CHAPTER

9

Turing Machines

9.1 | DEFINITIONS AND EXAMPLES

The two models of computation we have studied so far involve severe restrictions on either the amount of memory (an FA can remember only its current state) or the way the memory is accessed (a PDA can access only the top stack symbol). Machines implementing these models turn out to be significantly less powerful, at least in principle, than the real computers we are familiar with.

In this chapter we study an abstract machine introduced by the English mathematician Alan Turing (*Proceedings of the London Mathematical Society* 2:230–265, 1936) and for that reason now called a *Turing machine*. Although it may still seem substantially different from a modern electronic computer (which did not exist when Turing formulated the model), the differences have more to do with efficiency, and *how* the computations are carried out, than with the types of computations possible. The work of Turing and his contemporaries provided much of the theoretical foundation for the modern computer.

Turing began by considering a *human* computer (that is, a human who is solving some problem algorithmically using a pencil and paper). He decided that without any loss of generality, the computer could be assumed to operate under these three rules: First, the only things written on the paper are symbols from some fixed finite set; second, each step taken by the computer depends only on the symbol he is currently examining and on his "state of mind" at the time; and third, although his state of mind might change as a result of symbols he has seen or computations he has made, only a finite number of distinct states of mind are possible.

Turing then set out to build an abstract machine that obeys these rules and can duplicate what he took to be the primitive steps carried out by a human computer during a computation:

1. Examining an individual symbol on the paper;
2. Erasing a symbol or replacing it by another;
3. Transferring attention from one part of the paper to another.

Some of these elements should seem familiar. A Turing machine will have a finite alphabet of symbols (actually two alphabets, an input alphabet and a possibly larger alphabet for use during the computation) and a finite set of states, corresponding to the possible "states of mind" of the human computer. Instead of a sheet of paper, Turing specified a linear "tape," which has a left end and is potentially infinite to the right. The tape is marked off into squares, each of which can hold one symbol from the alphabet; if a square has no symbol on it, we say that it contains the *blank* symbol. For convenience, we may think of the squares as being numbered, left-to-right, starting with 0, although this numbering is not part of the official model and it is not necessary to refer to the numbers in describing the operation of the machine. We think of the reading and writing as being done by a *tape head*, which at any time is centered on one square of the tape. In our version of a Turing machine—which is similar although not identical to the one proposed by Turing—a single move is determined by the current state and the current tape symbol, and consists of three parts:

1. Replacing the symbol in the current square by another, possibly different symbol;
2. Moving the tape head one square to the right or left (except that if it is already centered on the leftmost square, it cannot be moved to the left), or leaving it where it is;
3. Moving from the current state to another, possibly different state.

The tape serves as the input device (the input is simply the string, assumed to be finite, of nonblank symbols on the tape originally), the memory available for use during the computation, and the output device (the output is the string of symbols left on the tape at the end of the computation). The most significant difference between the Turing machine and the simpler machines we have studied is that in a Turing machine, processing a string is no longer restricted to a single left-to-right pass through the input. The tape head can move in both directions and erase or modify any symbol it encounters. The machine can examine part of the input, modify it, take time out to execute some computations in a different area of the tape, return to re-examine the input, repeat any of these actions, and perhaps stop the processing before it has looked at all the input.

For similar reasons, we can dispense with one duty previously performed by certain states—that of indicating provisional acceptance of the string read so far. In particular, we can get by with two *final*, or *halting*, states, beyond which the computation need not continue: a state h_a that indicates acceptance and another h_r that indicates rejection. If the machine is intended simply to accept or reject the input string, then it can move to the appropriate halt state once it has enough information to make a decision. If it is supposed to carry out some other computation, the accepting state indicates that the computation has terminated normally; the state h_r can be used to indicate a "crash," arising from some abnormal situation in which the machine cannot carry out its mission as expected. In any case, the computation stops if the Turing maching reaches either of the two halt states. However—and this will turn out to be very important—it is also possible for the computation *not* to stop, and for the Turing machine to continue making moves forever.

Definition 9.1 Turing Machines

A *Turing machine* (TM) is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

Q is a finite set of states, assumed not to contain h_a or h_r, the two *halting* states (the same symbols will be used for the halt states of every TM);

Σ and Γ are finite sets, the *input* and *tape* alphabets, respectively, with $\Sigma \subseteq \Gamma$; Γ is assumed not to contain Δ, the *blank* symbol;

q_0, the initial state, is an element of Q;

$\delta : Q \times (\Gamma \cup \{\Delta\}) \rightarrow (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{R,L,S}\}$ is a partial function (that is, possibly undefined at certain points).

For elements $q \in Q$, $r \in Q \cup \{h_a, h_r\}$, $X, Y \in \Gamma \cup \{\Delta\}$, and $D \in \{\text{R,L,S}\}$, we interpret the formula

$$\delta(q, X) = (r, Y, D)$$

to mean that when T is in state q and the symbol on the current tape square is X, the machine replaces X by Y on that square, changes to state r, and either moves the tape head one square right, moves it one square left (if the tape head is not already on the leftmost square), or leaves it stationary, depending on whether D is R, L, or S, respectively. When r is either h_a or h_r in the formula, we say that T *halts*. Once it has halted, it cannot move further, since δ is not defined at any pair (h_a, X) or (h_r, X).

Finally, we permit the machine to crash by entering the reject state in case it tries to move the tape head off the left end of the tape. This is a way for the machine to halt that is not reflected by the transition function δ. If the tape head is currently on the leftmost square, the current state and tape symbol are q and a, respectively, and $\delta(q, a) = (r, b, \text{L})$, we will say that the machine leaves the tape head where it is, replaces the a by b, and enters the state h_r instead of r.

This terminology and these definitions are not completely standard. In our approach, a TM accepts a string by eventually entering the state h_a after it starts with that input. Sometimes acceptance is defined to mean *halting* (in any halt state), and the only other way the computation is allowed to terminate is by crashing because there is no move possible. In either approach, what is significant is that an observer can see that the TM has stopped its processing and why it has stopped.

Normally a TM begins with an input string $x \in \Sigma^*$ near the beginning of its tape and all other tape squares blank. We do not always insist on this, for reasons to be explained in Section 9.3; however, we do always assume that when a TM begins its operation, there are at most a finite number of nonblank symbols on the tape. It follows that at any stage of a TM's computation, this will still be true. To describe the status of a TM at some point, we must specify the current state, the complete contents of the tape (through the rightmost nonblank symbol), and the current position of the tape head. With this in mind, we represent a *configuration* of the TM by a pair

$$(q, x\underline{a}y)$$

where $q \in Q$, x and y are strings over $\Gamma \cup \{\Delta\}$ (either or both possibly null), a is a symbol in $\Gamma \cup \{\Delta\}$, and the underlined symbol represents the tape head position.

The notation is interpreted to mean that the string xay appears on the tape, beginning in square 0, that the tape head is on the square containing a, and that all squares to the right of y are blank. For a nonnull string w, writing $(q, x\underline{w})$ or $(q, x\underline{w}y)$ will mean that the tape head is positioned at the first symbol of w. If $(q, x\underline{a}y)$ represents a configuration, then y may conceivably end in one or more blanks, and we would also say that $(q, x\underline{a}y\Delta)$ represents the same configuration; usually, however, when we write $(q, x\underline{a}y)$ the string y will either be null or have a nonblank last symbol.

Just as in the case of PDAs, we can trace a sequence of moves by showing the configuration at each step. We write

$$(q, x\underline{a}y) \vdash_T (r, z\underline{b}w)$$

to mean that T passes from the configuration on the left to that on the right in one move, and

$$(q, x\underline{a}y) \vdash_T^* (r, z\underline{b}w)$$

to mean that T passes from the first configuration to the second in zero or more moves. For example, if T is currently in the configuration $(q, aab\underline{a}\Delta a)$ and $\delta(q, a) = (r, \Delta, \text{L})$, we would write

$$(q, aab\underline{a}\Delta a) \vdash_T (r, aa\underline{b}\Delta\Delta a)$$

The notations $\vdash_T$ and $\vdash_T^*$ are usually shortened to $\vdash$ and $\vdash^*$, respectively, as long as there is no ambiguity.

Input is provided to a TM by having the input string on the tape initially, beginning in square 1, and positioning the tape head on square 0, which is blank. The *initial configuration corresponding to input* x is therefore the configuration

$$(q_0, \underline{\Delta}x)$$

Now we can say how a TM accepts a string.

Definition 9.2 Acceptance by a TM

If $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is a Turing machine, and $x \in \Sigma^*$, x is accepted by T if, starting in the initial configuration corresponding to input x, T eventually reaches an accepting configuration. In other words, x is accepted if there exist $y, z \in (\Gamma \cup \{\Delta\})^*$ and $a \in \Gamma \cup \{\Delta\}$ so that

$$(q_0, \underline{\Delta}x) \vdash_T^* (h_a, y\underline{a}z)$$

The *language accepted by* T is the set $L(T)$ of input strings accepted by T.

When a Turing machine processes an input string x, there are three possibilities. The machine can accept the string, by entering the state h_a; it can explicitly reject x, by entering the state h_r; or it can enter an *infinite loop*, so that it never halts but continues moving forever. In either of the first two cases, an observer sees the outcome and can tell whether or not the string is accepted. In the third case, however, although the string will not be accepted, the observer will never find this out—there *is* no outcome, and he is left in suspense. As undesirable as this may seem, we will find that it is sometimes inevitable. In the examples in this chapter, we can construct the machine so that this problem does not arise, and every input string is either accepted or explicitly rejected.

In most simple examples, it will be helpful once again to draw transition diagrams, similar to but more complicated than those for FAs. The move

$$\delta(q, X) = (r, Y, D)$$

(where D is R, L, or S) will be represented as in Figure 9.1.

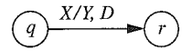


Figure 9.1 | A single Turing machine move.

Our first example should make it clear, if it is not already, that Turing machines are at least as powerful as finite automata.

