $i - 1 = n$, the string $X_i \cdots X_n$ is empty, and the right side of (7.2) is longer than the left side.

If two ID's are related by $\underset{M}{\vdash}$, we say that the second results from the first by one move. If one ID results from another by some finite number of moves, including zero moves, they are related by the symbol $\overset{*}{\underset{M}{\vdash}}$. We drop the subscript M from $\underset{M}{\vdash}$ or $\overset{*}{\underset{M}{\vdash}}$ when no confusion results.

The *language accepted by* M, denoted $L(M)$, is the set of those words in Σ^* that cause M to enter a final state when placed, justified at the left, on the tape of M, with M in state q_0, and the tape head of M at the leftmost cell. Formally, the language accepted by $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is

$$\{w \mid w \text{ in } \Sigma^* \text{ and } q_0 w \overset{*}{\vdash} \alpha_1 p \alpha_2 \text{ for some } p \text{ in } F, \text{ and } \alpha_1 \text{ and } \alpha_2 \text{ in } \Gamma^*\}.$$

Given a TM recognizing a language L, we assume without loss of generality that the TM halts, i.e., has no next move, whenever the input is accepted. However, for words not accepted, it is possible that the TM will never halt.

Example 7.1 The design of a TM M to accept the language $L = \{0^n 1^n \mid n \geq 1\}$ is given below. Initially, the type of M contains $0^n 1^n$ followed by an infinity of blanks. Repeatedly, M replaces the leftmost 0 by X, moves right to the leftmost 1, replacing it by Y, moves left to find the rightmost X, then moves one cell right to the leftmost 0 and repeats the cycle. If, however, when searching for a 1, M finds a blank instead, then M halts without accepting. If, after changing a 1 to a Y, M finds no more 0's, then M checks that no more 1's remain, accepting if there are none.

Let $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, X, Y, B\}$, and $F = \{q_4\}$. Informally, each state represents a statement or a group of statements in a program. State q_0 is entered initially and also immediately prior to each replacement of a leftmost 0 by an X. State q_1 is used to search right, skipping over 0's and Y's until it finds the leftmost 1. If M finds a 1 it changes it to Y, entering state q_2.

State q_2 searches left for an X and enters state q_0 upon finding it, moving right, to the leftmost 0, as it changes state. As M searches right in state q_1, if a B or X is encountered before a 1, then the input is rejected; either there are too many 0's or the input is not in $\mathbf{0^*1^*}$.

State q_0 has another role. If, after state q_2 finds the rightmost X, there is a Y immediately to its right, then the 0's are exhausted. From q_0, scanning Y, state q_3 is entered to scan over Y's and check that no 1's remain. If the Y's are followed by a B, state q_4 is entered and acceptance occurs; otherwise the string is rejected. The function δ is shown in Fig. 7.2. Figure 7.3 shows the computation of M on input 0011. For example, the first move is explained by the fact that $\delta(q_0, 0) = (q_1, X, R)$; the last move is explained by the fact that $\delta(q_3, B) = (q_4, B, R)$. The reader should simulate M on some rejected inputs such as 001101, 001, and 011.

State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

Fig. 7.2 The function δ.

$$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash$$
$$q_2 X0Y1 \vdash Xq_0 0Y1 \vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash$$
$$XXq_2 YY \vdash Xq_2 XYY \vdash XXq_0 YY \vdash XXYq_3 Y \vdash$$
$$XXYYq_3 \vdash XXYYBq_4$$

Fig. 7.3 A computation of M.

7.3 COMPUTABLE LANGUAGES AND FUNCTIONS

A language that is accepted by a Turing machine is said to be *recursively enumerable* (r.e.). The term "enumerable" derives from the fact that it is precisely these languages whose strings can be enumerated (listed) by a Turing machine. "Recursively" is a mathematical term predating the computer, and its meaning is similar to what the computer scientist would call "recursion." The class of r.e. languages is very broad and properly includes the CFL's.

The class of r.e. languages includes some languages for which we cannot mechanically determine membership. If $L(M)$ is such a language, then any Turing

machine recognizing $L(M)$ must fail to halt on some input not in $L(M)$. If w is in $L(M)$, M eventually halts on input w. However, as long as M is still running on some input, we can never tell whether M will eventually accept if we let it run long enough, or whether M will run forever.

It is convenient to single out a subclass of the r.e. sets, called the *recursive sets*, which are those languages accepted by at least one Turing machine that halts on all inputs (note that halting may or may not be preceded by acceptance). We shall see in Chapter 8 that the recursive sets are a proper subclass of the r.e. sets. Note also that by the algorithm of Fig. 6.8, every CFL is a recursive set.

The Turing machine as a computer of integer functions

In addition to being a language acceptor, the Turing machine may be viewed as a computer of functions from integers to integers. The traditional approach is to represent integers in *unary*; the integer $i \geq 0$ is represented by the string 0^i. If a function has k arguments, $i_1, i_2, \ldots, i_k$, then these integers are initially placed on the tape separated by 1's, as $0^{i_1}10^{i_2}1 \cdots 10^{i_k}$.

