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.

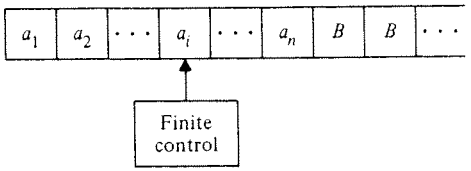

Fig. 7.1 Basic Turing machine.

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

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

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:

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

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 $\vdash_M$, 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 $\vdash_M^*$. We drop the subscript M from $\vdash_M$ or $\vdash_M^*$ 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

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

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 **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.

$$
\begin{aligned}
q_0 0011 &\vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash \\
q_2 X 0 Y 1 &\vdash X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash \\
X X q_2 Y Y &\vdash X q_2 X Y Y \vdash X X q_0 Y Y \vdash X X Y q_3 Y \vdash \\
X X Y Y q_3 &\vdash X X Y Y B q_4
\end{aligned}
$$

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:

$$q_00010 \vdash Bq_1010 \vdash B0q_110 \vdash B01q_20 \vdash$$
$$B0q_311 \vdash Bq_3011 \vdash q_3B011 \vdash Bq_0011 \vdash$$
$$BBq_111 \vdash BB1q_21 \vdash BB11q_2 \vdash BB1q_41 \vdash$$
$$BBq_41 \vdash Bq_4 \vdash B0q_6$$

On input 0100, M behaves as follows:

$$q_00100 \vdash Bq_1100 \vdash B1q_200 \vdash Bq_3110 \vdash$$
$$q_3B110 \vdash Bq_0110 \vdash BBq_510 \vdash BBBq_50 \vdash$$
$$BBBBq_5 \vdash BBBBBq_6$$

7.4 TECHNIQUES FOR TURING MACHINE CONSTRUCTION

Designing Turing machines by writing out a complete set of states and a next-move function is a noticeably unrewarding task. In order to describe complicated Turing machine constructions we need some "higher-level" conceptual tools. In this section we shall discuss the principal ones.

Storage in the finite control

The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other storing a symbol. It should be emphasized that this arrangement is for conceptual purposes only. No modification in the definition of the Turing machine has been made.

Example 7.3 Consider a Turing machine M that looks at the first input symbol, records it in its finite control, and checks that the symbol does not appear elsewhere on its input. Note that M accepts a regular set, but M will serve for demonstration purposes:

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], B, F),$$

where Q is $\{q_0, q_1\} \times \{0, 1, B\}$. That is, Q consists of the pairs $[q_0, 0]$, $[q_0, 1]$, $[q_0, B]$, $[q_1, 0]$, $[q_1, 1]$, and $[q_1, B]$. The set F is $\{[q_1, B]\}$. The intention is that the first component of the state controls the action, while the second component "remembers" a symbol.

We define δ as follows.

1) a) $\delta([q_0, B], 0) = ([q_1, 0], 0, R)$, b) $\delta([q_0, B], 1) = ([q_1, 1], 1, R)$.
Initially, q_0 is the control component of the state, and M moves right. The first component of M's state becomes q_1, and the first symbol seen is stored in the second component.

2) a) $\delta([q_1, 0], 1) = ([q_1, 0], 1, R)$, b) $\delta([q_1, 1], 0) = ([q_1, 1], 0, R)$.
If M has a 0 stored and sees a 1 or vice versa, then M continues to move to the right.

3) a) $\delta([q_1, 0], B) = ([q_1, B], 0, L)$, b) $\delta([q_1, 1], B) = ([q_1, B], 0, L)$.
M enters the final state $[q_1, B]$ if it reaches a blank symbol without having first encountered a second copy of the leftmost symbol.

If M reaches a blank in state $[q_1, 0]$, or $[q_1, 1]$, it accepts. For state $[q_1, 0]$ and symbol 0 or for state $[q_1, 1]$ and symbol 1, δ is not defined. Thus if M encounters the tape symbol stored in its state, M halts without accepting.

In general, we can allow the finite control to have k components, all but one of which store information.

Multiple tracks

We can imagine that the tape of the Turing machine is divided into k tracks, for any finite k. This arrangement is shown in Fig. 7.4, with $k = 3$. The symbols on the tape are considered k-tuples, one component for each track.

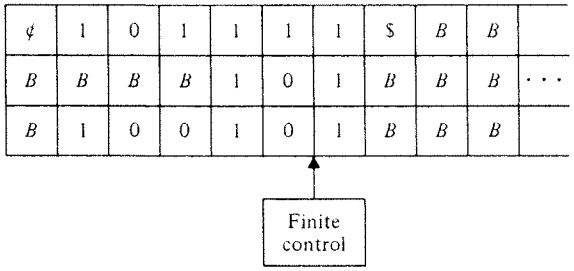

Fig. 7.4 A three-track Turing machine.

Example 7.4 The tape in Fig. 7.4 belongs to a Turing machine that takes a binary input greater than 2, written on the first track, and determines whether it is a prime. The input is surrounded by ¢ and \$ on the first track. Thus, the allowable input symbols are $[¢, B, B]$, $[0, B, B]$, $[1, B, B]$, and $[\$, B, B]$. These symbols can be identified with ¢, 0, 1, and \$, respectively, when viewed as input symbols. The blank symbol can be identified with $[B, B, B]$.

To test if its input is a prime, the TM first writes the number two in binary on the second track and copies the first track onto the third. Then the second track is subtracted, as many times as possible, from the third track, effectively dividing the third track by the second and leaving the remainder.

If the remainder is zero, the number on the first track is not a prime. If the remainder is nonzero, the number on the second track is increased by one. If the second track equals the first, the number on the first track is a prime, because it cannot be divided by any number lying properly between one and itself. If the second is less than the first, the whole operation is repeated for the new number on the second track.

In Fig. 7.4, the TM is testing to determine if 47 is a prime. The TM is dividing by 5; already 5 has been subtracted twice, so 37 appears on the third track.

Checking off symbols

Checking off symbols is a useful trick for visualizing how a TM recognizes languages defined by repeated strings, such as

$$\{ww \mid w \text{ in } \Sigma^*\}, \quad \{wcy \mid w \text{ and } y \text{ in } \Sigma^*, w \neq y\} \quad \text{or} \quad \{ww^R \mid w \text{ in } \Sigma^*\}.$$

It is also useful when lengths of substrings must be compared, such as in the languages

$$\{a^i b^i \mid i \geq 1\} \quad \text{or} \quad \{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}.$$

We introduce an extra track on the tape that holds a blank or $\surd$. The $\surd$ appears when the symbol below it has been considered by the TM in one of its comparisons.

Example 7.5 Consider a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, which recognizes the language $\{wcw \mid w \text{ in } (\mathbf{a} + \mathbf{b})^+\}$. Let

$$Q = \{[q, d] \mid q = q_1, q_2, \ldots, q_9 \text{ and } d = a, b, \text{ or } B\}.$$

The second component of the state is used to store an input symbol,

$$\Sigma = \{[B, d] \mid d = a, b, \text{ or } c\}.$$

The input symbol $[B, d]$ is identified with d. Remember that the two "tracks" are just conceptual tools; that is, $[B, d]$ is just another "name" for d:

$$\Gamma = \{[X, d] \mid X = B \text{ or } \surd \text{ and } d = a, b, c, \text{ or } B\},$$
$$q_0 = [q_1, B], \quad \text{and} \quad F = \{[q_9, B]\};$$

$[B, B]$ is identified with B, the blank symbol. For $d = a$ or b and $e = a$ or b we define δ as follows.

1) $\delta([q_1, B], [B, d]) = ([q_2, d], [\surd, d], R)$.
M checks the symbol scanned on the tape, stores the symbol in the finite control, and moves right.
2) $\delta([q_2, d], [B, e]) = ([q_2, d], [B, e], R)$.
M continues to move right, over unchecked symbols, looking for c.
3) $\delta([q_2, d], [B, c]) = ([q_3, d], [B, c], R)$.
On finding c, M enters a state with first component q_3.

4) $\delta([q_3, d], [\surd, e]) = ([q_3, d], [\surd, e], R)$.
M moves right over checked symbols.

5) $\delta([q_3, d], [B, d]) = ([q_4, B], [\surd, d], L)$.
M encounters an unchecked symbol. If the unchecked symbol matches the symbol stored in the finite control, M checks it and begins moving left. If the symbols disagree, M has no next move and so halts without accepting. M also halts if in state q_3 it reaches $[B, B]$ before finding an unchecked symbol.

6) $\delta([q_4, B], [\surd, d]) = ([q_4, B], [\surd, d], L)$.
M moves left over checked symbols.

7) $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$.
M encounters the symbol c.

8) $\delta([q_5, B], [B, d]) = ([q_6, B], [B, d], L)$.
If the symbol immediately to the left of c is unchecked, M proceeds left to find the rightmost checked symbol.

9) $\delta([q_6, B], [B, d]) = ([q_6, B], [B, d], L)$.
M proceeds left.