A TM Accepting $\{a, b\}^*\{aba\}\{a, b\}^*$ — EXAMPLE 9.1

Consider the language

$$L = \{a, b\}^*\{aba\}\{a, b\}^* = \{x \in \{a, b\}^* \mid x \text{ contains the substring } aba\}$$

L is a regular language, and we can draw an FA recognizing L as in Figure 9.2*a*. It is not surprising that constructing a Turing machine to accept L is also easy, and that in fact we can do it so that the transition diagrams look much alike. The TM is illustrated in Figure 9.2*b*. Its input and tape alphabets are both $\{a, b\}$. The initial state does not really correspond to a state in the FA, because the TM does not see any input until it moves the tape head past the initial blank.

Figure 9.2*b* shows explicitly the transitions to the reject state h_r at each point where a blank (the one to the right of the input) is encountered before an occurrence of aba has been found. Figure 9.2*c* shows a simplified diagram, even more similar to the transition diagram for the FA, in which these transitions are omitted. It is often convenient to simplify a diagram

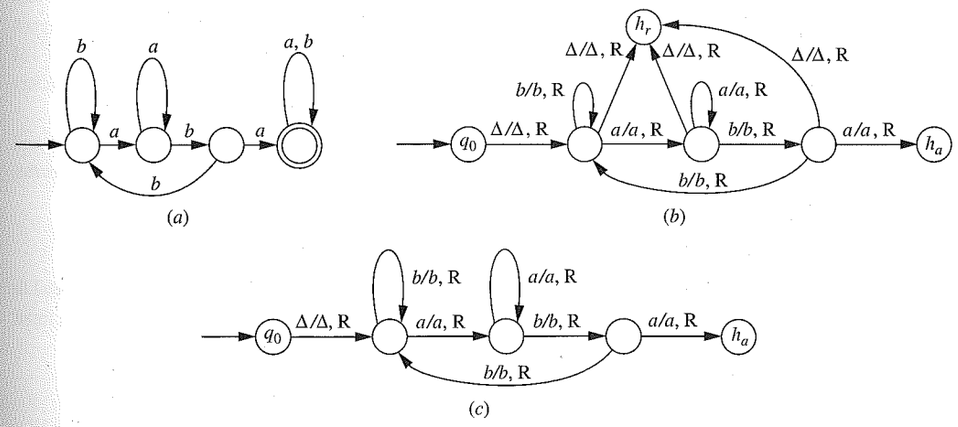


Figure 9.2 | An FA and a TM to accept $\{a, b\}^*\{aba\}\{a, b\}^*$.

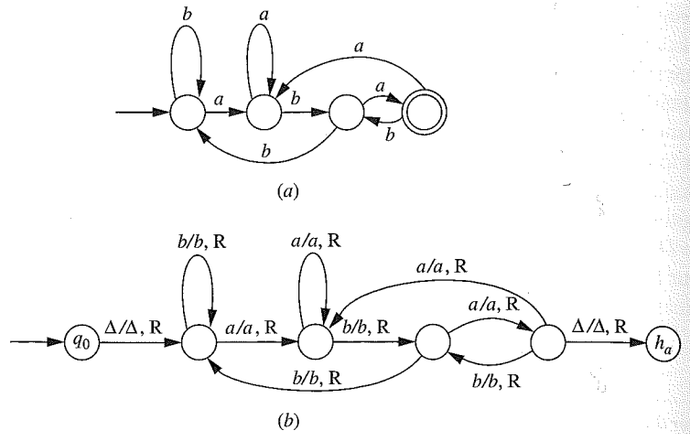


Figure 9.3 |
An FA and a TM to accept $\{a, b\}^*\{aba\}$.

this way; whenever we do, the diagram is to be interpreted as moving to the state h_r for each combination of state and tape symbol for which no move is shown explicitly. What a TM does with the tape head on a final move of this type is essentially arbitrary, since the computation is now over; we may as well assume in general that the tape head moves to the right, as in Figure 9.2*b*. (We will talk later about combining two or more TMs, so that a second one picks up where the first one stops. In this case, what the first machine does on its last move is not arbitrary; however, we will allow such a composite TM to carry out a two-phase computation only if the first phase halts normally in the accept state h_a.)

Because this language is regular, the TM in Figure 9.2*a* or Figure 9.2*b* is able to process input strings the way a finite automaton is forced to, moving the tape head to the right at each step and never changing any tape symbols. Any regular language can be accepted by a TM that mimics an FA this way. As we would expect, this type of processing will not be sufficient to recognize a nonregular language.

Note also that as soon as the TM discovers *aba* on the tape, it enters the state h_a and thereby accepts the entire input string, even though it may not have read all of it. Of course, some TMs must read all the input, even if the languages they accept are regular. For

$$L_1 = \{x \in \{a, b\}^* \mid x \text{ ends with } aba\}$$

for example, an FA and a TM are shown in Figure 9.3. Since the TM moves the tape head to the right on each move, it cannot accept without reading the blank to the right of the last input symbol. As in Figure 9.2*c*, the transitions to the reject state are not shown explicitly.

EXAMPLE 9.2 A TM Accepting *pal*

To see a little more of the power of Turing machines, let us construct a TM to accept the language *pal* of palindromes over $\{a, b\}$. Later in this chapter we will introduce the possibility of nondeterminism in a TM, which would allow us to build a machine simulating the PDA

in Example 7.2 directly. However, the flexibility of TMs allows us to select any algorithm, without restricting ourselves to a specific data structure such as a stack. We can easily formulate a deterministic approach by thinking of how a long string might be checked by hand. You might position your two forefingers at the the two ends. As your eyes jump repeatedly back and forth comparing the two end symbols, your fingers, which are the markers that tell your eyes how far to go, gradually move toward the center. In order to translate this into a TM algorithm, we can use blank squares for the markers at each end. Moving the markers toward the center corresponds to erasing (i.e., changing to blanks) the symbols that have just been tested. The tape head moves repeatedly back and forth, comparing the symbol at one end of the remaining nonblank string to the symbol at the other end. The transition diagram is shown in Figure 9.4. Again the tape alphabet is $\{a, b\}$, the same as the input alphabet. The machine takes the top path each time it finds an a at the beginning and attempts to find a matching a at the end.

If it encounters a b in state q_3, so that it is unable to match the a at the beginning, it enters the reject state h_r. (As in Figure 9.2*c*, this transition is not shown.) Similarly, it rejects from state q_6 if it is unable to match a b at the beginning.

We trace the moves made by the machine for three different input strings: a nonpalindrome, an even-length palindrome, and an odd-length palindrome.

$$\begin{aligned}(q_0, \underline{\Delta}abaa) &\vdash (q_1, \Delta\underline{a}baa) \vdash (q_2, \Delta\Delta\underline{b}aa) \vdash^* (q_2, \Delta\Delta baa\underline{\Delta})\\ &\vdash (q_3, \Delta\Delta ba\underline{a}) \vdash (q_4, \Delta\Delta b\underline{a}) \vdash^* (q_4, \Delta\underline{\Delta}ba)\\ &\vdash (q_1, \Delta\Delta\underline{b}a) \vdash (q_5, \Delta\Delta\Delta\underline{a}) \vdash (q_5, \Delta\Delta\Delta a\underline{\Delta})\\ &\vdash (q_6, \Delta\Delta\Delta\underline{a}) \vdash (h_r, \Delta\Delta\Delta a\underline{\Delta}) \quad \text{(reject)}\end{aligned}$$

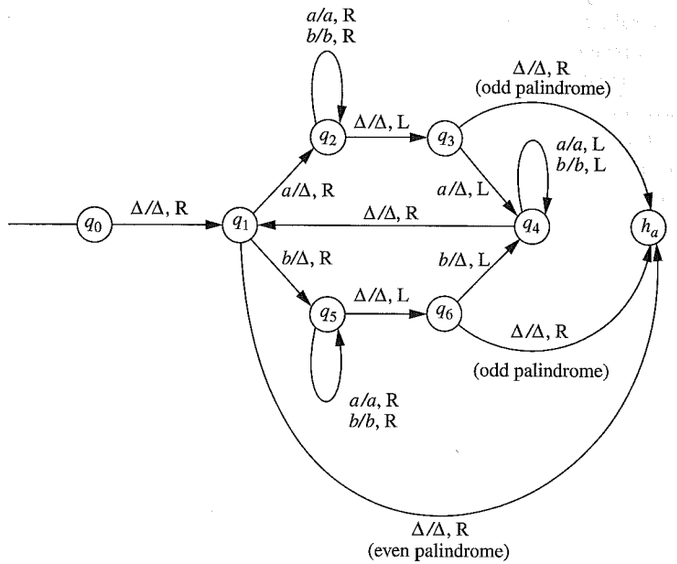


Figure 9.4 |
A TM to accept palindromes over $\{a, b\}$.

$$
\begin{aligned}
(q_0, \underline{\Delta}aa) &\vdash (q_1, \Delta\underline{a}a) \vdash (q_2, \Delta\Delta\underline{a}) \vdash (q_2, \Delta\Delta a\underline{\Delta}) \\
&\vdash (q_3, \Delta\Delta\underline{a}) \vdash (q_4, \Delta\Delta\underline{\Delta}) \vdash (q_1, \Delta\Delta\underline{\Delta}) \\
&\vdash (h_a, \Delta\Delta\Delta\underline{\Delta}) \quad \text{(accept)}
\end{aligned}
$$

$$
\begin{aligned}
(q_0, \underline{\Delta}aba) &\vdash (q_1, \Delta\underline{a}ba) \vdash (q_2, \Delta\Delta\underline{b}a) \vdash^* (q_2, \Delta\Delta ba\underline{\Delta}) \\
&\vdash (q_3, \Delta\Delta b\underline{a}) \vdash (q_4, \Delta\Delta\underline{b}) \vdash (q_4, \Delta\underline{\Delta}b) \\
&\vdash (q_1, \Delta\Delta\underline{b}) \vdash (q_5, \Delta\Delta\Delta\underline{\Delta}) \vdash (q_6, \Delta\Delta\underline{\Delta}) \\
&\vdash (h_a, \Delta\Delta\Delta\underline{\Delta}) \quad \text{(accept)}
\end{aligned}
$$

EXAMPLE 9.3 A TM Accepting $\{ss \mid s \in \{a, b\}^*\}$

For our third example of a Turing machine as a language acceptor, we consider a language that we know from Example 8.2 not to be context-free. Let

$$L = \{ss \mid s \in \{a, b\}^*\}$$

The idea behind the TM will be to separate the processing into two parts: first, finding the middle of the string, and making it easier for the TM to distinguish the symbols in the second half from those in the first half; second, comparing the two halves. We accomplish the first task by working our way in from both ends simultaneously, changing symbols to their uppercase versions as we go. This means that our tape alphabet will include A and B in addition to the input symbols a and b. Once we arrive at the middle—which will happen only if the string is of even length—we may change the symbols in the first half back to their original form. The second part of the processing is to start at the beginning again and, for each lowercase symbol in the first half, compare it to the corresponding uppercase symbol in the second. We keep track of our progress by changing lowercase symbols to uppercase and erasing the matching uppercase symbols.