If the TM halts (whether or not in an accepting state) with a tape consisting of 0^m for some m, then we say that $f(i_1, i_2, \ldots, i_k) = m$, where f is the function of k arguments computed by this Turing machine. Note that one TM may compute a function of one argument, a different function of two arguments, and so on. Also note that if TM M computes function f of k arguments, then f need not have a value for all different k-tuples of integers $i_1, \ldots, i_k$.

If $f(i_1, \ldots, i_k)$ is defined for all $i_1, \ldots, i_k$, then we say f is a *total recursive function*. A function $f(i_1, \ldots, i_k)$ computed by a Turing machine is called a *partial recursive function*. In a sense, the partial recursive functions are analogous to the r.e. languages, since they are computed by Turing machines that may or may not halt on a given input. The total recursive functions correspond to the recursive languages, since they are computed by TM's that always halt. All common arithmetic functions on integers, such as multiplication, $n!$, $\lceil \log_2 n \rceil$ and 2^{2^n} are total recursive functions.

Example 7.2 *Proper subtraction* $m \dot{-} n$ is defined to be $m - n$ for $m \geq n$, and zero for $m < n$. The TM

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \varnothing)$$

defined below, started with 0^m10^n on its tape, halts with $0^{m \dot{-} n}$ on its tape. M repeatedly replaces its leading 0 by blank, then searches right for a 1 followed by a 0 and changes the 0 to 1. Next, M moves left until it encounters a blank and then repeats the cycle. The repetition ends if

i) Searching right for a 0, M encounters a blank. Then, the n 0's in 0^m10^n have all been changed to 1's, and $n + 1$ of the m 0's have been changed to B. M replaces the $n + 1$ 1's by a 0 and n B's, leaving $m - n$ 0's on its tape.

ii) Beginning the cycle, M cannot find a 0 to change to a blank, because the first m 0's already have been changed. Then $n \geq m$, so $m \dot{-} n = 0$. M replaces all remaining 1's and 0's by B.

The function δ is described below.

1) $\delta(q_0, 0) = (q_1, B, R)$
Begin the cycle. Replace the leading 0 by B.

2) $\delta(q_1, 0) = (q_1, 0, R)$
$\delta(q_1, 1) = (q_2, 1, R)$
Search right, looking for the first 1.

3) $\delta(q_2, 1) = (q_2, 1, R)$
$\delta(q_2, 0) = (q_3, 1, L)$
Search right past 1's until encountering a 0. Change that 0 to 1.

4) $\delta(q_3, 0) = (q_3, 0, L)$
$\delta(q_3, 1) = (q_3, 1, L)$
$\delta(q_3, B) = (q_0, B, R)$
Move left to a blank. Enter state q_0 to repeat the cycle.

5) $\delta(q_2, B) = (q_4, B, L)$
$\delta(q_4, 1) = (q_4, B, L)$
$\delta(q_4, 0) = (q_4, 0, L)$
$\delta(q_4, B) = (q_6, 0, R)$
If in state q_2 a B is encountered before a 0, we have situation (i) described above. Enter state q_4 and move left, changing all 1's to B's until encountering a B. This B is changed back to a 0, state q_6 is entered, and M halts.

6) $\delta(q_0, 1) = (q_5, B, R)$
$\delta(q_5, 0) = (q_5, B, R)$
$\delta(q_5, 1) = (q_5, B, R)$
$\delta(q_5, B) = (q_6, B, R)$
If in state q_0 a 1 is encountered instead of a 0, the first block of 0's has been exhausted, as in situation (ii) above. M enters state q_5 to erase the rest of the tape, then enters q_6 and halts.

A sample computation of M on input 0010 is:

$$q_0 0010 \vdash Bq_1 010 \vdash B0q_1 10 \vdash B01q_2 0 \vdash$$
$$B0q_3 11 \vdash Bq_3 011 \vdash q_3 B011 \vdash Bq_0 011 \vdash$$
$$BBq_1 11 \vdash BB1q_2 1 \vdash BB11q_2 \vdash BB1q_4 1 \vdash$$
$$BBq_4 1 \vdash Bq_4 \quad \vdash B0q_6$$

On input 0100, M behaves as follows:

$$q_0 0100 \vdash Bq_1 100 \vdash B1q_2 00 \vdash Bq_3 110 \vdash$$
$$q_3 B110 \vdash Bq_0 110 \vdash BBq_5 10 \vdash BBBq_5 0 \vdash$$
$$BBBBq_5 \vdash BBBBBq_6$$

7.4 TECHNIQUES FOR TURING MACHINE CONSTRUCTION

Designing Turing [illegible]

	0	1	2	B
q_5	$(q_7, 0, L)$			
q_7		$(q_8, 1, L)$		
q_8	$(q_9, 0, L)$			(q_{10}, B, R)
q_9	$(q_9, 0, L)$			(q_0, B, R)
q_{10}		(q_{11}, B, R)		
q_{11}	(q_{11}, B, R)	(q_{12}, B, R)		

Fig. 7.6 Additional moves for TM performing multiplication.

Note that we could make more than one call to a subroutine if we rewrote the subroutine using a new set of states for each call.

7.5 MODIFICATIONS OF TURING MACHINES

One reason for the acceptance of the Turing machine as a general model of a computation is that the model with which we have been dealing is equivalent to many modified versions that would seem off-hand to have increased computing power. In this section we give informal proofs of some of these equivalence theorems.