10) $\delta([q_6, B], [\surd, d]) = ([q_1, B], [\surd, d], R)$.
M encounters a checked symbol and moves right to pick up another symbol for comparison. The first component of state becomes q_1 again.

11) $\delta([q_5, B], [\surd, d]) = ([q_7, B], [\surd, d], R)$.
M will be in state $[q_5, B]$ immediately after crossing c moving left. (See rule 7.) If a checked symbol appears immediately to the left of c, all symbols to the left of c have been checked. M must test whether all symbols to the right have been checked. If so, they must have compared properly with the symbols to the left of c, so M will accept.

12) $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$.
M moves right over c.

13) $\delta([q_8, B], [\surd, d]) = ([q_8, B], [\surd, d], R)$.
M moves to the right over checked symbols.

14) $\delta([q_8, B], [B, B]) = ([q_9, B], [\surd, B], L)$.
If M finds $[B, B]$, the blank, it halts and accepts. If M finds an unchecked symbol when its first component of state is q_8, it halts without accepting.

Shifting over

A Turing machine can make space on its tape by shifting all nonblank symbols a finite number of cells to the right. To do so, the tape head makes an excursion to the right, repeatedly storing the symbols read in its finite control and replacing them with symbols read from cells to the left. The TM can then return to the vacated cells and print symbols of its choosing. If space is available, it can push blocks of symbols left in a similar manner.

Example 7.6 We construct part of a Turing machine, $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, which may occasionally have a need to shift nonblank symbols two cells to the right. We suppose that M's tape does not contain blanks between nonblanks, so when it reaches a blank it knows to stop the shifting process. Let Q contain states of the form $[q, A_1, A_2]$ for $q = q_1$ or q_2, and A_1 and A_2 in Γ. Let X be a special symbol not used by M except in the shifting process. M starts the shifting process in state $[q_1, B, B]$. The relevant portions of the function δ are as follows.

1) $\delta([q_1, B, B], A_1) = ([q_1, B, A_1], X, R)$ for A_1 in $\Gamma - \{B, X\}$.
M stores the first symbol read in the third component of its state. X is printed on the cell scanned, and M moves to the right.

2) $\delta([q_1, B, A_1], A_2) = ([q_1, A_1, A_2], X, R)$ for A_1 and A_2 in $\Gamma - \{B, X\}$.
M shifts the symbol in the third component to the second component, stores the symbol being read in the third component, prints an X, and moves right.

3) $\delta([q_1, A_1, A_2], A_3) = ([q_1, A_2, A_3], A_1, R)$ for A_1, A_2, and A_3 in $\Gamma - \{B, X\}$.
M now repeatedly reads a symbol A_3, stores it in the third component of state, shifts the symbol previously in the third component, A_2, to the second component, deposits the previous second component, A_1, on the cell scanned, and moves right. Thus a symbol will be deposited two cells to the right of its original position.

4) $\delta([q_1, A_1, A_2], B) = ([q_1, A_2, B], A_1, R)$ for A_1 and A_2 in $\Gamma - \{B, X\}$.
When a blank is seen on the tape, the stored symbols are deposited on the tape.

5) $\delta([q_1, A_1, B], B) = ([q_2, B, B], A_1, L)$.
After all symbols have been deposited, M sets the first component of state to q_2 and moves left to find an X, which marks the rightmost vacated cell.

6) $\delta([q_2, B, B], A) = ([q_2, B, B], A, L)$ for A in $\Gamma - \{B, X\}$.
M moves left until an X is found. When X is found, M transfers to a state that we have assumed exists in Q and resumes its other functions.

Subroutines

As with programs, a "modular" or "top-down" design is facilitated if we use subroutines to define elementary processes. A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms. We shall here

describe only the use of parameterless, nonrecursive subroutines, but even these are quite powerful tools.

The general idea is to write part of a TM program to serve as a subroutine; it will have a designated initial state and a designated return state which temporarily has no move and which will be used to effect a return to the calling routine. To design a TM that "calls" the subroutine, a new set of states for the subroutine is made, and a move from the return state is specified. The call is effected by entering the initial state for the subroutine, and the return is effected by the move from the return state.

Example 7.7 The design of a TM M to implement the total recursive function "multiplication" is given below. M starts with 0^m10^n on its tape and ends with 0^{mn} surrounded by blanks. The general idea is to place a 1 after 0^m10^n and then copy the block of n 0's onto the right end m times, each time erasing one of the m 0's. The result is 10^n10^{mn}. Finally the prefix 10^n1 is erased, leaving 0^{mn}. The heart of the algorithm is a subroutine COPY, which begins in an ID $0^m1q_10^n10^i$ and eventually enters an ID $0^m1q_50^n10^{i+n}$. COPY is defined in Fig. 7.5. In state q_1, on seeing a 0, M changes it to a 2 and enters state q_2. In state q_2, M moves right, to the next blank, deposits the 0, and starts left in state q_3. In state q_3, M moves left to a 2. On reaching a 2, state q_1 is entered and the process repeats until the 1 is encountered, signaling that the copying process is complete. State q_4 is used to convert the 2's back to 0's, and the subroutine halts in q_5.

	0	1	2	B
q_1	$(q_2, 2, R)$	$(q_4, 1, L)$		
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$		$(q_3, 0, L)$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	$(q_1, 2, R)$	
q_4		$(q_5, 1, R)$	$(q_4, 0, L)$	

Fig. 7.5 δ for subroutine COPY.

To complete the program for multiplication, we add states to convert initial ID $q_00^m10^n$ to $B0^{m-1}1q_10^n1$. That is, we need the rules

$$\delta(q_0, 0) = (q_6, B, R),$$
$$\delta(q_6, 0) = (q_6, 0, R),$$
$$\delta(q_6, 1) = (q_1, 1, R).$$

Additional states are needed to convert an ID $B^i0^{m-i}1q_50^n10^{ni}$ to $B^{i+1}0^{m-i-1}1q_10^n10^{ni}$, which restarts COPY, and to check whether $i = m$, that is, all m 0's have been erased. In the case that $i = m$, the leading 10^n1 is erased and the computation halts in state q_{12}. These moves are shown in Fig. 7.6.

	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_0w. 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 $\vdash_M^*$, 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 $-1, -2, \ldots$.

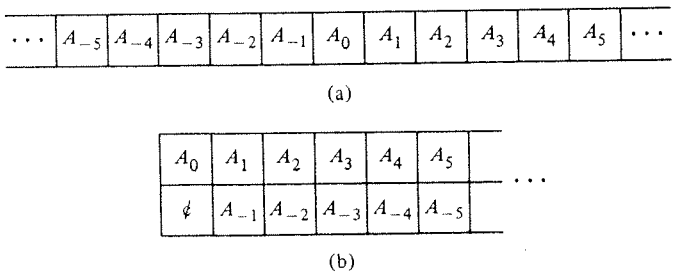

Fig. 7.7 (a) Tape of M_2. (b) Tape of M_1.

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.

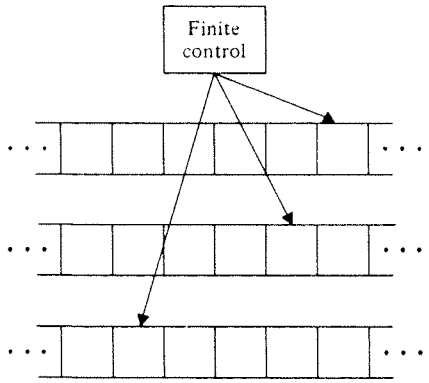

Fig. 7.8 Multitape Turing machine.

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	$\cdots$		$\cdots$	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

$$**BBBa_1\,BBB*BBa_2a_3a_4a_5\,B*a_6a_7a_8a_9Ba_{10}\,B*Ba_{11}a_{12}a_{13}Ba_{14}a_{15}*BBa_{16}a_{17}\,BBB**$$

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 $1 \leq i \leq k$. 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, ..., 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}, ..., 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.

In passing, we note that a pushdown automaton is equivalent to a nondeterministic TM with a read-only input on which the input head cannot move left, plus a storage tape with a rather peculiar restriction on the tape head. Whenever the storage tape head moves left, it must print a blank. Thus the storage tape to the right of the head is always completely blank, and the storage tape is effectively a stack, with the top at the right, rather than the left as in Chapter 5.

Multistack machines

A deterministic two-stack machine is a deterministic Turing machine with a read-only input and two storage tapes. If a head moves left on either tape, a blank is printed on that tape.

Lemma 7.3 An arbitrary single-tape Turing machine can be simulated by a deterministic two-stack machine.

Proof The symbols to the left of the head of the TM being simulated can be stored on one stack, while the symbols on the right of the head can be placed on the other stack. On each stack, symbols closer to the TM's head are placed closer to the top of the stack than symbols farther from the TM's head. □

Counter machines

We can prove a result stronger than Lemma 7.3. It concerns *counter machines*, which are off-line Turing machines whose storage tapes are semi-infinite, and whose tape alphabets contain only two symbols, Z and B (blank). Furthermore, the symbol Z, which serves as a bottom of stack marker, appears initially on the cell scanned by the tape head and may never appear on any other cell. An integer i can be stored by moving the tape head i cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. We can test whether a number is zero by checking whether Z is scanned by the head, but we cannot directly test whether two numbers are equal.