There are two ways that an input string can be rejected. If its length is odd, the TM will discover this in the first phase. If the string has even length but a symbol in the first half fails to match the corresponding symbol in the second half, the TM will reject the string during the second phase.

The TM suggested by this discussion is shown in Figure 9.5. Again we trace it for three strings: two that illustrate both ways the TM can reject the input, and one that is in the language.

$$
\begin{aligned}
(q_0, \underline{\Delta}aba) &\vdash (q_1, \Delta\underline{a}ba) \vdash (q_2, \Delta A\underline{b}a) \vdash^* (q_2, \Delta Aba\underline{\Delta}) \\
&\vdash (q_3, \Delta Ab\underline{a}) \vdash (q_4, \Delta A\underline{b}A) \vdash (q_4, \Delta\underline{A}bA) \\
&\vdash (q_1, \Delta A\underline{b}a) \vdash (q_2, \Delta AB\underline{A} \quad \vdash (q_3, \Delta A\underline{B}A) \\
&\vdash (h_r, \Delta AB\underline{A}) \quad \text{(reject)}
\end{aligned}
$$

$$
\begin{aligned}
(q_0, \underline{\Delta}abaa) &\vdash (q_1, \Delta\underline{a}baa) \vdash (q_2, \Delta A\underline{b}aa) \vdash^* (q_2, \Delta Abaa\underline{\Delta}) \\
&\vdash (q_3, \Delta Aba\underline{a}) \vdash (q_4, \Delta Ab\underline{a}A) \vdash^* (q_4, \Delta\underline{A}baA) \\
&\vdash (q_1, \Delta A\underline{b}aA) \vdash (q_2, \Delta AB\underline{a}A) \vdash (q_2, \Delta ABa\underline{A}) \\
&\vdash (q_3, \Delta AB\underline{a}A) \vdash (q_4, \Delta A\underline{B}AA) \vdash (q_1, \Delta AB\underline{A}A) \\
&\vdash (q_5, \Delta A\underline{B}AA) \vdash (q_5, \Delta\underline{A}bAA) \vdash (q_5, \underline{\Delta}abAA) \\
&\qquad \text{(first phase completed)}
\end{aligned}
$$

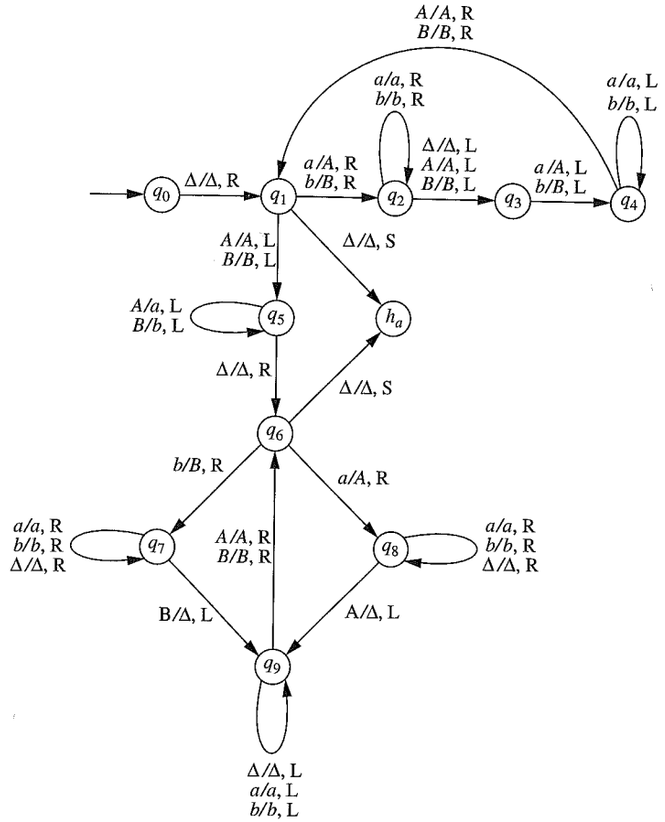


Figure 9.5 |
A Turing machine to accept $\{ss \mid s \in \{a, b\}^*\}$.

$$
\begin{aligned}
&\vdash (q_6, \Delta\underline{a}bAA) \vdash (q_8, \Delta A\underline{b}AA) \vdash (q_8, \Delta Ab\underline{A}A) \\
&\vdash (q_9, \Delta A\underline{b}\Delta A) \vdash (q_9, \Delta\underline{A}b\Delta A) \vdash (q_6, \Delta A\underline{b}\Delta A) \\
&\vdash (q_7, \Delta AB\underline{\Delta}A) \vdash (q_7, \Delta AB\Delta\underline{A}) \vdash (h_r, \Delta AB\Delta A\underline{\Delta}) \quad \text{(reject)}
\end{aligned}
$$

$$
\begin{aligned}
(q_0, \underline{\Delta}abab) &\vdash^* \cdots \\
&\qquad \text{(same as previous case, up to 3rd-from-last move)} \\
&\vdash (q_6, \Delta A\underline{b}\Delta B) \vdash (q_7, \Delta AB\underline{\Delta}B) \vdash (q_7, \Delta AB\Delta\underline{B}) \\
&\vdash (q_9, \Delta AB\underline{\Delta}) \vdash (q_9, \Delta A\underline{B}) \vdash (q_6, \Delta AB\underline{\Delta}) \\
&\vdash (h_a, \Delta AB\underline{\Delta}) \quad \text{(accept)}
\end{aligned}
$$

9.2 COMPUTING A PARTIAL FUNCTION WITH A TURING MACHINE

Any computer program whose purpose is to produce a specified output string for every legal input string can be thought of as computing a function from one set of strings to another. Similarly, a Turing machine T with input alphabet Σ can compute a function f whose domain is a subset of Σ^*. The idea is that for any string x in the domain of f, whenever T starts in the initial configuration corresponding to input x, T will eventually halt with the output string $f(x)$ on the tape.

TMs in Section 9.1, which were used as language acceptors, did their jobs simply by halting in the accepting state or failing to do so, depending on whether the input string was in the language being accepted. The contents of the tape at the end of the computation were not important. In the case of a TM computing a function f, the emphasis is on the output produced for an input string in the domain of f. We might say that for an input string not in the domain, the result of the computation is irrelevant. However, we would like the TM to compute precisely the function f, not some other function with a larger domain. Therefore, we will also specify that for an input string x not in the domain of f, the TM should not accept the input x. It follows that in the process of computing the function, the TM also incidentally accepts a language: the domain of the function.

It will be helpful in subsequent chapters to shift emphasis just slightly, and to talk about *partial* functions on Σ^*, rather than functions on subsets of Σ^*. This is largely a matter of convenience, and there is no real difference except in some of the terminology. A partial function f on Σ^* may be undefined at certain points (the points not in the domain of f); if it happens that f is defined everywhere on Σ^*, we often emphasize the fact by referring to f as a *total* function. In order for a Turing machine to compute f, it is appropriate for the values of f to be strings over the tape alphabet of the machine.

A TM can handle a function of several variables as well. If the input is to represent the k-tuple $(x_1, x_2, \ldots, x_k) \in (\Sigma^*)^k$, the only change required is to relax slightly the rule for the input to a TM, and to allow the initial tape to contain all k strings, separated by blanks.

Definition 9.3 A TM Computing a Function

Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a Turing machine, and let f be a partial function on Σ^* with values in Γ^*. We say that T computes f if for every $x \in \Sigma^*$ at which f is defined,

$$(q_0, \underline{\Delta}x) \vdash_T^* (h_a, \underline{\Delta}f(x))$$

and no other $x \in \Sigma^*$ is accepted by T.

If f is a partial function on $(\Sigma^*)^k$ with values in Γ^*, T computes f if for every k-tuple $(x_1, x_2, \ldots, x_k)$ at which f is defined,

$$(q_0, \underline{\Delta}x_1\Delta x_2\Delta \cdots \Delta x_k) \vdash_T^* (h_a, \underline{\Delta}f(x_1, x_2, \ldots, x_k))$$

and no other input that is a k-tuple of strings is accepted by T. For two alphabets Σ_1 and Σ_2, and a positive integer k, a partial function $f : (\Sigma_1^*)^k \to \Sigma_2^*$ is *Turing-computable*, or simply *computable*, if there is a Turing machine computing f.

It is still not quite correct to say that a TM computes only one function. One reason is that two functions can look exactly alike except for having officially different codomains (see Section 1.3). Another reason is that a TM might be viewed as computing either a function of one variable or a function of more than one. For example, if T computes the function $f : (\Sigma^*)^2 \to \Gamma^*$, then T also computes $f_1 : \Sigma^* \to \Gamma^*$ defined by $f_1(x) = f(x, \Lambda)$. We can say, however, that for any specified k, and any $C \subseteq \Gamma^*$, a given TM computes at most one function of k variables having codomain C.

Numerical functions of numerical arguments can also be computed by Turing machines, once we choose a way of representing the numbers by strings. We will restrict ourselves to natural numbers (nonnegative integers), and we generally use the "unary" representation, in which the integer n is represented by the string $1^n = 11\ldots1$.