Two-way infinite tape

A Turing machine with a two-way infinite tape is denoted by $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, as in the original model. As its name implies, the tape is infinite to the left as well as to the right. We denote an ID of such a device as for the one-way infinite TM. We imagine, however, that there is an infinity of blank cells both to the left and right of the current nonblank portion of the tape.

The relation $\vdash_M$, which relates two ID's if the ID on the right is obtained from the one on the left by a single move, is defined as for the original model with the exception that if $\delta(q, X) = (p, Y, L)$, then $qX\alpha \vdash_M pBY\alpha$ (in the original model, no move could be made), and if $\delta(q, X) = (p, B, R)$, then $qX\alpha \vdash p\alpha$ (in the original, the B would appear to the left of p).

The initial ID is $q_0 w$. While there was a left end to the tape in the original model, there is no left end of the tape for the Turing machine to "fall off," so it can proceed left as far as it wishes. The relation $\overset{*}{\vdash}_M$, as usual, relates two ID's if the one on the right can be obtained from the one on the left by some number of moves.

Theorem 7.1 L is recognized by a Turing machine with a two-way infinite tape if and only if it is recognized by a TM with a one-way infinite tape.

Proof The proof that a TM with a two-way infinite tape can simulate a TM with a one-way infinite tape is easy. The former marks the cell to the left of its initial head position and then simulates the latter. If during the simulation the marked cell is reached, the simulation terminates without acceptance.

Conversely, let $M_2 = (Q_2, \Sigma_2, \Gamma_2, \delta_2, q_2, B, F_2)$ be a TM with a two-way infinite tape. We construct M_1, a Turing machine simulating M_2 and having a tape that is infinite to the right only. M_1 will have two tracks, one to represent the cells of M_2's tape to the right of, and including, the tape cell initially scanned, the other to represent, in reverse order, the cells to the left of the initial cell. The relationship between the tapes of M_2 and M_1 is shown in Fig. 7.7, with the initial cell of M_2 numbered 0, the cells to the right 1, 2, ..., and the cells to the left $-1, -2, \ldots$.

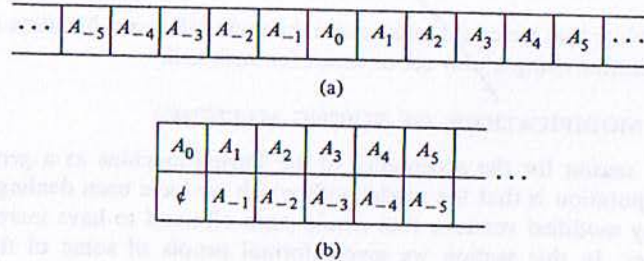

Fig. 7.7 (a) Tape of M_2. (b) Tape of M_1.

The first cell of M_1's tape holds the symbol ¢ in the lower track, indicating that it is the leftmost cell. The finite control of M_1 tells whether M_2 would be scanning a symbol appearing on the upper or on the lower track of M_1.

It should be fairly evident that M_1 can be constructed to simulate M_2, in the sense that while M_2 is to the right of the initial position of its input head, M_1 works on the upper track. While M_2 is to the left of its initial tape head position, M_1 works on its lower track, moving in the direction opposite to the direction in which M_2 moves. The input symbols of M_1 are symbols with a blank on the lower track and an input symbol of M_2 on the upper track. Such a symbol can be identified with the corresponding input symbol of M_2. B is identified with $[B, B]$.

We now give a formal construction of $M_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, B, F_1)$. The states, Q_1, of M_1 are all objects of the form $[q, U]$ or $[q, D]$, where q is in Q_2, plus the symbol q_1. Note that the second component will indicate whether M_1 is working on the upper (U for up) or lower (D for down) track. The tape symbols in Γ_1 are all objects of the form $[X, Y]$, where X and Y are in Γ_2. In addition, Y may be ¢, a symbol not in Γ_2. Σ_1 consists of all symbols $[a, B]$, where a is in Σ_2. F_1 is $\{[q, U], [q, D] \mid q \text{ is in } F_2\}$. We define δ_1 as follows.

1) For each a in $\Sigma_2 \cup \{B\}$,

$$\delta_1(q_1, [a, B]) = ([q, U], [X, ¢], R) \quad \text{if} \quad \delta_2(q_2, a) = (q, X, R).$$

If M_2 moves right on its first move, M_1 prints ¢ in the lower track to mark the end of tape, sets its second component of state to U, and moves right. The first component of M_1's state holds the state of M_2. On the upper track, M_1 prints the symbol X that is printed by M_2.

2) For each a in $\Sigma_2 \cup \{B\}$,

$$\delta_1(q_1, [a, B]) = ([q, D], [X, ¢], R) \quad \text{if} \quad \delta_2(q_2, a) = (q, X, L).$$

If M_2 moves left on its first move, M_1 records the next state of M_2 and the symbol printed by M_2 as in (1) but sets the second component of its state to D and moves right. Again, ¢ is printed in the lower track to mark the left end of the tape.