An example of a counter machine is shown in Fig. 7.11; ¢ and \$ are customarily used for end markers on the input. Here Z is the nonblank symbol on each tape. An instantaneous description of a counter machine can be described by the state, the input tape contents, the position of the input head, and the distance of the storage heads from the symbol Z (shown here as d_1 and d_2). We call these distances the *counts* on the tapes. The counter machine, then, can really only store a count on each tape and tell if that count is zero.

Lemma 7.4 A four-counter machine can simulate an arbitrary Turing machine.

Proof From Lemma 7.3, it suffices to show that two counter tapes can simulate one stack. Let a stack have $k - 1$ tape symbols, $Z_1, Z_2, \ldots, Z_{k-1}$. Then we can

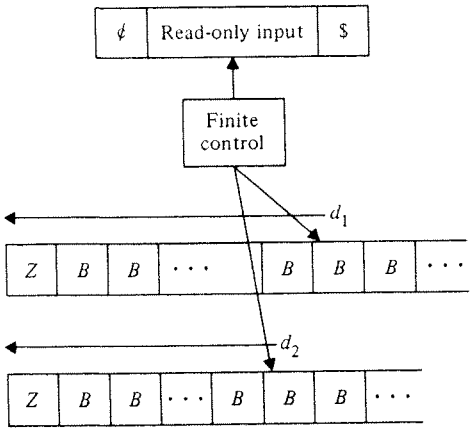

Fig. 7.11 Counter machine.

represent the stack $Z_{i_1} Z_{i_2} \cdots Z_{i_m}$ uniquely by the count in base k

$$j = i_m + ki_{m-1} + k^2 i_{m-2} + \cdots + k^{m-1} i_1. \tag{7.3}$$

Note that not every integer represents a stack; in particular, those whose base-k representation contains the digit 0 do not.

Suppose that the symbol Z_r is pushed onto the top (right end) of the stack $Z_{i_1} Z_{i_2} \cdots Z_{i_m}$. The count associated with $Z_{i_1} Z_{i_2} \cdots Z_{i_m} Z_r$ is $jk + r$. To get this new count, the counter machine repeatedly moves the head of the first counter one cell to the left and the head of the second, k cells to the right. When the head of the first counter reaches the nonblank symbol, the second counter will hold the count jk. It is a simple matter to add r to the count.

If, instead, the top symbol Z_{i_m} of the stack were popped, j should be replaced by $\lfloor j/k \rfloor$, the integer part of j/k. We repeatedly decrement the count on the first counter by k and then add one to the second count. When the first count is zero, the second count will be $\lfloor j/k \rfloor$.

To complete the description of the simulation, we must show how the four-counter machine can tell what symbol is at the top of each stack. If the count j is stored on one counter, the four-counter machine can copy j to another counter, computing j mod k in its finite control. Note that j mod k is i_m if j is given by (7.3). □

Theorem 7.9 A two-counter machine can simulate an arbitrary Turing machine.

Proof By Lemma 7.4, it is sufficient to show how to simulate four counters with two. Let four counters have counts i, j, k, and ℓ. One counter can represent these four by the number $n = 2^i 3^j 5^k 7^\ell$. Since 2, 3, 5, and 7 are primes, i, j, k, and ℓ can be uniquely recovered from n.

To increment i, j, k, or ℓ by 1, we multiply n by 2, 3, 5, or 7, respectively. To do so, if we have another counter set to zero, we can move the head of this counter 2, 3, 5, or 7 cells to the right each time we move the head of the first counter one cell to the left. When the first counter holds zero, the second will hold $2n$, $3n$, $5n$, or $7n$, respectively. To decrement i, j, k, or ℓ by 1, n is, by a similar process, divided by 2, 3, 5, or 7, respectively.

We must also show how the two-counter machine can determine the next move of the four-counter machine. The two-counter machine always scans the same cell of the input tape as the four-counter machine does. The state of the four-counter machine is stored in the finite control of the two-counter machine. Thus, to determine the move of the four-counter machine, the two-counter machine has only to determine which, if any, of i, j, k, and ℓ are 0. By passing n from one counter to the other, the finite control of the two-counter machine can determine if n is divisible by 2, 3, 5, 7, or any product of these. □

Limits on the number of states and symbols

Another way to restrict a TM is to limit the size of the tape alphabet or the number of states. If the tape alphabet, number of tapes, and number of states are all limited, then there is only a finite number of different Turing machines, so the restricted model is less powerful than the original.† If we do not restrict the tape alphabet, then three states and one tape are sufficient to recognize any r.e. set; this result is left as an exercise. We shall, however, prove a result about limited tape alphabets.

Theorem 7.10 If $L \subseteq (\mathbf{0} + \mathbf{1})^*$ and L is r.e., then L is accepted by a one-tape TM with tape alphabet $\{0, 1, B\}$.

Proof Let $L = L(M_1)$, where $M_1 = (Q, \{0, 1\}, \Gamma, \delta, q_0, B, F)$. Suppose Γ has between $2^{k-1} + 1$ and 2^k symbols, so k bits are sufficient to encode any tape symbol of M_1. We may design M_2, with tape alphabet $\{0, 1, B\}$ to simulate M_1. The tape of M_2 will consist of a sequence of codes for symbols of M_1. The finite control of M_2 remembers the state of M_1 and also remembers the position of M_2's tape head, modulo k, so M_2 can know when it is at the beginning of a coded tape symbol of M_1.

At the beginning of the simulation of a move of M_1, the head of M_2 is at the left end of a binary-coded symbol of M_1. M_2 scans the next $k - 1$ symbols to its right, to determine the move of M_1. Then M_1 replaces the symbols scanned to reflect the move of M_1, positions its tape head at the left end of the code for the next symbol scanned by M_1, and changes the state of M_1. If that state is accepting,

† However, there are such restricted Turing machines that are "universal" (see Section 8.3) in the sense that given as input an encoding of a transition function for some TM M and an input w to M, the universal machine accepts if and only if M accepts w. For example, it is known that there is a universal TM with one tape, 5 states, and 7 tape symbols.

M_2 accepts; otherwise, M_2 is ready to simulate the next move of M_1. A special case occurs if M_2 finds its head positioned at a blank when it should be reading a code for a tape symbol of M_1. In this case, M_1 has just moved to a position it has never before reached. M_2 must write the binary code for M_1's blank on the cell scanned and the $k-1$ cells to its right, after which it may simulate a move of M_1 as before.

One important detail is left to be explained. M_2's input is a binary string w in $(\mathbf{0}+\mathbf{1})^*$ representing w itself, rather than a string of coded 0's and 1's representing w. Therefore, before simulating M_1, M_2 must replace w by its code. To do so, M_2 uses the "shifting over" trick, using B for the symbol X described in Section 7.4, where "shifting over" was introduced. For each input symbol, starting with the leftmost, the string to the right of the symbol is shifted $k-1$ places right, and then the symbol and the $k-1$ B's introduced are replaced by the k-bit binary code for the symbol. □

We can apply the same binary coding technique even if the input alphabet is not $\{0, 1\}$. We therefore state the following corollary and leave its proof as an exercise.

Corollary If L is an r.e. set over any alphabet whatsoever, then L is accepted by an off-line TM that has only one tape besides the input, and whose alphabet for that tape is $\{0, 1, B\}$.

Theorem 7.11 Every Turing machine can be simulated by an off-line Turing machine having one storage tape with two symbols, 0 (blank) and 1. The Turing machine can print a 0 or 1 over a 0, but cannot print a 0 over a 1.

Proof We leave this to the reader. The "trick" is to create successive ID's of the original Turing machine on the tape of the new one. Tape symbols are, of course, encoded in binary. Each ID is copied over, making the changes necessary to simulate a move of the old machine.

In addition to the binary encoding of the original symbol, the TM doing the simulating needs cells to indicate the position of the head in the ID being copied, and cells to indicate that the binary representation of a symbol has already been copied. □

EXERCISES

7.1 Design Turing machines to recognize the following languages.

a) $\{0^n1^n0^n \mid n \geq 1\}$.
b) $\{ww^R \mid w \text{ is in } (\mathbf{0}+\mathbf{1})^*\}$.
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

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

where g and h are primitive recursive functions.

BIBLIOGRAPHIC NOTES

The Turing machine is the invention of Turing [1936]. Alternative formulations can be found in Kleene [1936], Church [1936], or Post [1936]. For a discussion of Church's hypothesis, see Kleene [1952], Davis [1958], or Rogers [1967]. Other formalisms equivalent to the partial recursive functions include the λ-calculus (Church [1941]), recursive functions (Kleene [1952]), and Post systems (Post [1943]).