Definition 9.4 Computing a Numerical Function

Let $T = (Q, \{1\}, \Gamma, q_0, \delta)$ be a Turing machine. If f is a partial function from $\mathcal{N}$, the set of natural numbers, to itself, T computes f if for every n at which f is defined,

$$(q_0, \underline{\Delta}1^n) \vdash_T^* (h_a, \underline{\Delta}1^{f(n)})$$

and for every other natural number n, T fails to accept the input 1^n. Similarly, if f is a partial function from $\mathcal{N}^k$ to $\mathcal{N}$, T computes f if for every k-tuple $(n_1, n_2, \ldots, n_k)$ at which f is defined,

$$(q_0, \underline{\Delta}1^{n_1}\Delta 1^{n_2}\Delta \cdots \Delta 1^{n_k}) \vdash_T^* (h_a, \underline{\Delta}1^{f(n_1,n_2,\ldots,n_k)})$$

and T fails to accept if the input is any k-tuple at which f is not defined.

Reversing a String — EXAMPLE 9.4

We consider the reverse function

$$rev : \{a, b\}^* \to \{a, b\}^*$$

The TM we construct in Figure 9.6 to compute the function will reverse the input string "in place" by moving from the ends toward the middle, at each step swapping a symbol in the first half with the matching one in the second half. In order to keep track of the progress made so far, symbols will also be changed to uppercase. A pass that starts in state q_1 with a lowercase symbol on the left changes it to the corresponding uppercase symbol and remembers it (by going to state q_2 in the case of an a and q_4 in the case of a b) as it moves the tape head to the right. When the TM arrives at q_3 or q_5, if there is a lowercase symbol on the right corresponding to the one on

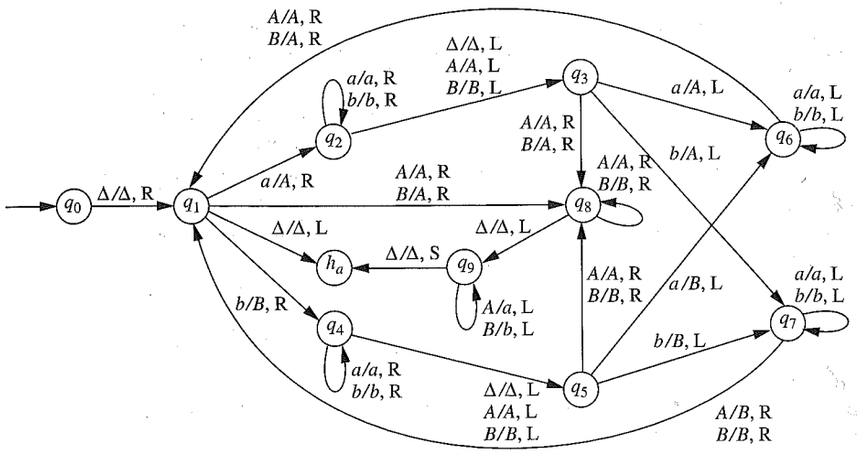


Figure 9.6 |
Reversing a string.

the left, the TM sees it, remembers it (by going to either q_6 or q_7), and changes it to the symbol it remembers from the left. The tape head is moved back to the left, and the first uppercase symbol that is encountered is changed to the (uppercase) symbol that had been on the right. For even-length strings, the last swap will return the TM to q_1, and at that point the absence of any more lowercase symbols sends the machine to q_8 and the final phase of processing. In the case of an odd-length string, the last pass will not be completed normally, because the machine will discover at either q_3 or q_5 that there is no lowercase symbol to swap with the one on the left (which therefore turns out to have been the middle symbol of the string). When the swaps have been completed, all that remains is to move the tape head to the end of the string and make one final pass back to the left, changing all the uppercase symbols back to lowercase.

We trace the TM in Figure 9.6 for the odd-length string *abb* and the even-length string *baba*.

$(q_0, \underline{\Delta}abb) \vdash (q_1, \Delta\underline{a}bb) \vdash (q_2, \Delta A\underline{b}b) \vdash (q_2, \Delta Ab\underline{b})$
$\vdash (q_2, \Delta Abb\underline{\Delta}) \vdash (q_3, \Delta Ab\underline{b}) \vdash (q_7, \Delta A\underline{b}A)$
$\vdash (q_7, \Delta\underline{A}bA) \vdash (q_1, \Delta B\underline{b}A) \vdash (q_4, \Delta BB\underline{A})$
$\vdash (q_5, \Delta B\underline{B}A) \vdash (q_8, \Delta BB\underline{A}) \vdash (q_8, \Delta BBA\underline{\Delta})$
$\vdash (q_9, \Delta BB\underline{A}) \vdash (q_9, \Delta B\underline{B}a) \vdash (q_9, \Delta\underline{B}ba)$
$\vdash (q_9, \underline{\Delta}bba) \vdash (h_a, \underline{\Delta}bba)$

$(q_0, \underline{\Delta}baba) \vdash (q_1, \Delta\underline{b}aba) \vdash (q_4, \Delta B\underline{a}ba) \vdash (q_4, \Delta Ba\underline{b}a)$
$\vdash (q_4, \Delta Baba\underline{\Delta}) \vdash (q_5, \Delta Bab\underline{a}) \vdash (q_6, \Delta Ba\underline{b}B)$
$\vdash (q_6, \Delta B\underline{a}bB) \vdash (q_2, \Delta AA\underline{b}B) \vdash (q_2, \Delta AAb\underline{B})$
$\vdash (q_1, \Delta A\underline{a}bB) \vdash (q_7, \Delta A\underline{A}AB) \vdash (q_1, \Delta AB\underline{A}B)$
$\vdash (q_3, \Delta AA\underline{b}B) \vdash (q_8, \Delta ABAB\underline{\Delta}) \vdash (q_9, \Delta ABA\underline{B})$
$\vdash^* (q_9, \underline{\Delta}abab) \vdash (h_a, \underline{\Delta}abab)$

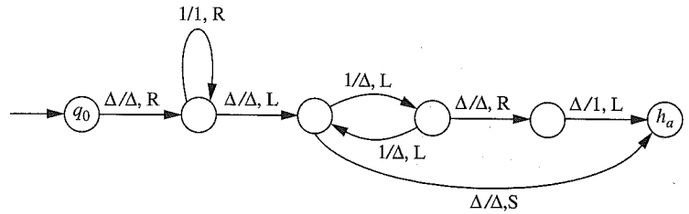


Figure 9.7 |
A Turing machine to compute n mod 2.

n mod 2 — EXAMPLE 9.5

The numerical function that assigns to each natural number n the remainder when n is divided by 2 can be computed by moving to the end of the input string, making a pass from right to left in which the 1's are counted and simultaneously erased, and either leaving a single 1 (if the original number was odd) or leaving nothing. The TM that performs this computation is shown in Figure 9.7.

The Characteristic Function of a Set — EXAMPLE 9.6

For any language $L \subseteq \Sigma^*$, the *characteristic function* of L is the function $\chi_L : \Sigma^* \to \{0, 1\}$ defined by the formula

$$\chi_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise} \end{cases}$$

Computing the function χ_L is therefore similar in one respect to accepting the language L (see Section 9.1); instead of distinguishing between strings in L and strings not in L by accepting or not accepting, the TM accepts every input, and distinguishes between the two types of strings by ending up in the configuration $(h_a, \underline{\Delta}1)$ in one case and the configuration $(h_a, \underline{\Delta}0)$ in the other.

If we have a TM T computing χ_L, we can easily obtain one that accepts L. All we have to do is modify T so that when it leaves output 0, it enters the state h_r instead of h_a. Sometimes it is possible to go the other way; a simple example is the language L of palindromes over $\{a, b\}$ (Example 9.2). A TM accepting L is shown in Figure 9.4, and a TM computing χ_L is shown in Figure 9.8.

It is obtained from the previous one by identifying the places in the transition diagram where the TM might reject, and modifying the TM so that instead of entering the state h_r in those situations, it continues in a way that ends up in state h_a with output 0. For any language L accepted by a TM T that halts on every input string, another TM can be constructed from T that computes χ_L, although the construction may be more complicated than in this example. A TM of either type effectively allows an observer to decide whether a given input string is in L; a "no" answer is produced in one case by the input being rejected and in the other case by output 0.

As we saw in Section 9.1, however, a TM can accept a language L and still leave the question of whether $x \in L$ unanswered for some strings x, by looping forever on those inputs.

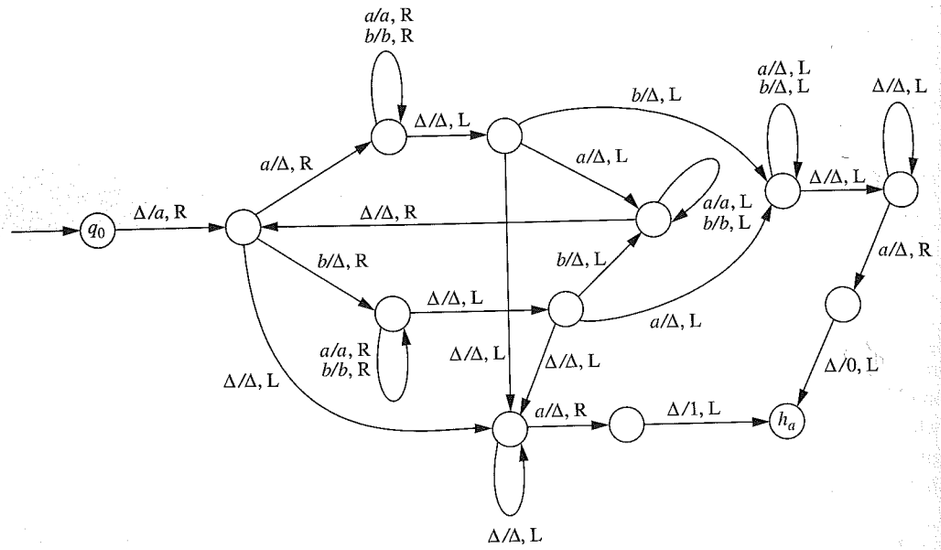


Figure 9.8 |
Computing χ_L for the set of palindromes.

(If we could somehow *see* that the TM was in an infinite loop, we would have the answer; but if we were depending on the TM to tell us, we would wait forever.) In this case, a TM computing the function χ_L would be better, because it would guarantee an answer for every input. Unfortunately, it is no longer clear that such a machine can be obtained from T. We will return to this question in Chapter 10.