3) For each $[X, Y]$ in Γ_1, with $Y \neq ¢$, and $A = L$ or R,

$$\delta_1([q, U], [X, Y]) = ([p, U], [Z, Y], A) \quad \text{if} \quad \delta_2(q, X) = (p, Z, A).$$

M_1 simulates M_2 on the upper track.

4) For each $[X, Y]$ in Γ_1, with $Y \neq ¢$,

$$\delta_1([q, D], [X, Y]) = ([p, D], [X, Z], A) \quad \text{if} \quad \delta_2(q, Y) = (p, Z, \bar{A}).$$

Here A is L if $\bar{A}$ is R, and A is R if $\bar{A}$ is L. M_1 simulates M_2 on the lower track of M_1. The direction of head motion of M_1 is opposite to that of M_2.

5) $\delta_1([q, U], [X, ¢]) = \delta_1([q, D], [X, ¢])$

$$= ([p, C], [Y, ¢], R) \quad \text{if} \quad \delta_2(q, X) = (p, Y, A).$$

Here $C = U$ if $A = R$, and $C = D$ if $A = L$. M_1 simulates a move of M_2 on the cell initially scanned by M_2. M_1 next works on the upper or lower track, depending on the direction in which M_2 moves. M_1 will always move right in this situation. □

Multitape Turing machines

A multitape Turing machine is shown in Fig. 7.8. It consists of a finite control with k tape heads and k tapes; each tape is infinite in both directions. On a single move, depending on the state of the finite control and the symbol scanned by each of the tape heads, the machine can:

1) change state;
2) print a new symbol on each of the cells scanned by its tape heads;
3) move each of its tape heads, independently, one cell to the left or right, or keep it stationary.

Initially, the input appears on the first tape, and the other tapes are blank. We shall not define the device more formally, as the formalism is cumbersome and a straightforward generalization of the notation for single-tape TM's.

Theorem 7.2 If a language L is accepted by a multitape Turing machine, it is accepted by a single-tape Turing machine.

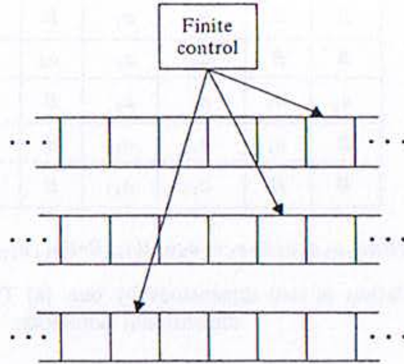

Fig. 7.8 Multitape Turing machine.

Proof Let L be accepted by M_1, a TM with k tapes. We can construct M_2, a one-tape TM with $2k$ tracks, two tracks for each of M_1's tapes. One track records the contents of the corresponding tape of M_1 and the other is blank, except for a marker in the cell that holds the symbol scanned by the corresponding head of M_1. The arrangement is illustrated in Fig. 7.9. The finite control of M_2 stores the state of M_1, along with a count of the number of head markers to the right of M_2's tape head.

Head 1		X				
Tape 1	A_1	A_2	$\cdots$			A_m
Head 2				X		
Tape 2	B_1	B_2	$\cdots$		$\cdots$	B_m
Head 3	X					
Tape 3	C_1	C_2	$\cdots$		$\cdots$	C_m

Fig. 7.9 Simulation of three tapes by one.

Each move of M_1 is simulated by a sweep from left to right and then from right to left by the tape head of M_2. Initially, M_2's head is at the leftmost cell containing a head marker. To simulate a move of M_1, M_2 sweeps right, visiting each of the cells with head markers and recording the symbol scanned by each head of M_1. When M_2 crosses a head marker, it must update the count of head markers to its right. When no more head markers are to the right, M_2 has seen the symbols scanned by each of M_1's heads, so M_2 has enough information to determine the move of M_1. Now M_2 makes a pass left, until it reaches the leftmost head marker. The count of markers to the right enables M_2 to tell when it has gone far enough. As M_2 passes each head marker on the leftward pass, it updates the tape symbol of M_1 "scanned" by that head marker, and it moves the head marker one symbol left or right to simulate the move of M_1. Finally, M_2 changes the state of M_1 recorded in M_2's control to complete the simulation of one move of M_1. If the new state of M_1 is accepting, then M_2 accepts. □

Note that in the first simulation of this section—that of a two-way infinite tape TM by a one-way infinite tape TM, the simulation was move for move. In the present simulation, however, many moves of M_2 are needed to simulate one move of M_1. In fact, since after k moves, the heads of M_1 can be $2k$ cells apart, it takes about $\sum_{i=1}^{k} 2i \approx 2k^2$ moves of M_2 to simulate k moves of M_1. (Actually, $2k$ more moves may be needed to simulate heads moving to the right.) This quadratic slowdown that occurs when we go from a multitape TM to a single tape TM is inherently necessary for certain languages. While we defer a proof to Chapter 12, we shall here give an example of the efficiency of multitape TM's.

Example 7.8 The language $L = \{ww^R \mid w \text{ in } (\mathbf{0} + \mathbf{1})^*\}$ can be recognized on a single-tape TM by moving the tape head back and forth on the input, checking symbols from both ends, and comparing them. The process is similar to that of Example 7.5.