Off-line TM's are discussed by Hartmanis, Lewis, and Stearns [1965]. An important result about multihead TM's—that they can be simulated without loss of time with one head per tape—is found in Hartmanis and Stearns [1965]. The one case not covered by the latter paper, when the multihead machine runs in *real* time (a number of moves proportional to the input length), was handled by Fischer, Meyer, and Rosenberg [1972], and Leong and Seiferas [1977] contains the latest on reducing the complexity of that construction. RAM's were formally considered by Cook and Reckhow [1973].

Theorem 7.9, that two counters can simulate a TM, was proved by Minsky [1961]; the proof given here is taken from Fischer [1966]. Theorem 7.11, on TM's that can only print 1's over 0's, is from Wang [1957]. Exercise 7.9, limiting the number of states, is from Shannon [1956]. In fact, as the latter paper assumes acceptance by halting rather than by final state, it shows that only two states are needed.

A number of texts provide an introduction to the theory of Turing machines and recursive functions. These include Davis [1958, 1965], Rogers [1967], Yasuhara [1971], Jones [1973], Brainerd and Landweber [1974], Hennie [1977], and Machtey and Young [1978].

CHAPTER

8

UNDECIDABILITY

We now consider the classes of recursive and recursively enumerable languages. The most interesting aspect of this study concerns languages whose strings are interpreted as codings of instances of problems. Consider the problem of determining if an arbitrary Turing machine accepts the empty string. This problem may be formulated as a language problem by encoding TM's as strings of 0's and 1's. The set of all strings encoding TM's that accept the empty string is a language that is recursively enumerable but not recursive. From this we conclude that there can be no algorithm to decide which TM's accept the empty string and which do not.

In this chapter we shall show that many questions about TM's, as well as some questions about context-free languages and other formalisms, have no algorithms for their solution. In addition we introduce some fundamental concepts from the theory of recursive functions, including the hierarchy of problems induced by the consideration of Turing machines with "oracles."

8.1 PROBLEMS

Informally we use the word *problem* to refer to a question such as: "Is a given CFG ambiguous?" In the case of the *ambiguity problem*, above, an instance of the problem is a particular CFG. In general, an *instance* of a problem is a list of arguments, one argument for each parameter of the problem. By restricting our attention to problems with yes-no answers and encoding instances of the problem by strings over some finite alphabet, we can transform the question of whether there exists an algorithm for solving a problem to whether or not a particular language is recursive. While it may seem that we are throwing out a lot of impor-

tant problems by looking only at yes-no problems, in fact such is not the case. Many general problems have yes-no versions that are provably just as difficult as the general problem.

Consider the ambiguity problem for CFG's. Call the yes-no version AMB. A more general version of the problem, called FIND, requires producing a word with two or more parses if one exists and answering "no" otherwise. An algorithm for FIND can be used to solve AMB. If FIND produces a word w, then answer "yes"; if FIND answers "no," then answer "no." Conversely, given an algorithm for AMB we can produce an algorithm for FIND. The algorithm first applies AMB to the grammar G. If AMB answers "no" our algorithm answers "no." If AMB answers "yes," the algorithm systematically begins to generate all words over the terminal alphabet of G. As soon as a word w is generated, it is tested to see if it has two or more parse trees. Note that the algorithm does not begin generating words unless G is ambiguous, so some w eventually will be found and printed. Thus we indeed have an algorithm. The portion of the algorithm that tests w for two or more parses is left as an exercise.

The process whereby we construct an algorithm for one problem (such as FIND), using a supposed algorithm for another (AMB), is called a *reduction* (of FIND to AMB). In general, when we reduce problem A to problem B we are showing that B is at least as hard as A. Thus in this case, as in many others, the yes-no problem AMB is no easier than the more general version of the problem. Later we shall show that there is no algorithm for AMB. By the reduction of AMB to FIND we conclude there is no algorithm for FIND either, since the existence of an algorithm for FIND implies the existence of an algorithm for AMB, a contradiction.

One further instructive point concerns the coding of the grammar G. As all Turing machines have a fixed alphabet, we cannot treat the 4-tuple notation $G = (V, T, P, S)$ as the encoding of G without modification. We can encode 4-tuples as binary strings as follows. Let the metasymbols in 4-tuples, that is, the left and right parentheses, brackets, comma and $\rightarrow$, be encoded by 1, 10, 100, ..., 10^5, respectively. Let the ith grammar symbol (in any chosen order) be encoded by 10^{i+5}. In this encoding, we cannot tell the exact symbols used for either terminals or nonterminals. Of course renaming nonterminals does not affect the language generated, so their symbols are not important. Although we ordinarily view the identities of the terminals as important, for this problem the actual symbols used for the terminals is irrelevant, since renaming the terminals does not affect the ambiguity or unambiguity of a grammar.

Decidable and undecidable problems

A problem whose language is recursive is said to be *decidable*. Otherwise, the problem is *undecidable*. That is, a problem is undecidable if there is no algorithm that takes as input an instance of the problem and determines whether the answer to that instance is "yes" or "no."

An unintuitive consequence of the definition of "undecidable" is that problems with only a single instance are trivially decidable. Consider the following problem based on Fermat's conjecture. Is there no solution in positive integers to the equation $x^i + y^i = z^i$ if $i \geq 3$? Note that x, y, z, and i are not parameters but bound variables in the statement of the problem. There is one Turing machine that accepts any input and one that rejects any input. One of these answers Fermat's conjecture correctly, even though we do not know which one. In fact there may not even be a resolution to the conjecture using the axioms of arithmetic. That is, Fermat's conjecture may be true, yet there may be no arithmetic proof of that fact. The possibility (though not the certainty) that this is the case follows from Gödel's incompleteness theorem, which states that any consistent formal system powerful enough to encompass number theory must have statements that are true but not provable within the system.

It should not disturb the reader that a conundrum like Fermat's conjecture is "decidable." The theory of undecidability is concerned with the existence or nonexistence of algorithms for solving problems with an infinity of instances.

8.2 PROPERTIES OF RECURSIVE AND RECURSIVELY ENUMERABLE LANGUAGES

A number of theorems in this chapter are proved by reducing one problem to another. These reductions involve combining several Turing machines to form a composite machine. The state of the composite TM has a component for each individual component machine. Similarly the composite machine has separate tapes for each individual machine. The details of the composite machine are usually tedious and provide no insight. Thus we choose to informally describe the constructions.

Given an algorithm (TM that always halts), we can allow the composite TM to perform one action if the algorithm accepts and another if it does not accept. We could not do this if we were given an arbitrary TM rather than an algorithm, since if the TM did not accept, it might run forever, and the composite machine would never initiate the next task. In pictures, an arrow into a box labeled "start" indicates a start signal. Boxes with no "start" signal are assumed to begin operating when the composite machine does. Algorithms have two outputs, "yes" and "no," which can be used as start signals or as a response by the composite machine. Arbitrary TM's have only a "yes" output, which can be used for the same purposes.

We now turn to some basic closure properties of the classes of recursive and r.e. sets.

Theorem 8.1 The complement of a recursive language is recursive.

Proof Let L be a recursive language and M a Turing machine that halts on all inputs and accepts L. Construct M' from M so that if M enters a final state on input w, then M' halts without accepting. If M halts without accepting, M' enters a

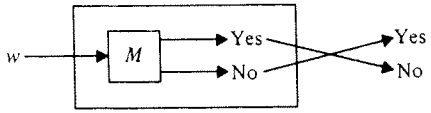

Fig. 8.1 Construction showing that recursive languages are closed under complementation.

final state. Since one of these two events occurs, M' is an algorithm. Clearly $L(M')$ is the complement of L and thus the complement of L is a recursive language. Figure 8.1 pictures the construction of M' from M. □

Theorem 8.2 The union of two recursive languages is recursive. The union of two recursively enumerable languages is recursively enumerable.

Proof Let L_1 and L_2 be recursive languages accepted by algorithms M_1 and M_2. We construct M, which first simulates M_1. If M_1 accepts, then M accepts. If M_1 rejects, then M simulates M_2 and accepts if and only if M_2 accepts. Since both M_1 and M_2 are algorithms, M is guaranteed to halt. Clearly M accepts $L_1 \cup L_2$.

For recursively enumerable languages the above construction does not work, since M_1 may not halt. Instead M can simultaneously simulate M_1 and M_2 on separate tapes. If either accepts, then M accepts. Figure 8.2 shows the two constructions of this theorem. □

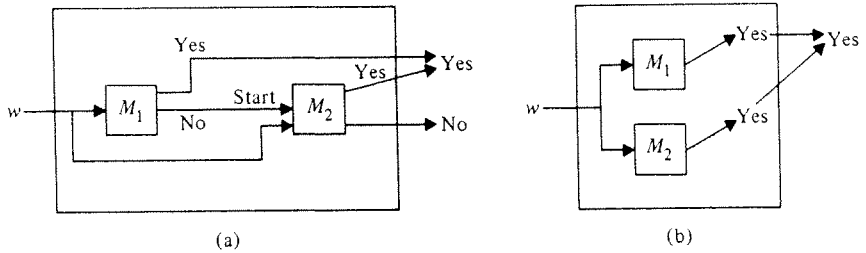

Fig. 8.2 Construction for union.