9.3 | COMBINING TURING MACHINES

One of the purposes of this chapter is to suggest the power and generality of Turing machines. As you can tell from the examples so far, much of the work that goes on during a TM computation consists of routine, repetitive tasks such as moving the tape head from one side of a string to the other or erasing a portion of the tape. If we were required to describe every TM monolithically, showing all the low-level details, we would quickly reach a practical limit on the complexity of the problems we could solve. The natural way to construct a complicated TM (or any other complicated algorithm or piece of software) is to build it from simpler, reusable components.

In the simplest case, we can construct a composite Turing machine by executing first one TM and then another. If T_1 and T_2 are TMs, with disjoint sets of nonhalting states and transition functions δ_1 and δ_2, respectively, we write T_1T_2 to denote this composite TM. The set of states is the union of the two sets. T_1T_2 begins in the initial

state of T_1 and executes the moves of T_1 (using the function δ_1) up to the point where T_1 would halt; for any move that would cause T_1 to halt in the accepting state, T_1T_2 executes the same move except that it moves instead to the initial state of T_2. At this point the tape head is positioned at the square on which T_1 halted. From this point on, the moves of T_1T_2 are the moves of T_2 (using the function δ_2). If either T_1 or T_2 would reject during this process, T_1T_2 does also, and T_1T_2 accepts precisely if and when T_2 accepts.

In order to use this composite machine in a larger context, in a manner similar to a transition diagram but without showing the states explicitly, we might also write

$$T_1 \rightarrow T_2$$

We can also make the composition conditional, depending on the current tape symbol when T_1 halts. We might write

$$T_1 \xrightarrow{a} T_2$$

to stand for the composite machine $T_1T'T_2$, where T' is described by the diagram in Figure 9.9. This composite machine can be described informally as follows: It executes the TM T_1 (rejecting if T_1 rejects, and looping if T_1 loops); if and when T_1 accepts, it executes T_2 if the current tape symbol is a and rejects otherwise.

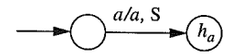


Figure 9.9 |

It is easier to understand at this point why we said in Section 9.1 that a TM is not always assumed to start with the tape head on the leftmost square of the tape. When a TM built to carry out some specific task is used as a component of a larger TM, it is likely to be called in the middle of a computation, when the tape head is at the spot on the tape where that task needs to be performed. It may be that the TM's only purpose is to change in some other specific way the tape contents and/or head position so as to create a beginning configuration appropriate for the component that follows.

Some TMs that would only halt in the rejecting state when viewed as self-contained machines (e.g., the TM that executes the algorithm "move the tape head one square to the left") can be used successfully in combination with others. On the other hand, if a TM halts normally when run independently, then it will halt normally when it is used as a component of a larger machine, provided that the tape has been prepared properly before its use. For example, a TM T expecting to find an input string z needs to begin in a configuration of the form $(q, y\underline{\Delta}z)$. As long as T halts normally when processing input z in the ordinary way, the processing of z in this way does not depend on y. (The reason is that if T halts normally when started in the configuration $(q, \underline{\Delta}z)$, then in particular T will never attempt to move its tape head to the left of the blank; therefore, starting in the configuration $(q, y\underline{\Delta}z)$, T will never actually see any of the symbols of y.) The correct execution of T does, however, depend on the tape being blank to the right of z, unless more is known about the space required for the computation involving z.

In order to be able to describe composite TMs without having to describe every primitive operation as a separate TM, it is sometimes useful to use a mixed notation in which some but not all of the states of a TM are shown. For example, the diagram in Figure 9.10*a*, which is an abbreviated version of the one in Figure 9.10*b*, has a fairly obvious meaning. If the current tape symbol is a, execute the TM T; if it is b, halt in

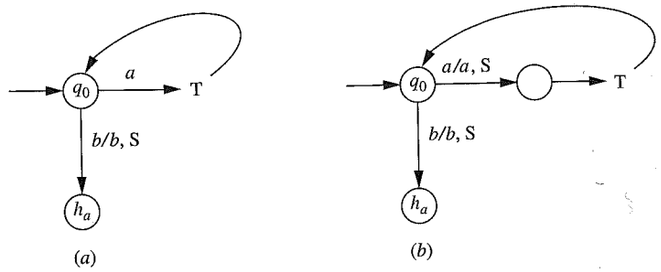


Figure 9.10 |

the accepting state; and if it is anything else, reject. In the first case, assuming T halts normally, repeat the execution of T until T halts normally scanning some symbol other than a; if at that point the tape symbol is b, halt normally, otherwise reject. (The machine might also reject during one of the iterations of T; and it might loop forever, either because one of the iterations of T does or because T halts normally with current tape symbol a every time.)

Although giving a completely precise definition of an arbitrary combination of TMs would be complicated, it is usually clear in specific examples what is involved. There is one possible source of confusion, however, in the notation we are adopting. Consider a TM T of the form $T_1 \xrightarrow{a} T_2$. If T_1 halts normally scanning some symbol not specified explicitly (i.e., other than a), T rejects. However, if T_2 halts normally, T does also—even though *no* tape symbols are specified explicitly. We could avoid this seeming inconsistency by saying that if T_1 halts normally scanning a symbol other than a, T halts normally, except that T would then not be equivalent to the composition $T_1T'T_2$ described above, and this seems undesirable. In our notation, if at the end of one sub-TM's operation at least one way is specified for the composite TM to continue, then any option that allows accepting at that point must be shown explicitly, as in Figures 9.11*a* and 9.11*b*. (The second figure is a shortened form of the first.)

Some basic TM building blocks, such as moving the head a specified number of positions in one direction or the other, writing a specific symbol in the current square, and searching to one direction or the other for a specified symbol, are straightforward and do not need to be spelled out. We consider a few slightly more involved operations.

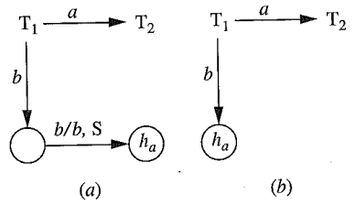


Figure 9.11 |

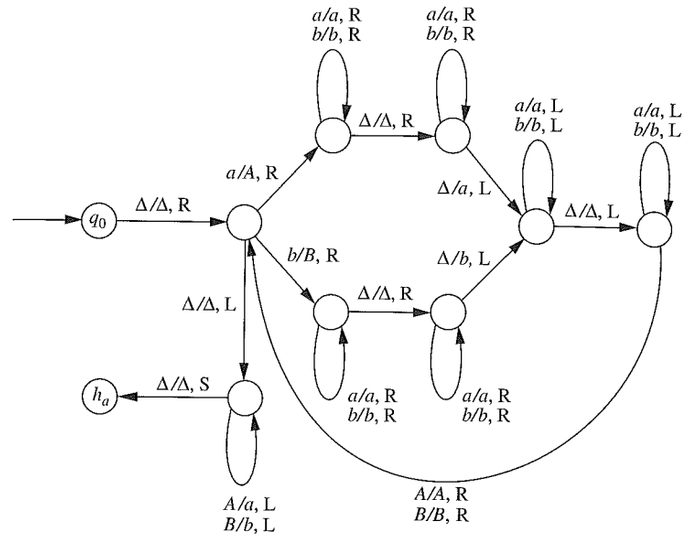


Figure 9.12 |
A Turing machine to copy strings.

Copying a String — EXAMPLE 9.7

Let us construct a TM that creates a copy of its input string, to the right of the input but with a blank separating the copy from the original. We must be careful to specify the final position of the tape head as well; let us say that if the initial configuration is $(q_0, \underline{\Delta}x)$, where x is a string of nonblank symbols, then the final configuration should be $(h_a, \underline{\Delta}x\Delta x)$. The TM will examine the first symbol, copy it in the right place, examine the second, copy it, and so on. It will keep track of its progress by changing the symbols it has copied to uppercase. We assume for simplicity that the input alphabet is $\{a, b\}$; all that is needed in a more general situation is a modified ("uppercase") version of each symbol in the input alphabet. When the copying is complete, the uppercase symbols will be changed back to the original. The TM is shown in Figure 9.12.

Deleting a Symbol — EXAMPLE 9.8

It is often useful to delete a symbol from a string. A Turing machine does this by changing the tape contents from $y\underline{a}z$ to $y\underline{z}$, where $y \in (\Sigma \cup \{\Delta\})^*$, $a \in \Sigma \cup \{\Delta\}$, and $z \in \Sigma^*$. (Remember that $y\underline{z}$ means that the tape head is positioned on the first symbol of z, or on a blank if z is null.) Again we assume that the input alphabet is $\{a, b\}$. The TM starts by replacing the symbol to be deleted by a blank, so that it can be located easily later. It moves to the right end of the string z and makes a single pass from right to left, moving symbols one square to the left as it goes, until it hits the blank. The transition diagram is shown in Figure 9.13. The states labeled q_a and q_b are what allow the machine to remember a symbol between the time it erases it and the time it writes it in the next square to the left. Of course, before it writes each symbol, it reads the symbol being written over, which determines the state it should go to next.

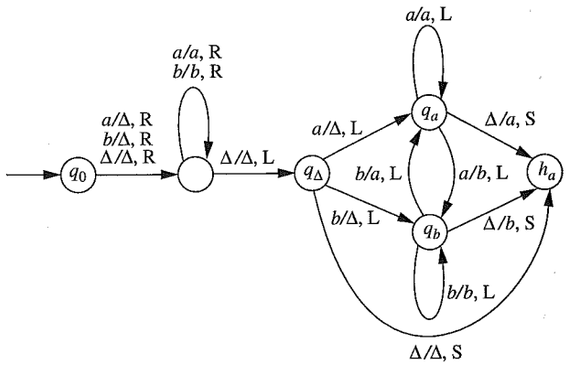


Figure 9.13 |
A Turing machine to delete a symbol.

Inserting a symbol a, or changing the tape contents from $y\underline{z}$ to $y\underline{a}z$, would be done virtually the same way, except that the single pass would go from left to right, and the move that starts it off would write a instead of Δ. You are asked to complete this machine in Exercise 9.13.