To recognize L with a two-tape TM, the input is copied onto the second tape. The input on one tape is compared with the reversal on the other tape by moving the heads in opposite directions, and the length of the input checked to make sure it is even.

Note that the number of moves used to recognize L by the one-tape machine is approximately the square of the input length, while with a two-tape machine, time proportional to the input length is sufficient.

Nondeterministic Turing machines

A nondeterministic Turing machine is a device with a finite control and a single, one-way infinite tape. For a given state and tape symbol scanned by the tape head, the machine has a finite number of choices for the next move. Each choice consists of a new state, a tape symbol to print, and a direction of head motion. Note that the nondeterministic TM is not permitted to make a move in which the next state is selected from one choice, and the symbol printed and/or direction of head motion are selected from other choices. The nondeterministic TM accepts its input if any sequence of choices of moves leads to an accepting state.

As with the finite automaton, the addition of nondeterminism to the Turing machine does not allow the device to accept new languages. In fact, the combination of nondeterminism with any of the extensions presented or to be presented,

such as two-way infinite or multitape TM's, does not add additional power. We leave these results as exercises, and prove only the basic result regarding the simulation of a nondeterministic TM by a deterministic one.

Theorem 7.3 If L is accepted by a nondeterministic Turing machine, M_1, then L is accepted by some deterministic Turing machine, M_2.

Proof For any state and tape symbol of M_1, there is a finite number of choices for the next move. These can be numbered 1, 2, ... Let r be the maximum number of choices for any state-tape symbol pair. Then any finite sequence of choices can be represented by a sequence of the digits 1 through r. Not all such sequences may represent choices of moves, since there may be fewer than r choices in some situations.

M_2 will have three tapes. The first will hold the input. On the second, M_2 will generate sequences of the digits 1 through r in a systematic manner. Specifically, the sequences will be generated with the shortest appearing first. Sequences of equal length are generated in numerical order.

For each sequence generated on tape 2, M_2 copies the input onto tape 3 and then simulates M_1 on tape 3, using the sequence on tape 2 to dictate the moves of M_1. If M_1 enters an accepting state, M_2 also accepts. If there is a sequence of choices leading to acceptance, it will eventually be generated on tape 2. When simulated, M_2 will accept. But if no sequence of choices of moves of M_1 leads to acceptance, M_2 will not accept. □

Multidimensional Turing machines

Let us consider another modification of the Turing machine that adds no additional power—the multidimensional Turing machine. The device has the usual finite control, but the tape consists of a k-dimensional array of cells infinite in all $2k$ directions, for some fixed k. Depending on the state and symbol scanned, the device changes state, prints a new symbol, and moves its tape head in one of $2k$ directions, either positively or negatively, along one of the k axes. Initially, the input is along one axis, and the head is at the left end of the input.

At any time, only a finite number of rows in any dimension contain nonblank symbols, and these rows each have only a finite number of nonblank symbols. For example, consider the tape configuration of the two-dimensional TM shown in Fig. 7.10(a). Draw a rectangle about the nonblank symbols, as also shown in Fig. 7.10(a). The rectangle can be represented row by row on a single tape, as shown in Fig. 7.10(b). The $*$'s separate the rows. A second track may be used to indicate the position of the two-dimensional TM's tape head.

We shall prove that a one-dimensional TM can simulate a two-dimensional TM, leaving the generalization to more than two dimensions as an exercise.

Theorem 7.4 If L is accepted by a two-dimensional TM M_2, then L is accepted by a one-dimensional TM M_1.

B	B	B	a_1	B	B	B
B	B	a_2	a_3	a_4	a_5	B
a_6	a_7	a_8	a_9	B	a_{10}	B
B	a_{11}	a_{12}	a_{13}	B	a_{14}	a_{15}
B	B	a_{16}	a_{17}	B	B	B

$$**BBBa_1BBB*BBa_2a_3a_4a_5B*a_6a_7a_8a_9Ba_{10}B*Ba_{11}a_{12}a_{13}Ba_{14}a_{15}*BBa_{16}a_{17}BBB**$$

Fig. 7.10 Simulation of two dimensions by one. (a) Two-dimensional tape. (b) One-dimensional simulation.

Proof M_1 represents the tape of M_2 as in Fig. 7.10(b). M_1 will also have a second tape used for purposes we shall describe, and the tapes are two-way infinite. Suppose that M_2 makes a move in which the head does not leave the rectangle already represented by M_1's tape. If the move is horizontal, M_1 simply moves its head marker one cell left or right after printing a new symbol and changing the state of M_2 recorded in M_1's control. If the move is vertical, T_1 uses its second tape to count the number of cells between the tape head position and the $*$ to its left. Then M_1 moves to the $*$ to the right, if the move is down, or the $*$ to the left if the move is up, and puts the tape head marker at the corresponding position in the new *block* (region between $*$'s) by using the count on the second tape.