Theorem 8.3 If a language L and its complement $\bar{L}$ are both recursively enumerable, then L (and hence $\bar{L}$) is recursive.

Proof Let M_1 and M_2 accept L and $\bar{L}$ respectively. Construct M as in Fig. 8.3 to simulate simultaneously M_1 and M_2. M accepts w if M_1 accepts w and rejects w if M_2 accepts w. Since w is in either L or $\bar{L}$, we know that exactly one of M_1 or M_2 will accept. Thus M will always say either "yes" or "no," but will never say both. Note that there is no *a priori* limit on how long it may take before M_1 or M_2 accepts, but it is certain that one or the other will do so. Since M is an algorithm that accepts L, it follows that L is recursive. □

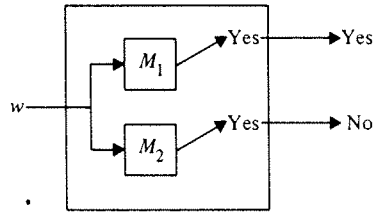

Fig. 8.3 Construction for Theorem 8.3.

Theorems 8.1 and 8.3 have an important consequence. Let L and $\bar{L}$ be a pair of complementary languages. Then either

1) both L and $\bar{L}$ are recursive,
2) neither L nor $\bar{L}$ is r.e., or
3) one of L and $\bar{L}$ is r.e. but not recursive; the other is not r.e.

An important technique for showing a problem undecidable is to show by diagonalization that the complement of the language for that problem is not r.e. Thus case (2) or (3) above must apply. This technique is essential in proving our first problem undecidable. After that, various forms of reductions may be employed to show other problems undecidable.

8.3 UNIVERSAL TURING MACHINES AND AN UNDECIDABLE PROBLEM

We shall now use diagonalization to show a particular problem to be undecidable. The problem is: "Does Turing machine M accept input w?" Here, both M and w are parameters of the problem. In formalizing the problem as a language we shall restrict w to be over alphabet $\{0, 1\}$ and M to have tape alphabet $\{0, 1, B\}$. As the restricted problem is undecidable, the more general problem is surely undecidable as well. We choose to work with the more restricted version to simplify the encoding of problem instances as strings.

Turing machine codes

To begin, we encode Turing machines with restricted alphabets as strings over $\{0, 1\}$. Let

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$

be a Turing machine with input alphabet $\{0, 1\}$ and the blank as the only additional tape symbol. We further assume that $Q = \{q_1, q_2, \ldots, q_n\}$ is the set of states, and that q_2 is the only final state. Theorem 7.10 assures us that if $L \subseteq (\mathbf{0} + \mathbf{1})^*$ is accepted by any TM, then it is accepted by one with alphabet $\{0, 1, B\}$. Also, there

is no need for more than one final state in any TM, since once it accepts it may as well halt.

It is convenient to call symbols 0, 1, and B by the synonyms X_1, X_2, X_3, respectively. We also give directions L and R the synonyms D_1 and D_2, respectively. Then a generic move $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ is encoded by the binary string

$$0^i10^j10^k10^\ell10^m. \tag{8.1}$$

A binary code for Turing machine M is

$$111\ \text{code}_1\ 11\ \text{code}_2\ 11 \cdots 11\ \text{code}_r\ 111, \tag{8.2}$$

where each code_i is a string of the form (8.1), and each move of M is encoded by one of the code_i's. The moves need not be in any particular order, so each TM actually has many codes. Any such code for M will be denoted $\langle M \rangle$.

Every binary string can be interpreted as the code for at most one TM; many binary strings are not the code of any TM. To see that decoding is unique, note that no string of the form (8.1) has two 1's in a row, so the code_i's can be found directly. If a string fails to begin and end with exactly three 1's, has three 1's other than at the end, or has two pair of 1's with other than five blocks of 0's in between, then the string represents no TM.

The pair M and w is represented by a string of the form (8.2) followed by w. Any such string will be denoted $\langle M, w \rangle$.

Example 8.1 Let $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$ have moves:

$$\delta(q_1, 1) = (q_3, 0, R),$$
$$\delta(q_3, 0) = (q_1, 1, R),$$
$$\delta(q_3, 1) = (q_2, 0, R),$$
$$\delta(q_3, B) = (q_3, 1, L).$$

Thus one string denoted by $\langle M, 1011 \rangle$ is

$$11101001000101001100010101010010011$$

$$00010010010100110001000100010010111011$$

Note that many different strings are also codes for the pair $\langle M, 1011 \rangle$, and any of these may be referred to by the notation $\langle M, 1011 \rangle$.

A non-r.e. language

Suppose we have a list of $(\mathbf{0} + \mathbf{1})^*$ in canonical order (see Section 7.7), where w_i is the ith word, and M_j is the TM whose code, as in (8.2) is the integer j written in

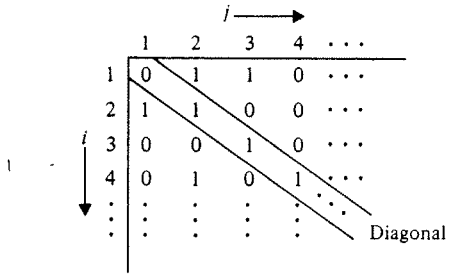

Fig. 8.4 Hypothetical table indicating acceptance of words by TM's.

binary. Imagine an infinite table that tells for all i and j whether w_i is in $L(M_j)$. Figure 8.4 suggests such a table;† 0 means w_i is not in $L(M_j)$ and 1 means it is.

We construct a language L_d by using the diagonal entries of the table to determine membership in L_d. To guarantee that no TM accepts L_d, we insist that w_i is in L_d if and only if the (i, i) entry is 0, that is, if M_i does not accept w_i. Suppose that some TM M_j accepted L_d. Then we are faced with the following contradiction. If w_j is in L_d, then the (j, j) entry is 0, implying that w_j is not in $L(M_j)$ and contradicting $L_d = L(M_j)$. On the other hand, if w_j is not in L_d, then the (j, j) entry is 1, implying that w_j is in $L(M_j)$, which again contradicts $L_d = L(M_j)$. As w_j is either in or not in L_d, we conclude that our assumption, $L_d = L(M_j)$, is false. Thus, no TM in the list accepts L_d, and by Theorem 7.10, no TM whatsoever accepts L_d.

We have thus proved

Lemma 8.1 L_d is not r.e.

The universal language

Define L_u, the "universal language," to be $\{\langle M, w \rangle \mid M \text{ accepts } w\}$. We call L_u "universal" since the question of whether any particular string w in $(\mathbf{0} + \mathbf{1})^*$ is accepted by any particular Turing machine M is equivalent to the question of whether $\langle M', w \rangle$ is in L_u, where M' is the TM with tape alphabet $\{0, 1, B\}$ equivalent to M constructed as in Theorem 7.10.

Theorem 8.4 L_u is recursively enumerable.

Proof We shall exhibit a three-tape TM M_1 accepting L_u. The first tape of M_1 is the input tape, and the input head on that tape is used to look up moves of the TM M when given code $\langle M, w \rangle$ as input. Note that the moves of M are found between the first two blocks of three 1's. The second tape of M_1 will simulate the tape of M.

---

† Actually as all low-numbered Turing machines accept the empty set, the correct portion of the table shown has all 0's.

The alphabet of M is $\{0, 1, B\}$, so each symbol of M's tape can be held in one tape cell of M_1's second tape. Observe that if we did not restrict the alphabet of M, we would have to use many cells of M_1's tape to simulate one of M's cells, but the simulation could be carried out with a little more work. The third tape holds the state of M, with q_i represented by 0^i. The behavior of M_1 is as follows:

1) Check the format of tape 1 to see that it has a prefix of the form (8.2) and that there are no two codes that begin with 0^i10^j1 for the same i and j. Also check that if $0^i10^j10^k10^\ell10^m$ is a code, then $1 \leq j \leq 3$, $1 \leq \ell \leq 3$, and $1 \leq m \leq 2$. Tape 3 can be used as a scratch tape to facilitate the comparison of codes.
2) Initialize tape 2 to contain w, the portion of the input beyond the second block of three 1's. Initialize tape 3 to hold a single 0, representing q_1. All three tape heads are positioned on the leftmost symbols. These symbols may be marked so the heads can find their way back.
3) If tape 3 holds 00, the code for the final state, halt and accept.
4) Let X_j be the symbol currently scanned by tape head 2 and let 0^i be the current contents of tape 3. Scan tape 1 from the left end to the second 111, looking for a substring beginning 110^i10^j1. If no such string is found, halt and reject; M has no next move and has not accepted. If such a code is found, let it be $0^i10^j10^k10^\ell10^m$. Then put 0^k on tape 3, print X_ℓ on the tape cell scanned by head 2 and move that head in direction D_m. Note that we have checked in (1) that $1 \leq \ell \leq 3$ and $1 \leq m \leq 2$. Go to step (3).