The *Delete* machine transforms $y\underline{a}z$ to $y\underline{z}$. What if it is called when the tape contents are not $y\underline{a}z$, but $y\underline{a}z\Delta w$, where w is some arbitrary string of symbols? At first it might seem that the TM ought to be designed so as to finish with $y\underline{z}\Delta w$ on the tape. A closer look, however, shows that this is unreasonable. Unless we know something about the computations that have gone on before, or unless the rightmost nonblank symbol has been marked somehow so that we can recognize it, we have no way of finding it. The instructions "move the tape head to the square containing the rightmost nonblank symbol" cannot ordinarily be executed by a TM (Exercise 9.12). In general, a Turing machine is designed according to specifications, which say that if it starts in a certain configuration it should halt normally in some other specified configuration. The specifications may say nothing about what the result should be if the TM starts in some different configuration. The machine may satisfy its specifications and yet behave unpredictably, perhaps halting in h_r or looping forever, in these abnormal situations.

EXAMPLE 9.9 Another Way of Accepting *pal*

Suppose that *Copy* is the TM in Figure 9.12 and *Reverse* is the one in Figure 9.6. Let *Equal* be a TM that works as follows: When started with tape $\underline{\Delta}x\Delta y$, where $x, y \in \{a, b\}^*$, *Equal* accepts if and only if $x = y$. Then the composite TM shown in Figure 9.14 accepts the language of palindromes over $\{a, b\}$ by comparing the input string to its reverse and accepting if and only if the two are equal.

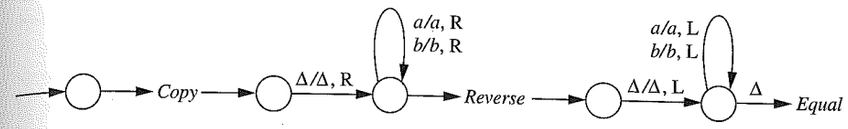


Figure 9.14 |
Another way of accepting *pal*.

9.4 | VARIATIONS OF TURING MACHINES: MULTITAPE TMs

The version of Turing machines introduced in Section 9.1 is not the only possible one. There are a number of variations in which slightly different conventions are followed with regard to starting configuration, permissible moves, protocols followed to accept strings, and so forth. In addition, the basic TM model can be enhanced in several natural ways. In this section we mention a few of the variations and investigate one enhanced version, the multitape TM, in some detail.

A more user-friendly TM, such as one with additional tapes, can make it easier to describe the implementation of an algorithm: The discussion can highlight the individual data items stored on the various tapes, without getting bogged down in the bookkeeping techniques that would be necessary if all the data were stored on and retrieved from a single tape. Thus it will often be useful to have these enhanced versions available in subsequent discussions. However, it will turn out that in spite of the extra convenience, there is no change in the ultimate computing power. Seeing the details needed to show this may help you to appreciate the power of a Turing machine. Finally, the discussion in this section will provide a useful example of how one type of computing machine can simulate another.

In order to compare two classes of abstract machines with regard to *computing power*, we must start by saying what we mean by this term. At this point, we are not considering speed, efficiency, or convenience; we are concerned only with whether the two types of machines can solve the same problems and get the same answers. A Turing machine of any type gives an "answer," first by accepting or failing to accept, and second by producing a particular output when it halts in the accepting state. This means that if we want to show that machines of type B are at least as powerful as those of type A, we need to show that for any machine T_A of type A, there is a machine T_B of type B that accepts exactly the same input strings as T_A and produces exactly the same output as T_A whenever it halts in the accepting state.

First we mention briefly a few minor variations on the basic model, each of them slightly *more* restrictive. One possibility is to require that in each move the tape head move either to the right or to the left. In this version, the values of the transition function δ are elements of $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{L,R}\}$ instead of $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{L,R,S}\}$. A second possibility is to say that a move can include writing a tape symbol or moving the tape head but not both. In this case δ would take values in $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\} \cup \{\text{L,R}\})$, where L and R are assumed

not to be elements of Γ. In both cases it is easy to see that the restrictions do not reduce the power of the machine. You are referred to the exercises for the details.

One identifiable difference between a Turing machine and a typical human computer is that a TM has a one-dimensional tape with a left end, rather than sheets of paper that might be laid out in both directions. One way to try to increase the power of the machine, therefore, might be to remove one or both of these restrictions: to make the "tape" two-dimensional, or to remove the left end and make the tape potentially infinite in both directions. In either case we would start by specifying the rules under which the machine would operate, and the conventions that would be followed with regard to input and output. Again the conclusion is that the power of the machine is not significantly changed by either addition, and again we leave the details to the exercises.

Rather than modifying the tape, we might instead add extra tapes. We could decide in that case whether to have a single tape head, which would be positioned at the same square on all the tapes, or a head on each tape that could move independently of the others. We choose the second option. An n-tape Turing machine will be specifiable as before by a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$. It will make a move on the basis of its current state and the n-tuple of tape symbols currently being examined; since the tape heads move independently, we describe the transition function as a partial function

$$\delta : Q \times (\Gamma \cup \{\Delta\})^n \to (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\})^n \times \{\mathrm{R,L,S}\}^n$$

The notion of configuration generalizes in a straightforward way: A configuration of an n-tape TM is specified by an $(n+1)$-tuple of the form

$$(q, x_1\underline{a_1}y_1, x_2\underline{a_2}y_2, \ldots, x_n\underline{a_n}y_n)$$

with the same restrictions as before on the strings x_i and y_i.

We take the initial configuration corresponding to input string x to be

$$(q_0, \underline{\Delta}x, \underline{\Delta}, \ldots, \underline{\Delta})$$

In other words, the first tape is the one used for the input. We will also say that the output of an n-tape TM is the final contents of tape 1. Tapes 2 through n are used for auxiliary working space, and when the TM halts their contents are ignored. In particular, such a TM computes a function f if, whenever it begins with an input string x in (or representing an element in) the domain of f, it halts in some configuration $(h_a, \underline{\Delta}f(x), \ldots)$, where the contents of tapes 2 through n are arbitrary, and otherwise it fails to accept.

It is obvious that for any $n \geq 2$, n-tape TMs are at least as powerful as ordinary 1-tape TMs. To simulate an ordinary TM, a TM with n tapes simply acts as if tape 1 were its only one and leaves the others blank. We now show the converse.

Theorem 9.1
Let $n \geq 2$ and let $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$ be an n-tape Turing machine. Then there is a one-tape TM $T_2 = (Q_2, \Sigma, \Gamma_2, q_2, \delta_2)$, with $\Gamma_1 \subseteq \Gamma_2$, satisfying the following two conditions.

1. $L(T_2) = L(T_1)$; that is, for any $x \in \Sigma^*$, T_2 accepts input x if and only if T_1 accepts input x.
2. For any $x \in \Sigma^*$, if

$$(q_1, \underline{\Delta}x, \underline{\Delta}, \ldots, \underline{\Delta}) \vdash_{T_1}^* (h_a, y\underline{a}z, y_2\underline{a_2}z_2, \ldots, y_n\underline{a_n}z_n)$$

(for some $a, a_i \in \Gamma_1 \cup \{\Delta\}$ and $y, z, y_i, z_i \in (\Gamma_1 \cup \{\Delta\})^*$), then

$$(q_2, \underline{\Delta}x) \vdash_{T_2}^* (h_a, y\underline{a}z)$$

In other words, if T_1 accepts input x, then T_2 accepts input x and produces the same output as T_1.

Proof
We give the proof for $n = 2$. It will be easy to see how to extend it to the general case.

We construct a one-tape TM that is capable of *simulating* the original two-tape machine, in an almost literal sense. It will act like, even *look* like, a machine with two tapes as it carries out its moves. The way to get a single tape to "look like" two tapes is to use a more complicated tape alphabet, so that what is on a single square makes it look like two squares, each with its own symbol. By adding even more symbols to the alphabet, we can take care of another technical problem with the simulation, how to keep track of the locations of the individual tape heads.

The tape alphabet Γ_2 includes the following kinds of symbols:

1. Ordinary symbols in $\Gamma = \Gamma_1 \cup \{\Delta\}$. These are necessary because input and output symbols of T_2 are of this form.
2. Elements of $\Gamma \times \Gamma$. A symbol (X, Y) of this type in square i is thought of as representing X in square i of the first tape and Y in square i of the second. We think of the tape as having two "tracks," corresponding to the two tapes of T_1. The two tracks do not exist initially but will be created gradually as T_2 moves its tape head farther and farther to the right.
3. Elements of $(\Gamma \times \Gamma') \cup (\Gamma' \times \Gamma) \cup (\Gamma' \times \Gamma')$, where Γ' contains the same symbols as Γ, marked with $'$ to distinguish them from those of Γ. At any time during the simulation, there is one primed symbol on each track to designate the location of the tape head on the corresponding tape of T_1. A pair in which both symbols are primed represents the situation in which the tape head locations in T_1 are the same on both tapes.
4. An extra symbol, #, which is inserted initially into the leftmost square, making it easy to find the beginning of the tape whenever we want.

The next step of T_2, after inserting the symbol #, is to change the blank that is now in square 1 to the symbol (Δ', Δ'), signifying that the "head" on each track is now on square 1, and then move the actual head back to square 0. From now on, the two tracks will extend as far as T_2 has ever moved its tape head to the right; whenever it advances one square farther, it converts from the old single symbol to the new double symbol.

At this point the actual simulation starts. T_1 makes moves of the form

$$\delta(p, a_1, a_2) = (q, b_1, b_2, D_1, D_2)$$

where a_1 and a_2 are the symbols in the current squares of the respective tapes. Because T_2 has only one tape head, it must determine which move it is to make at the next step by locating the primed symbols on the two tracks of its tape; it then carries out the move by making the appropriate changes to both its tracks, including the creation of new primed symbols to reflect any changes in the positions of the tape heads of T_1.

It is obviously possible for T_2 to use its states to "remember" the current state p of T_1. With this in mind, we can describe more precisely the steps T_2 might follow in simulating a single move of T_1, starting in the leftmost square of its tape.