Now consider the situation when M_2's head moves off the rectangle represented by M_1. If the move is vertical, add a new block of blanks to the left or right, using the second tape to count the current length of blocks. If the move is horizontal, M_1 uses the "shifting over" technique to add a blank at the left or right end of each block, as appropriate. Note that double $*$'s mark the ends of the region used to hold blocks, so M_1 can tell when it has augmented all blocks. After creating room to make the move, M_1 simulates the move of M_2 as described above. □

Multihead Turing machines

A k-head Turing machine has some fixed number, k, of heads. The heads are numbered 1 through k, and a move of the TM depends on the state and on the symbol scanned by each head. In one move, the heads may each move independently left, right, or remain stationary.

Theorem 7.5 If L is accepted by some k-head TM M_1, it is accepted by a one-head TM M_2.

Proof The proof is similar to that of Theorem 7.2 for multitape TM's. M_2 has $k + 1$ tracks on its tape; the last holds the tape of M_1 and the ith holds a marker

indicating the position of the ith tape head for $1 \le i \le k$. The details are left for an exercise. □

Off-line Turing machines

An *off-line* Turing machine is a multitape TM whose input tape is read-only. Usually we surround the input by endmarkers, ¢ on the left and \$ on the right. The Turing machine is not allowed to move the input tape head off the region between ¢ and \$. It should be obvious that the off-line TM is just a special case of the multitape TM, and therefore is no more powerful than any of the models we have considered. Conversely, an off-line TM can simulate any TM M by using one more tape than M. The first thing the off-line TM does is copy its own input onto the extra tape, and it then simulates M as if the extra tape were M's input. The need for off-line TM's will become apparent in Chapter 12, when we consider limiting the amount of storage space to less than the input length.

7.6 CHURCH'S HYPOTHESIS

The assumption that the intuitive notion of "computable function" can be identified with the class of partial recursive functions is known as *Church's hypothesis* or the *Church-Turing thesis*. While we cannot hope to "prove" Church's hypothesis as long as the informal notion of "computable" remains an informal notion, we can give evidence for its reasonableness. As long as our intuitive notion of "computable" places no bound on the number of steps or the amount of storage, it would seem that the partial recursive functions are intuitively computable, although some would argue that a function is not "computable" unless we can bound the computation in advance or at least establish whether or not the computation eventually terminates.

What is less clear is whether the class of partial recursive functions includes all "computable" functions. Logicians have presented many other formalisms such as the λ-calculus, Post systems, and general recursive functions. All have been shown to define the same class of functions, i.e., the partial recursive functions. In addition, abstract computer models, such as the *random access machine* (RAM), also give rise to the partial recursive functions.

The RAM consists of an infinite number of memory words, numbered 0, 1, ..., each of which can hold any integer, and a finite number of arithmetic registers capable of holding any integer. Integers may be decoded into the usual sorts of computer instructions. We shall not define the RAM model more formally, but it should be clear that if we choose a suitable set of instructions, the RAM may simulate any existing computer. The proof that the Turing machine formalism is as powerful as the RAM formalism is given below. Some other formalisms are discussed in the exercises.

Simulation of random access machines by Turing machines

Theorem 7.6 A Turing machine can simulate a RAM, provided that the elementary RAM instructions can themselves be simulated by a TM.

Proof We use a multitape TM M to perform the simulation. One tape of M holds the words of the RAM that have been given values. The tape looks like

$$\#0*v_0\#1*v_1\#10*v_2\#\cdots\#i*v_i\#\cdots,$$

where v_i is the contents, in binary, of the ith word. At all times, there will be some finite number of words of the RAM that have been used, and M needs only to keep a record of values up to the largest numbered word that has been used so far.

The RAM has some finite number of arithmetic registers. M uses one tape to hold each register's contents, one tape to hold the *location counter*, which contains the number of the word from which the next instruction is to be taken, and one tape as a *memory address register* on which the number of a memory word may be placed.

Suppose that the first 10 bits of an instruction denote one of the standard computer operations, such as LOAD, STORE, ADD, and so on, and that the remaining bits denote the address of an operand. While we shall not discuss the details of implementation for all standard computer instructions, an example should make the techniques clear. Suppose the location counter tape of M holds number i in binary. M searches its first tape from the left, looking for $\#i*$. If a blank is encountered before finding $\#i*$, there is no instruction in word i, so the RAM and M halt. If $\#i*$ is found, the bits following $*$ up to the next $\#$ are examined. Suppose the first 10 bits are the code for "ADD to register 2," and the remaining bits are some number j in binary. M adds 1 to i on the location counter tape and copies j onto the memory address tape. Then M searches for $\#j*$ on the first tape, again starting from the left (note that $\#0*$ marks the left end). If $\#j*$ is not found, we assume word j holds 0 and go on to the next instruction of the RAM. If $\#j*v_j\#$ is found, v_j is added to the contents of register 2, which is stored on its own tape. We then repeat the cycle with the next instruction.

Observe that although the RAM simulation used a multitape Turing machine, by Theorem 7.2 a single tape TM would suffice, although the simulation would be more complicated. □

7.7 TURING MACHINES AS ENUMERATORS

We have viewed Turing machines as recognizers of languages and as computers of functions on the nonnegative integers. There is a third useful view of Turing machines, as generating devices. Consider a multitape TM M that uses one tape as an *output tape*, on which a symbol, once written, can never be changed, and whose

tape head never moves left. Suppose also that on the output tape, M writes strings over some alphabet Σ, separated by a marker symbol $\#$. We can define $G(M)$, the *language generated by* M, to be the set of w in Σ^* such that w is eventually printed between a pair of $\#$'s on the output tape.