It is straightforward to check that M_1 accepts $\langle M, w\rangle$ if and only if M accepts w. It is also true that if M runs forever on w, M_1 will run forever on $\langle M, w\rangle$, and if M halts on w without accepting, M_1 does the same on $\langle M, w\rangle$. □

The existence of M_1 is sufficient to prove Theorem 8.4. However, by Theorems 7.2 and 7.10, we can find a TM with one semi-infinite tape and alphabet $\{0, 1, B\}$ accepting L_u. We call this particular TM M_u, the *universal Turing machine*, since it does the work of any TM with input alphabet $\{0, 1\}$.

By Lemma 8.1, the diagonal language L_d is not r.e., and hence not recursive. Thus by Theorem 8.1, $\bar{L}_d$ is not recursive. Note that $\bar{L}_d = \{w_i \mid M_i \text{ accepts } w_i\}$. We can prove the universal language $L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\}$ not to be recursive by reducing $\bar{L}_d$ to L_u. Thus L_u is an example of a language that is r.e. but not recursive. In fact, $\bar{L}_d$ is another example of such a language.

Theorem 8.5 L_u is not recursive.

Proof Suppose A were an algorithm recognizing L_u. Then we could recognize $\bar{L}_d$ as follows. Given string w in $(\mathbf{0} + \mathbf{1})^*$, determine by an easy calculation the value of i such that $w = w_i$. Integer i in binary is the code for some TM M_i. Feed $\langle M_i, w_i\rangle$ to algorithm A and accept w if and only if M_i accepts w_i. The construction is shown in Fig. 8.5. It is easy to check that the constructed algorithm accepts w if

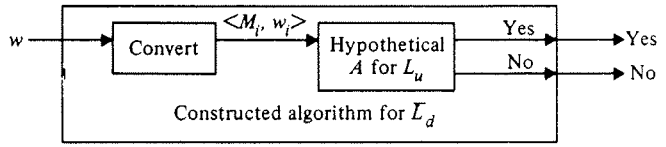

Fig. 8.5 Reduction of $\bar{L}_d$ to L_u.

and only if $w = w_i$ and w_i is in $L(M_i)$. Thus we have an algorithm for $\bar{L}_d$. Since no such algorithm exists, we know our assumption, that algorithm A for L_u exists, is false. Hence L_u is r.e. but not recursive. □

8.4 RICE'S THEOREM AND SOME MORE UNDECIDABLE PROBLEMS

We now have an example of an r.e. language that is not recursive. The associated problem "Does M accept w?" is undecidable, and we can use this fact to show that other problems are undecidable. In this section we shall give several examples of undecidable problems concerning r.e. sets. In the next three sections we shall discuss some undecidable problems taken from outside the realm of TM's.

Example 8.2 Consider the problem: "Is $L(M) \neq \varnothing$?" Let $\langle M\rangle$ denote a code for M as in (8.2). Then define

$$L_{ne} = \{\langle M\rangle \mid L(M) \neq \varnothing\} \quad \text{and} \quad L_e = \{\langle M\rangle \mid L(M) = \varnothing\}.$$

Note that L_e and L_{ne} are complements of one another, since every binary string denotes some TM; those with a bad format denote the TM with no moves. All these strings are in L_e. We claim that L_{ne} is r.e. but not recursive and that L_e is not r.e.

We show that L_{ne} is r.e. by constructing a TM M to recognize codes of TM's that accept nonempty sets. Given input $\langle M_i\rangle$, M nondeterministically guesses a string x accepted by M_i and verifies that M_i does indeed accept x by simulating M_i on input x. This step can also be carried out deterministically if we use the pair generator described in Section 7.7. For pair (j, k) simulate M_i on the jth binary string (in canonical order) for k steps. If M_i accepts, then M accepts $\langle M_i\rangle$.

Now we must show that L_e is not recursive. Suppose it were. Then we could construct an algorithm for L_u, violating Theorem 8.5. Let A be a hypothetical algorithm accepting L_e. There is an algorithm B that, given $\langle M, w\rangle$, constructs a TM M' that accepts $\varnothing$ if M does not accept w and accepts $(\mathbf{0} + \mathbf{1})^*$ if M accepts w. The plan of M' is shown in Fig. 8.6. M' ignores its input x and instead simulates M on input w, accepting if M accepts.

Note that M' is not B. Rather, B is like a compiler that takes $\langle M, w\rangle$ as "source program" and produces M' as "object program." We have described what B must do, but not how it does it. The construction of B is simple. It takes $\langle M, w\rangle$

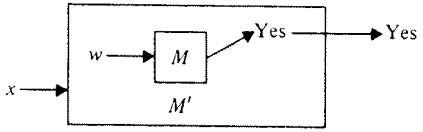

Fig. 8.6 The TM M'.

and isolates w. Say $w = a_1 a_2 \cdots a_n$ is of length n. B creates $n + 3$ states $q_1, q_2, \ldots, q_{n+3}$ with moves

$$\delta(q_1, X) = (q_2, \$, R) \text{ for any } X \text{ (print marker)},$$

$$\delta(q_i, X) = (q_{i+1}, a_{i-1}, R) \text{ for any } X \text{ and } 2 \leq i \leq n + 1 \text{ (print } w\text{)},$$

$$\delta(q_{n+2}, X) = (q_{n+2}, B, R) \text{ for } X \neq B \text{ (erase tape)},$$

$$\delta(q_{n+2}, B) = (q_{n+3}, B, L),$$

$$\delta(q_{n+3}, X) = (q_{n+3}, X, L) \text{ for } X \neq \$ \text{ (find marker)}.$$

Having produced the code for these moves, B then adds $n + 3$ to the indices of the states of M and includes the move

$$\delta(q_{n+3}, \$) = (q_{n+4}, \$, R) \text{ /* start up } M \text{ */}$$

and all the moves of M in its generated TM. The resulting TM has an extra tape symbol \$, but by Theorem 7.10 we may construct M' with tape alphabet $\{0, 1, B\}$, and we may surely make q_2 the accepting state. This step completes the algorithm B, and its output is the desired M' of Fig. 8.6.

Now suppose algorithm A accepting L_e exists. Then we construct an algorithm C for L_u as in Fig. 8.7. If M accepts w, then $L(M') \neq \varnothing$; so A says "no" and C says "yes." If M does not accept w, then $L(M') = \varnothing$, A says "yes," and C says "no." As C does not exist by Theorem 8.5, A cannot exist. Thus, L_e is not recursive. If L_{ne} were recursive, L_e would be also by Theorem 8.1. Thus L_{ne} is r.e. but not recursive. If L_e were r.e., then L_e and L_{ne} would be recursive by Theorem 8.3. Thus L_e is not r.e.

Example 8.3 Consider the language

$$L_r = \{\langle M \rangle \mid L(M) \text{ is recursive}\}$$

and

$$L_{nr} = \{\langle M \rangle \mid L(M) \text{ is not recursive}\}.$$

Note that L_r is not $\{\langle M \rangle \mid M$ halts on all inputs$\}$, although it includes the latter language. A TM M could accept a recursive language even though M itself might loop forever on some words not in $L(M)$; some other TM equivalent to M must always halt, however. We claim neither L_r nor L_{nr} is r.e.

Suppose L_r were r.e. Then we could construct a TM for $\bar{L}_u$ which we know

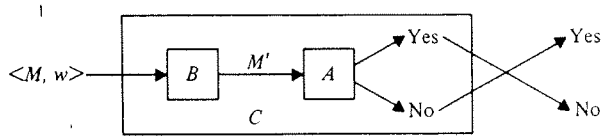

Fig. 8.7 Algorithm constructed for L_u assuming that algorithm A for L_e exists.

does not exist. Let M_r be a TM accepting L_r. We may construct an algorithm A that takes $\langle M, w \rangle$ as input and produces as output a TM M' such that

$$L(M') = \begin{cases} \varnothing & \text{if } M \text{ does not accept } w, \\ L_u & \text{if } M \text{ accepts } w. \end{cases}$$

Note that L_u is not recursive, so M' accepts a recursive language if and only if M does not accept w. The plan of M' is shown in Fig. 8.8. As in the previous example, we have described the output of A. We leave the construction of A to the reader.

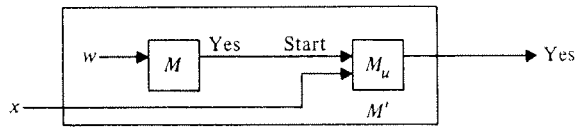

Fig. 8.8 The TM M'.