1. Move the head to the right until a pair of the form (a_1', c) is found (c may or may not be a primed symbol), and remember a_1. Move back to the # at the beginning.
2. Locate the "head" on the second track the same way, by finding the pair of the form (d, a_2') (d may or may not be a primed symbol).
3. If the move of T_1 that has now been determined is (q, b_1, b_2, D_1, D_2) as above, remember the state q, change the pair (d, a_2') to (d, b_2), and move the tape head in direction D_2.
4. If the current square contains #, reject, since T_1 would have crashed by trying to move the tape head off tape 2. If not, and if the new square does not contain a pair of symbols (because $D_2 = \text{R}$ and T_2 has not previously examined positions this far to the right), convert the symbol a there to the pair (a, Δ'); if the new square does contain a pair, say (a, b), convert it to (a, b'). Move the tape head back to the beginning.
5. Locate the pair (a_1', c) again, as in step 1. Change it to (b_1, c) and move the tape head in direction D_1.
6. As in step 4, either reject, or change the single symbol a to the pair (a', Δ), or change the pair (a, b) to the pair (a', b). Return to the beginning of the tape.

The diagram below illustrates one iteration in a simple case.

#	Δ	0′	Δ	0	1	Δ	
	0	1	0	1′			

#	Δ	0′	Δ	0	1	Δ	…
	0	1	0	0	Δ′		

#	Δ′	Δ	Δ	0	1	Δ	
	0	1	0	0	Δ′		

The symbols a_1 and a_2 are 0 and 1, b_1 and b_2 are Δ and 0, and D_1 and D_2 are L and R, respectively, so that the move being simulated is

$$\delta(p, 0, 1) = (q, \Delta, 0, \text{L}, \text{R})$$

The second line represents the situation after step 4: The single symbol 1 has been changed to a pair of symbols, and the second one is primed to designate the resulting head position on the second tape. The third line shows the configuration after step 6.

As long as the halt state h_a has not been reached, iterating these six steps allows T_2 to simulate the moves of T_1 correctly. If and when the new state is h_a, T_2 must carry out the following additional steps in order to finish up in the correct configuration.

7. Make a pass through the tape, converting each pair (a, b) to the single symbol a. (One of these symbols a will be a primed symbol.)
8. Delete the #, so that the remaining symbols begin in square 0.
9. Move the tape head to the primed symbol, change it to the corresponding unprimed symbol, and halt in state h_a with the head in that position.

Corollary 9.1. Any language that is accepted by an n-tape TM can be accepted by an ordinary TM, and any function that is computed by an n-tape TM can be computed by an ordinary TM.

Proof The proof is immediate from Theorem 9.1.

9.5 NONDETERMINISTIC TURING MACHINES

Nondeterminism plays different roles in the two simpler models of computation we studied earlier. It is convenient but not essential in the case of FAs, whereas the language *pal* is an example of a context-free language that cannot be accepted by any deterministic PDA. Turing machines have enough computing power that once again nondeterminism fails to add any more. Any language that can be accepted by a nondeterministic TM can be accepted by an ordinary one. The argument we present to show this involves a simulation more complex than that in the previous section. Nevertheless, the idea of the proof is straightforward, and the fact that the details get complicated can be taken as evidence that TMs capable of implementing complex algorithms can be constructed from the same kinds of routine operations we have used previously.

A *nondeterministic Turing machine* (NTM) $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is defined exactly the same way as an ordinary TM, except that values of the transition function δ are subsets, rather than single elements, of the set $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{R,L,S}\}$. We do not need to say that δ is a *partial* function, because now $\delta(q, a)$ is allowed to take the value $\emptyset$.

The notation for a TM configuration is also unchanged. To say that

$$(p, x\underline{a}y) \vdash_T (q, w\underline{b}z)$$

now means that beginning in the first configuration, there is at least one move that will produce the second. Similarly,

$$(p, x\underline{a}y) \vdash_T^* (q, w\underline{b}z)$$

means that there is at least one sequence of zero or more moves that takes T from the first configuration to the second. With this definition, we may still say that a string $x \in \Sigma^*$ is accepted by T if for some $a \in \Gamma \cup \{\Delta\}$ and some $y, z \in (\Gamma \cup \{\Delta\})^*$,

$$(q_0, \underline{\Delta}x) \vdash_T^* (h_a, y\underline{a}z)$$

The idea of *output* will not be as useful in the nondeterministic case, because for a given NTM there could conceivably be an infinite set of possible outputs. NTMs that produce output, such as those in Exercise 9.29, will be used primarily as components of larger machines. When we compare NTMs to ordinary TMs, we will restrict ourselves to machines used as language acceptors.

Because every TM can be interpreted as a nondeterministic TM, it is obvious that a language accepted by a TM can be accepted by an NTM. The converse is what we need to show.

Theorem 9.2
Let $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$ be an NTM. Then there is an ordinary (deterministic) TM $T_2 = (Q_2, \Sigma, \Gamma_2, q_2, \delta_2)$ with $L(T_2) = L(T_1)$.

Proof
The TM T_2 we are looking for will have the property that for any $x \in \Sigma^*$, T_2 accepts input x if and only if there is *some* sequence of moves of T_1 on input x that would cause it to accept. The strategy for constructing T_2 is simply to let it try *every* sequence of moves of T_1, one sequence at a time, accepting if and only if it finds a sequence that would cause T_1 to halt in the accepting state.

Although there may be many configurations in which T_1 has a choice of moves, there is some fixed upper bound on the number of choices it might ever have. We assume for the sake of simplicity that this maximum number is 2. The proof we present in this case will generalize easily. There is no harm in assuming further that whenever there are any moves at all, there are exactly two, which we label 0 and 1. (The order is arbitrary, and it is possible that both are actually the same move.)

For any input string x, we can use a *computation tree* such as the one in Figure 9.15 to represent the sequences of moves T_1 might make on input x. Nodes in the tree represent configurations of T_1. The root is the initial configuration corresponding to input x, and the children of any node N correspond to the configurations T_1 might reach in one step from the configuration N. The convention we have adopted implies that every interior node has exactly two children, and a leaf node represents a halting configuration.

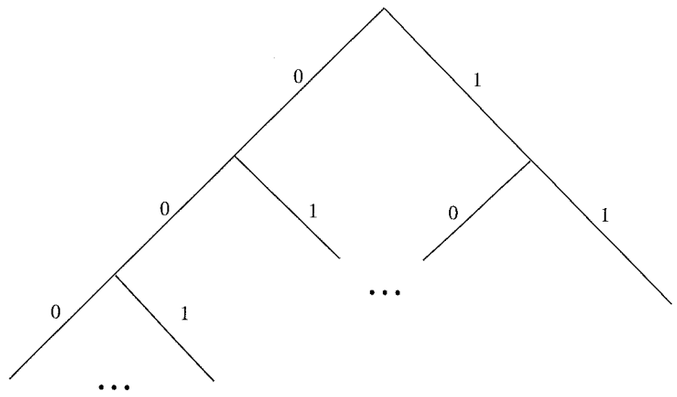


Figure 9.15 |
The computation tree for a nondeterministic TM

We can therefore think of T_2's job as searching the tree for accepting configurations. Because the tree might be infinite, a *breadth-first* approach is appropriate: T_2 will try all possible single moves, then all possible sequences of two moves, then all possible sequences of three moves, and so on. The machine we actually construct will be inefficient in that every sequence of $n+1$ moves will involve repeating a sequence of n moves tried previously. Even if the tree is finite (which means that for some n, every possible sequence of n moves T_1 can make on input x leads to a halt), T_2 will still loop forever if T_1 never accepts: It will attempt to try longer and longer sequences of moves, and the effect will be that it ends up repeating the same sequences of moves, in the same order, over and over. However, if $x \in L(T_1)$, then for some n there is a sequence of n moves that causes T_1 to accept input x, and T_2 will eventually get around to trying that sequence.

We will take advantage of Theorem 9.1 by giving T_2 three tapes. The first is used only to save the original input string, and its contents are never changed. The second is used to keep track of the sequence of moves of T_1 that T_2 is currently attempting to execute. The third is the "working tape," corresponding to T_1's tape, where T_2 actually carries out the steps specified by the current string on tape 2. Every time T_2 begins trying a new sequence, the third tape is erased and the input from tape 1 re-copied onto it.

A particular sequence of moves will be represented by a string of binary digits. The string 001, for example, represents the following sequence: first, the move representing the first (i.e., 0th) of the two choices from the initial configuration C_0, which takes T_1 to some configuration C_1; next, the first possible move from the configuration C_1, which leads to some configuration C_2; next, the second possible move from C_2. Because moves

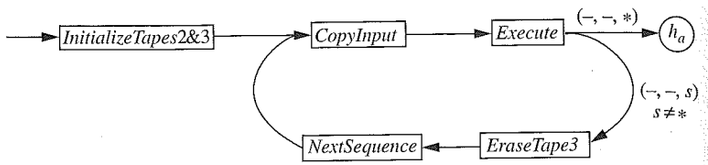


Figure 9.16 |
Simulating a nondeterministic TM by a three-tape TM.

0 and 1 may be the same, there may be several strings of digits that describe the same sequences of moves. There may also be strings of digits that do not correspond to sequences of moves, because the first few moves cause T_1 to halt. When T_2 encounters a digit that does not correspond to an executable move, it will abandon the string.

We will use the *canonical* ordering of $\{0, 1\}^*$, in which the strings are arranged in the order

$$\Lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots, 111, 0000, \ldots$$

(For two strings of different lengths, the shorter one comes first, and the order of two strings of the same length is numerical.) Given a string α representing a sequence of moves, T_2 generates the next string in this ordering by interpreting α as a binary representation and adding 1—unless $\alpha = 1^k$, in which case the next string is 0^{k+1}.

It is now easy to describe the general structure of T_2. It is composed of five smaller TMs called *InitializeTapes2&3*, *CopyInput*, *Execute*, *EraseTape3*, and *NextSequence*, combined as in Figure 9.16.