Note that unless M runs forever, $G(M)$ is finite. Also, we do not require that words be generated in any particular order, or that any particular word be generated only once. If L is $G(M)$ for some TM M, then L is an r.e. set, and conversely. The recursive sets also have a characterization in terms of generators; they are exactly the languages whose words can be generated in order of increasing size. These equivalences will be proved in turn.

Characterization of r.e. sets by generators

Lemma 7.1 If L is $G(M_1)$ for some TM M_1, then L is an r.e. set.

Proof Construct TM M_2 with one more tape than M_1. M_2 simulates M_1 using all but M_2's input tape. Whenever M_1 prints $\#$ on its output tape, M_2 compares its input with the word just generated. If they are the same, M_2 accepts; otherwise M_2 continues to simulate M_1. Clearly M_2 accepts an input x if and only if x is in $G(M_1)$. Thus $L(M_2) = G(M_1)$. □

The converse of Lemma 7.1 is somewhat more difficult. Suppose M_1 is a recognizer for some r.e. set $L \subseteq \Sigma^*$. Our first (and unsuccessful) attempt at designing a generator for L might be to generate the words in Σ^* in some order w_1, $w_2, \ldots$, run M_1 on w_1, and if M_1 accepts, generate w_1. Then run M_1 on w_2, generating w_2 if M_1 accepts, and so on. This method works if M_1 is guaranteed to halt on all inputs. However, as we shall see in Chapter 8, there are languages L that are r.e. but not recursive. If such is the case, we must contend with the possibility that M_1 never halts on some w_i. Then M_2 never considers w_{i+1}, $w_{i+2}, \ldots$, and so cannot generate any of these words, even if M_1 accepts them.

We must therefore avoid simulating M_1 indefinitely on any one word. To do this we fix an order for enumerating words in Σ^*. Next we develop a method of generating all pairs (i, j) of positive integers. The simulation proceeds by generating a pair (i, j) and then simulating M_1 on the ith word, for j steps.

We fix a *canonical order* for Σ^* as follows. List words in order of size, with words of the same size in "numerical order." That is, let $\Sigma = \{a_0, a_1, \ldots, a_{k-1}\}$, and imagine that a_i is the "digit" i in base k. Then the words of length n are the numbers 0 through $k^n - 1$ written in base k. The design of a TM to generate words in canonical order is not hard, and we leave it as an exercise.

Example 7.9 If $\Sigma = \{0, 1\}$, the canonical order is ϵ, 0, 1, 00, 01, 10, 11, 000, 001, ...

Note that the seemingly simpler order in which we generate the shortest representation of 0, 1, 2, ... in base k will not work as we never generate words like $a_0 a_0 a_1$, which have "leading 0's."

Next consider generating pairs (i, j) such that each pair is generated after some finite amount of time. This task is not so easy as it seems. The naive approach, (1, 1), (1, 2), (1, 3), ... never generates any pairs with $i > 1$. Instead, we shall generate pairs in order of the sum $i + j$, and among pairs of equal sum, in order of increasing i. That is, we generate (1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), ... The pair (i, j) is the $\{[(i + j - 1)(i + j - 2)]/2 + i\}$th pair generated. Thus this ordering has the desired property that there is a finite time at which any particular pair (i, j) is generated.

A TM generating pairs (i, j) in this order in binary is easy to design, and we leave its construction to the reader. We shall refer to such a TM as the *pair generator* in the future. Incidentally, the ordering used by the pair generator demonstrates that pairs of integers can be put into one-to-one correspondence with the integers themselves, a seemingly paradoxical result that was discovered by Georg Kantor when he showed that the rationals (which are really the ratios of two integers) are equinumerous with the integers.

Theorem 7.7 A language is r.e. if and only if it is $G(M_2)$ for some TM M_2.

Proof With Lemma 7.1 we have only to show how an r.e. set $L = L(M_1)$ can be generated by a TM M_2. M_2 simulates the pair generator. When (i, j) is generated, M_2 produces the ith word w_i in canonical order and simulates M_1 on w_i for j steps. If M_1 accepts on the jth step (counting the initial ID as step 1), then M_2 generates w_i.

Surely M_2 generates no word not in L. If w is in L, let w be the ith word in canonical order for the alphabet of L, and let M_1 accept w after exactly j moves. As it takes only a finite amount of time for M_2 to generate any particular word in canonical order or to simulate M_1 for any particular number of steps, we know that M_2 will eventually produce the pair (i, j). At that stage, w will be generated by M_2. Thus $G(M_2) = L$. □

Corollary If L is an r.e. set, then there is a generator for L that enumerates each word in L exactly once.

Proof M_2 described above has that property, since it generates w_i only when considering the pair (i, j), where j is exactly the number of steps taken by M_1 to accept w_i. □

Characterization of recursive sets by generators

We shall now show that the recursive sets are precisely those sets whose words can be generated in canonical order.

Lemma 7.2 If L is recursive, then there is a generator for L that prints the words of L in canonical order and prints no other words.