Given A and M_r we could construct a TM accepting $\bar{L}_u$, shown in Fig. 8.9, which behaves as follows. On input $\langle M, w \rangle$ the TM uses A to produce M', uses M_r to determine if the set accepted by M' is recursive, and accepts if and only if $L(M')$ is recursive. But $L(M')$ is recursive if and only if $L(M') = \varnothing$, which means M does not accept w. Thus the TM of Fig. 8.9 accepts $\langle M, w \rangle$ if and only if $\langle M, w \rangle$ is in $\bar{L}_u$.

Now let us turn to L_{nr}. Suppose we have a TM M_{nr} accepting L_{nr}. Then we may use M_{nr} and an algorithm B, to be constructed by the reader, to accept $\bar{L}_u$. B takes $\langle M, w \rangle$ as input and produces as output a TM M' such that

$$L(M') = \begin{cases} \Sigma^* & \text{if } M \text{ accepts } w, \\ L_u & \text{if } M \text{ does not accept } w. \end{cases}$$

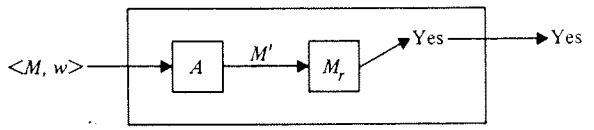

Fig. 8.9 Hypothetical TM for $\bar{L}_u$.

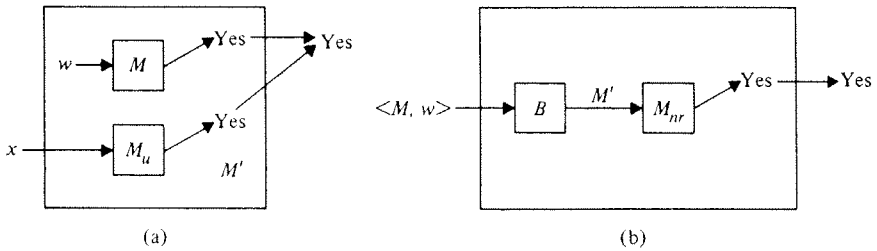

Fig. 8.10 Constructions used in proof that L_{nr} is not r.e. (a) M'. (b) TM for $\bar{L}_u$.

Thus M' accepts a recursive language if and only if M accepts w. M', which B must produce, is shown in Fig. 8.10(a), and a TM to accept $\bar{L}_u$ given B and M_{nr}, is shown in Fig. 8.10(b). The TM of Fig. 8.10(b) accepts $\langle M, w\rangle$ if and only if $L(M')$ is not recursive, or equivalently, if and only if M does not accept w. That is, the TM accepts $\langle M, w\rangle$ if and only if $\langle M, w\rangle$ is in $\bar{L}_u$. Since we have already shown that no such TM exists, the assumption that M_{nr} exists is false. We conclude that L_{nr} is not r.e.

Rice's Theorem for recursive index sets

The above examples show that we cannot decide if the set accepted by a Turing machine is empty or recursive. The technique of proof can also be used to show that we cannot decide if the set accepted is finite, infinite, regular, context free, has an even number of strings, or satisfies many other predicates. What then can we decide about the set accepted by a TM? Only the trivial predicates, such as "Does the TM accept an r.e. set?," which are either true for all TM's or false for all TM's.

In what follows we shall discuss languages that represent properties of r.e. languages. That is, the languages are sets of TM codes such that membership of $\langle M\rangle$ in the language depends only on $L(M)$, not on M itself. Later we shall consider languages of TM codes that depend on the TM itself, such as "M has 27 states," which may be satisfied for some but not all of the TM's accepting a given language.

Let $\mathscr{S}$ be a set of r.e. languages, each a subset of $(\mathbf{0}+\mathbf{1})^*$. $\mathscr{S}$ is said to be a *property of the r.e. languages*. A set L has property $\mathscr{S}$ if L is an element of $\mathscr{S}$. For example, the property of being infinite is $\{L \mid L \text{ is infinite}\}$. $\mathscr{S}$ is a *trivial* property if $\mathscr{S}$ is empty or $\mathscr{S}$ consists of all r.e. languages. Let $L_{\mathscr{S}}$ be the set $\{\langle M\rangle \mid L(M) \text{ is in } \mathscr{S}\}$.

Theorem 8.6 (*Rice's Theorem*) Any nontrivial property $\mathscr{S}$ of the r.e. languages is undecidable.

Proof Without loss of generality assume that $\varnothing$ is not in $\mathscr{S}$ (otherwise consider $\bar{\mathscr{S}}$). Since $\mathscr{S}$ is nontrivial, there exists L with property $\mathscr{S}$. Let M_L be a TM

accepting L. Suppose $\mathscr{S}$ were decidable. Then there exists an algorithm $M_{\mathscr{S}}$ accepting $L_{\mathscr{S}}$. We use M_L and $M_{\mathscr{S}}$ to construct an algorithm for L_u as follows. First construct an algorithm A that takes $\langle M, w\rangle$ as input and produces $\langle M'\rangle$ as output, where $L(M')$ is in $\mathscr{S}$ if and only if M accepts w ($\langle M, w\rangle$ is in L_u).

The design of M' is shown in Fig. 8.11. First M' ignores its input and simulates M on w. If M does not accept w, then M' does not accept x. If M accepts w, then M' simulates M_L on x, accepting x if and only if M_L accepts x. Thus M' either accepts $\varnothing$ or L depending on whether M accepts w.

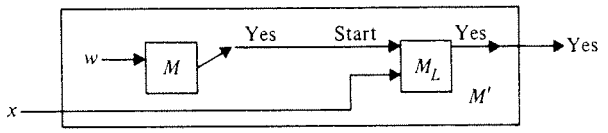

Fig. 8.11 M' used in Rice's theorem.

We may use the hypothetical $M_{\mathscr{S}}$ to determine if $L(M')$ is in $\mathscr{S}$. Since $L(M')$ is in $\mathscr{S}$ if and only if $\langle M, w\rangle$ is in L_u, we have an algorithm for ~~recognizing~~ L_u, a contradiction. Thus $\mathscr{S}$ must be undecidable. Note how this proof generalizes Example 8.2. □

Theorem 8.6 has a great variety of consequences, some of which are summarized in the following corollary.

Corollary The following properties of r.e. sets are not decidable:

a) emptiness,
b) finiteness,
c) regularity,
d) context-freedom.

Rice's Theorem for recursively enumerable index sets

The condition under which a set $L_{\mathscr{S}}$ is r.e. is far more complicated. We shall show that $L_{\mathscr{S}}$ is r.e. if and only if $\mathscr{S}$ satisfies the following three conditions.

1) If L is in $\mathscr{S}$ and $L \subseteq L'$, for some r.e. L', then L' is in $\mathscr{S}$ (the *containment property*).
2) If L is an infinite language in $\mathscr{S}$, then there is a finite subset of L in $\mathscr{S}$.
3) The set of finite languages in $\mathscr{S}$ is *enumerable*, in the sense that there is a Turing machine that generates the (possibly) infinite string $\text{code}_1\#\text{code}_2\#\ \ldots$, where code_i is a code for the ith finite language in $\mathscr{S}$ (in

any order). The code for the finite language $\{w_1, w_2, \ldots, w_n\}$ is just $w_1, w_2, \ldots, w_n$.

We prove this characterization with a series of lemmas.

Lemma 8.2 If $\mathscr{S}$ does not have the containment property, then $L_{\mathscr{S}}$ is not r.e.

Proof We generalize the proof that L_{nr} is not r.e. Let L_1 be in $\mathscr{S}$, $L_1 \subseteq L_2$, and let L_2 not be in $\mathscr{S}$. [For the case where $\mathscr{S}$ was the nonrecursive sets, we chose $L_1 = L_u$ and $L_2 = (\mathbf{0} + \mathbf{1})^*$.] Construct algorithm A that takes as input $\langle M, w\rangle$ and produces as output TM M' with the behavior shown in Fig. 8.12, where M_1 and M_2 accept L_1 and L_2, respectively. If M accepts w, then M_2 is started, and M' accepts x whenever x is in either L_1 or L_2. If M does not accept w, then M_2 never starts, so M' accepts x if and only if x is in L_1. As $L_1 \subseteq L_2$,

$$L(M') = \begin{cases} L_2 & \text{if } M \text{ accepts } w, \\ L_1 & \text{if } M \text{ does not accept } w. \end{cases}$$

Thus $L(M')$ is in $\mathscr{S}$ if and only if M does not accept w.

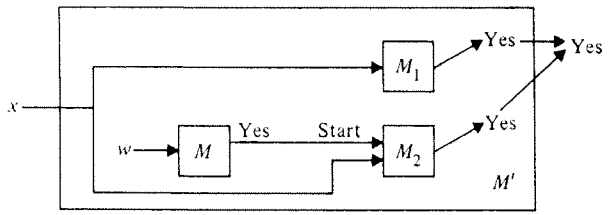

Fig. 8.12 The TM M'.