Proof Let $L = L(M_1) \subseteq \Sigma^*$, where M_1 halts on every input. Construct M_2 to generate L as follows. M_2 generates (on a scratch tape) the words in Σ^*, one at a time, in canonical order. After generating some word w, M_2 simulates M_1 on w. If M_1 accepts w, M_2 generates w. Since M_1 is guaranteed to halt, we know that M_2 will finish processing each word after a finite time and will therefore eventually consider each particular word in Σ^*. Clearly M_2 generates L in canonical order. □

The converse of Lemma 7.2, that if L can be generated in canonical order then L is recursive, is also true. However, there is a subtlety of which we should be aware. In Lemma 7.2 we could actually construct M_2 from M_1. However, given a TM M generating L in canonical order, we know a halting TM recognizing L exists, but there is no algorithm to exhibit that TM.

Suppose M_1 generates L in canonical order. The natural thing to do is to construct a TM M_2 that on input w simulates M_1 until M_1 either generates w or a word beyond w in canonical order. In the former case, M_2 accepts w, and in the latter case, M_2 halts without accepting w. However, if L is finite, M_1 may never halt after generating the last word in L, so M_1 may generate neither w nor any word beyond. In this situation M_2 would not halt. This problem arises only when L is finite, even though we know every finite set is accepted by a Turing machine that halts on all inputs. Unfortunately, we cannot determine whether a TM generates a finite set or, if finite, which finite set it is. Thus we know that a halting Turing machine accepting L, the language generated by M_1, always exists, but there is no algorithm to exhibit the Turing machine.

Theorem 7.8 L is recursive if and only if L is generated in canonical order.

Proof The "only if" part was established by Lemma 7.2. For the "if" part, when L is infinite, M_2 described above is a halting Turing machine for L. Clearly, when L is finite, there is a finite automaton accepting L, and thus L can be accepted by a TM that halts on all inputs. Note that in general we cannot exhibit a particular halting TM that accepts L, but the theorem merely states that one such TM exists. □

7.8 RESTRICTED TURING MACHINES EQUIVALENT TO THE BASIC MODEL

In Section 7.5 we considered generalizations of the basic TM model. As we have seen, these generalizations have no more computational power than the basic model. We conclude this chapter by considering some models that at first appear less powerful than the TM but indeed are just as powerful. For the most part, these models will be variations of the pushdown automaton defined in Chapter 5.

EXERCISES

7.1 Design Turing machines to recognize the following languages.

a) $\{0^n1^n0^n \mid n \geq 1\}$.

b) $\{ww^R \mid w \text{ is in } (0 + 1)^*\}$.

c) The set of strings with an equal number of 0's and 1's.

7.2 Design Turing machines to compute the following functions.

a) $\lceil \log_2 n \rceil$ b) $n!$ c) n^2

7.3 Show that if L is accepted by a k-tape, ℓ-dimensional, nondeterministic TM with m heads per tape, then L is accepted by a deterministic TM with one semi-infinite tape and one tape head.

7.4 A *recursive function* is a function defined by a finite set of rules that for various arguments specify the function in terms of variables, nonnegative integer constants, the *successor* (add one) function, the function itself, or an expression built from these by composition of functions. For example, *Ackermann's function* is defined by the rules:

1) $A(0, y) = 1$
2) $A(1, 0) = 2$
3) $A(x, 0) = x + 2$ for $x \geq 2$
4) $A(x + 1, y + 1) = A(A(x, y + 1), y)$

a) Evaluate $A(2, 1)$.

* b) What function of one variable is $A(x, 2)$?

* c) Evaluate $A(4, 3)$.

* **7.5** Give recursive definitions for

a) $n + m$ b) $n \dot{-} m$ c) nm d) $n!$

** **7.6** Show that the class of recursive functions is identical to the class of partial recursive functions.

7.7 A function is *primitive recursive* if it is a finite number of applications of composition and *primitive recursion*† applied to constant 0, the successor function, or a projection function $P_i(x_1, \ldots, x_n) = x_i$.

a) Show that every primitive recursive function is a total recursive function.

** b) Show that Ackermann's function is not primitive recursive.

** c) Show that adding the minimization operator, min $(f(x))$ defined as the least x such that $f(x) = 0$, yields all partial recursive functions.

7.8 Design a Turing machine to enumerate $\{0^n1^n \mid n \geq 1\}$.

** **7.9** Show that every r.e. set is accepted by a TM with only two nonaccepting states and one accepting state.

* **7.10** Complete the proof of Theorem 7.11, that tapes symbols 0 (blank) and 1, with no 1 overprinted by 0, are sufficient for an off-line TM to accept any r.e. language.

7.11 Consider an off-line TM model that cannot write on any tape but has three pebbles that can be placed on the auxiliary tape. Show that the model can accept any r.e. language.

---

† A primitive recursion is a definition of $f(x_1, \ldots, x_n)$ by

$$f(x_1, \ldots, x_n) = \text{if } x_n = 0 \text{ then}$$
$$g(x_1, \ldots, x_{n-1})$$
$$\text{else}$$
$$h(x_1, \ldots, x_n, f(x_1, \ldots, x_{n-1}, x_n - 1))$$

where g and h are primitive recursive functions.