We again leave it to the reader to design the "compiler" A that takes $\langle M, w\rangle$ as input and connects them with the fixed Turing machines M_1 and M_2 to construct the M' shown in Fig. 8.12. Having constructed A, we can use a TM $M_{\mathscr{S}}$ for $L_{\mathscr{S}}$ to accept $\bar{L}_u$, as shown in Fig. 8.13. This TM accepts $\langle M, w\rangle$ if and only if M' accepts a language in $\mathscr{S}$, or equivalently, if and only if M does not accept w. As such a TM does not exist, we know $M_{\mathscr{S}}$ cannot exist, so $L_{\mathscr{S}}$ is not r.e. □

We now turn to the second property of recursively enumerable index sets.

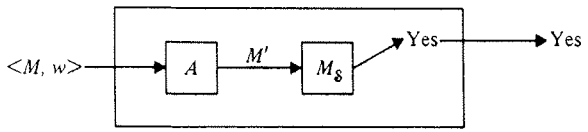

Fig. 8.13 Hypothetical TM to accept $\bar{L}_u$.

Lemma 8.3 If $\mathscr{S}$ has an infinite language L such that no finite subset of L is in $\mathscr{S}$, then $L_{\mathscr{S}}$ is not r.e.

Proof Suppose $L_{\mathscr{S}}$ were r.e. We shall show that $\bar{L}_u$ would be r.e. as follows. Let M_1 be a TM accepting L. Construct algorithm A to take a pair $\langle M, w\rangle$ as input and produce as output a TM M' that accepts L if w is not in $L(M)$ and accepts some finite subset of L otherwise. As shown in Fig. 8.14, M' simulates M_1 on its input x. If M_1 accepts x, then M' simulates M on w for $|x|$ moves. If M fails to accept w after $|x|$ moves, then M' accepts x. We leave the design of algorithm A as an exercise.

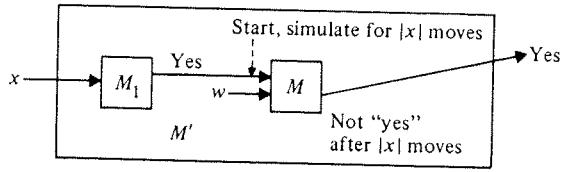

Fig. 8.14 Construction of M'.

If w is in $L(M)$, then M accepts w after some number of moves, say j. Then $L(M') = \{x \mid x \text{ is in } L \text{ and } |x| < j\}$, which is a finite subset of L. If w is not in $L(M)$, then $L(M') = L$. Hence, if M does not accept w, $L(M')$ is in $\mathscr{S}$, and if M accepts w, $L(M')$, being a finite subset of L, is not in $\mathscr{S}$ by the hypothesis of the lemma. An argument that is by now standard proves that if $L_{\mathscr{S}}$ is r.e., so is L_u. Since the latter is not r.e., we conclude the former is not either. □

Finally, consider the third property of r.e. index sets.

Lemma 8.4 If $L_{\mathscr{S}}$ is r.e., then the list of binary codes for the finite sets in $\mathscr{S}$ is enumerable.

Proof We use the pair generator described in Section 7.7. When (i, j) is generated, we treat i as the binary code of a finite set, assuming 0 is the code for comma, 10 the code for zero, and 11 the code for one. We may in a straightforward manner construct a TM $M^{(i)}$ (essentially a finite automaton) that accepts exactly the words in the finite language represented by i. We then simulate the enumerator for $L_{\mathscr{S}}$ for j steps. If it has printed $M^{(i)}$, we print the code for the finite set represented by i, that is, the binary representation of i itself, followed by a delimiter symbol #. In any event, after the simulation we return control to the pair generator, which generates the pair following (i, j). □

Theorem 8.7 $L_{\mathscr{S}}$ is r.e. if and only if

1) If L is in $\mathscr{S}$ and $L \subseteq L'$, for some r.e. L', then L' is in $\mathscr{S}$.
2) If L is an infinite set in $\mathscr{S}$, then there is some finite subset L' of L that is in $\mathscr{S}$.
3) The set of finite languages in $\mathscr{S}$ is enumerable.

Proof The "only if" part is Lemmas 8.2, 8.3, and 8.4. For the "if" part, suppose (1), (2), and (3) hold. We construct a TM M_1 that recognizes $\langle M \rangle$ if and only if $L(M)$ is in $\mathscr{S}$ as follows. M_1 generates pairs (i, j) using the pair generator. In response to (i, j), M_1 simulates M_2, which is an enumerator of the finite sets in $\mathscr{S}$, for i steps. We know M_2 exists by condition (3). Let L_1 be the last set completely printed out by M_2. [If there is no set completely printed, generate the next (i, j) pair.] Then simulate M for j steps on each word in L_1. If M accepts all words in L_1, then M_1 accepts $\langle M \rangle$. If not, M_1 generates the next (i, j)-pair.

We use conditions (1) and (2) to show that $L(M_1) = L_{\mathscr{S}}$. Suppose L is in $L_{\mathscr{S}}$, and let M be any TM with $L(M) = L$. By condition (2), there is a finite $L' \subseteq L$ in $\mathscr{S}$ (take $L' = L$ if L is finite). Let L' be generated after i steps of M_2, and let j be the maximum number of steps taken by M to accept a word in L' (if $L' = \varnothing$, let $j = 1$). Then when M_1 generates (i, j), if not sooner, M_1 will accept $\langle M \rangle$.

Conversely, suppose M_1 accepts $\langle M \rangle$. Then there is some (i, j) such that within j steps M accepts every word in some finite language L' such that M_2 generates L' within its first i steps. Then L' is in $\mathscr{S}$, and $L' \subseteq L(M)$. By condition (1), $L(M)$ is in $\mathscr{S}$, so $\langle M \rangle$ is in $L_{\mathscr{S}}$. We conclude that $L(M_1) = L_{\mathscr{S}}$. □

Theorem 8.7 has a great variety of consequences. We summarize some of them as corollaries and leave others as exercises.

Corollary 1 The following properties of r.e. sets are not r.e.

a) $L = \varnothing$.
b) $L = \Sigma^*$.
c) L is recursive.
d) L is not recursive.
e) L is a *singleton* (has exactly one member).
f) L is a regular set.
g) $L - L_u \neq \varnothing$.

Proof In each case condition (1) is violated, except for (b), where (2) is violated, and (g), where (3) is violated. □

Corollary 2 The following properties of r.e. sets are r.e.

a) $L \neq \varnothing$.
b) L contains at least 10 members.
c) w is in L for some fixed word w.
d) $L \cap L_u \neq \varnothing$.

Problems about Turing machines

Does Theorem 8.6 say that everything about Turing machines is undecidable? The answer is no. That theorem has to do only with properties of the language accepted, not properties of the Turing machine itself. For example, the question "Does a given Turing machine have an even number of states?" is clearly decidable. When dealing with properties of Turing machines themselves, we must use our ingenuity. We give two examples.

Example 8.4 It is undecidable if a Turing machine with alphabet $\{0, 1, B\}$ ever prints three consecutive 1's on its tape. For each Turing machine M_i we construct $\hat{M}_i$, which on blank tape simulates M_i on blank tape. However, $\hat{M}_i$ uses 01 to encode a 0 and 10 to encode a 1. If M_i's tape has a 0 in cell j, $\hat{M}_i$ has 01 in cells $2j - 1$ and $2j$. If M_i changes a symbol, $\hat{M}_i$ changes the corresponding 1 to 0, then the paired 0 to 1. One can easily design $\hat{M}_i$ so that $\hat{M}_i$ never has three consecutive 1's on its tape. Now further modify $\hat{M}_i$ so that if M_i accepts, $\hat{M}_i$ prints three consecutive 1's and halts. Thus $\hat{M}_i$ prints three consecutive 1's if and only if M_i accepts ϵ. By Theorem 8.6, it is undecidable whether a TM accepts ϵ, since the predicate "ϵ is in L" is not trivial. Thus the question of whether an arbitrary Turing machine ever prints three consecutive 1's is undecidable.

Example 8.5 It is decidable whether a single-tape Turing machine started on blank tape scans any cell four or more times. If the Turing machine never scans any cell four or more times, than every *crossing sequence* (sequence of states in which the boundary between cells is crossed, assuming states change before the head moves) is of length at most three. But there is a finite number of distinct crossing sequences of length three or less. Thus either the Turing machine stays within a fixed bounded number of tape cells, in which case finite automaton techniques answer the question, or some crossing sequence repeats. But if some crossing sequence repeats, then the TM moves right with some easily detectable pattern, and the question is again decidable.

8.5 UNDECIDABILITY OF POST'S CORRESPONDENCE PROBLEM

Undecidable problems arise in a variety of areas. In the next three sections we explore some of the more interesting problems in language theory and develop techniques for proving particular problems undecidable. We begin with Post's Correspondence Problem, it being a valuable tool in establishing other problems to be undecidable.

An instance of *Post's Correspondence Problem* (*PCP*) consists of two lists, $A = w_1, \ldots, w_k$ and $B = x_1, \ldots, x_k$, of strings over some alphabet Σ. This instance