InitializeTapes2&3 writes the symbol 0 in square 1 of tape 2, to represent the sequence of moves to be tried first, and places the symbol # in square 0 of tape 3. This marker allows T_2 to detect, and recover from, an attempt by T_1 to move its head off the tape. *CopyInput* copies the original input string x from tape 1 onto tape 3, so that tape 3 has contents $\#\underline{\Delta}x$. *Execute* (which we will discuss in more detail shortly) is the TM that actually simulates the action of T_1 on this input, by executing the sequence of moves currently specified on tape 2. Its crucial feature is that it finishes with a symbol s in the current square of tape 3, and $s = *$ if and only if the sequence of moves causes T_1 to accept. In this case T_2 accepts, and otherwise it continues with the *EraseTape3* component. *EraseTape3* restores tape 3 to the configuration $\#\underline{\Delta}$. It is able to complete this operation because the length of the string on tape 2 limits how far to the right the rightmost nonblank symbol on tape 3 can be. Finally, *NextSequence* is the component already mentioned, which updates the string of digits on tape 2 using the operation similar to adding 1 in binary. Figure 9.17 shows a one-tape TM that executes this transformation on an input string of 0's and 1's; *NextSequence* is the three-tape TM that carries out this transformation on the second tape only, ignoring tapes 1 and 3.

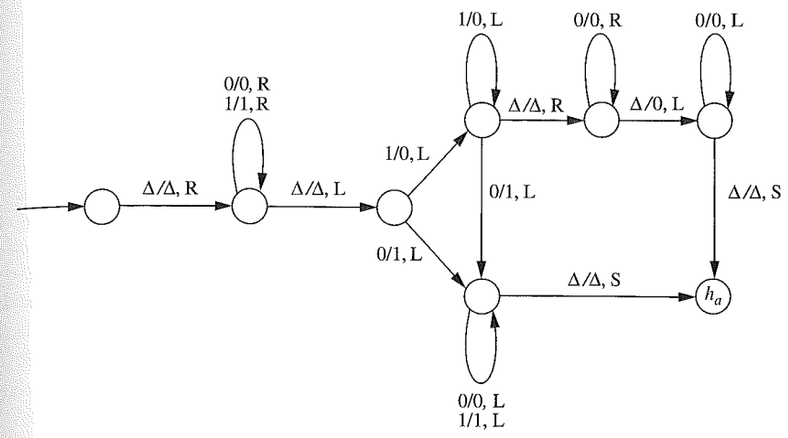
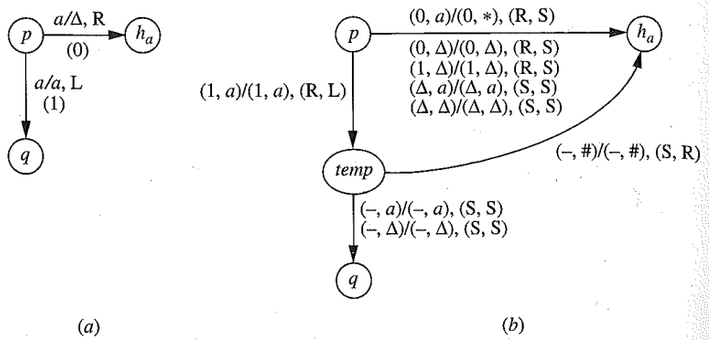


Figure 9.17 |
The one-tape version of *NextSequence*.

The problem of constructing T_2 has now been reduced to that of constructing the component *Execute*, which must simulate the sequence of moves of T_1 specified by the string of digits on tape 2. To describe this component in complete detail would be very tedious. Instead of trying to do this, we consider what a typical small portion of the transition diagram for T_1 might look like (Figure 9.18*a*), and give the corresponding portion of the diagram for *Execute* (Figure 9.18*b*). The states of *Execute* include all those of T_1, and others as well. We continue to assume that the maximum number of choices at any point in T_1's operation is 2. We simplify things still further by assuming that $\Gamma_1 = \{a\}$, so that a and Δ are the only symbols on T_1's tape. Finally, since tape 1 is ignored by *Execute*, we have presented the portion of this machine in Figure 9.18*b* as a two-tape machine.

Suppose that Figure 9.18*a* shows all the transitions from state p. Thus, if the current tape symbol is a, there are two moves. The move that accepts is arbitrarily designated move 0 and the other move 1. If the current symbol is Δ, T_1 rejects. The nonhalting move with tape symbol a may also cause it to reject when it attempts to move the tape head left. Because *Execute* should not reject, we need to specify six moves from state p, one for each combination of the three possible symbols on tape 2 (0, 1, and Δ) and the two on tape 3. (The symbol # also occurs on tape 3, but it will not occur as the current symbol in state p.)

The move made by *Execute* that simulates the accepting move of T_1 leaves the symbol $*$ in the current position on tape 3 and causes *Execute* to accept. This is the first of the transitions shown from p to h in Figure 9.18*b*. Note that it ignores the instruction to move the head on tape 3 to the right, on



(a) (b)

Figure 9.18 |
A typical small portion of *Execute*.

the assumption that once T_1 halts the current position on the tape is irrelevant. The next two transitions from p to h correspond to the moves that cause T_1 to reject. The last two are for the situation in which the symbol on tape 2 is Δ, indicating that the current sequence of moves has been simulated completely.

The other move from state p in *Execute* is the move corresponding to choice 1. This occurs when the current digit on tape 2 is 1 and the symbol on tape 3 is a. The reason this move does not go directly to state q is that *Execute* must first test to see if the move is possible for T_1. It does this by moving the head on tape 3 to the left and entering a "temporary" state; from this state, any symbol on tape 3 other than # indicates that T_1 made the move safely, and *Execute* can then go to state q as T_1 would have in the first place. The symbol # indicates a crash by T_1, and *Execute* halts normally after returning the tape head to the square to the right of #.

Although this small example does not illustrate every situation that can occur, it should help to convince you that the complete transition diagram for T_1 can eventually be transformed into a corresponding diagram for *Execute*. The conclusion is that it is indeed possible for T_2 to simulate every possible finite sequence of moves of T_1.

EXAMPLE 9.10 A Simple Example of Nondeterminism

Consider the TM *Double* that works as follows. Using the *Copy* TM from Example 9.7, modified for a one-symbol input alphabet, it makes a copy of the numerical input. It then deletes the blank separating the original input from the copy and returns the tape head to square 0. Just as the name suggests, it doubles the value of the input.

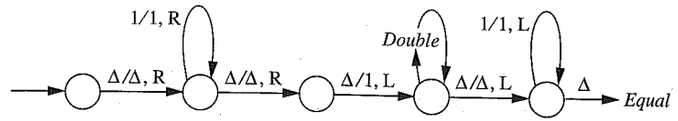


Figure 9.19 |
An NTM to accept strings of length 2^i.

Now look at the nondeterministic TM T in Figure 9.19. T moves past the input string, places a single 1 on the tape, separated from the input by a blank, and after positioning the tape head on the blank, executes *Double* zero or more times before returning the tape head to square 0. Finally, it executes the TM *Equal*, which works as follows: Starting with tape contents $\underline{\Delta}x\Delta y$, where x and y are strings of 1's, *Equal* accepts if and only if $x = y$ (see Example 9.9).

The nondeterminism in T lies in the indeterminate number of times *Double* is executed. When *Equal* is finally executed, the string following the input on the tape represents some arbitrary power of 2. If the original input string happens to represent a power of 2, say 2^i, then there is a sequence of choices T can make that will cause the input to be accepted—namely, the sequence in which *Double* is executed exactly i times. On the other hand, if the input string is not a power of 2, it will fail to be accepted, because in the last step it is compared to a string that *must* be a power of 2. Our conclusion is that T accepts the language $\{1^{2^i} \mid i \geq 0\}$.

We do not need nondeterminism in order to accept this language. (Nondeterminism is never necessary, by Theorem 9.2.) It merely simplifies the description. One deterministic way to test whether an integer is a power of 2 is to test the integer first to see if it is 1 and, if not, perform a sequence of divisions by 2. If at any step before reaching 1 we get a nonzero remainder, we answer no. If we finally obtain the quotient 1 without any of the divisions producing a remainder, we answer yes. We would normally say, however, that *multiplying* by 2 is easier than *dividing* by 2. An easier approach, therefore, might be to start with 1 and perform a sequence of multiplications by 2. We could compare the result of each of these to the original input, accepting if we eventually obtained a number equal to it, and either rejecting if we eventually obtained a number larger than the input, or simply letting the iterations continue forever.

The nondeterministic solution T in Figure 9.19 is closer to the second approach, except that instead of comparing the input to each of the numbers 2^i, it guesses a value of i and tests that value only. Removing the nondeterminism means replacing the guess by an iteration in which all the values are tested; a deterministic TM that did this would simply be a more efficient version of the TM constructed in the proof of Theorem 9.2, which tests all possible sequences of moves of T.

9.6 | UNIVERSAL TURING MACHINES

In our discussions so far, a Turing machine is created to execute a specific algorithm. If we have a Turing machine for computing one function, then computing a different

function or doing some other calculation requires a different machine. Originally, electronic computers were limited in a similar way, and changing the calculation to be performed meant rewiring the machine.

A 1936 paper by Turing, however, anticipated the *stored-program* computers you are familiar with. Although a modern computer is still "hard-wired," it is completely flexible in that the task it performs is to execute the instructions stored in its memory, and these can represent any conceivable algorithm. Turing describes a "universal computing machine" that works as follows. It is a TM T_u whose input consists essentially of a program and a data set for the program to process. The program takes the form of a string specifying some other (special-purpose) TM T_1, and the data set is a second string z interpreted as input to T_1. T_u then simulates the processing of z by T_1. In this section we will describe one such universal Turing machine T_u.

The first step is to formulate a notational system in which we can encode both an arbitrary TM T_1 and an input string z over an arbitrary alphabet as strings $e(T_1)$ and $e(z)$ over some fixed alphabet. The crucial aspect of the encoding is that it must not destroy any information; given the strings $e(T_1)$ and $e(z)$, we must be able to reconstruct the Turing machine T_1 and the string z. We will use the alphabet $\{0, 1\}$, although we must remember that the TM we are encoding may have a much larger alphabet. We start by assigning positive integers to each state, each tape symbol, and each of the three "directions" S, L, and R in the TM T_1 we want to encode.

At this point, a slight technical problem arises. We want the encoding function e to be one-to-one, so that a string of 0's and 1's encodes at most one TM. Consider two TMs T_1 and T_2 that are identical except that the tape symbols of T_1 are a and b and those of T_2 are a and c. If we really want to call these two TMs different, then in order to guarantee that their encodings are different, we must make sure that the integers assigned to b and c are different. To accommodate *any* TM and still ensure that the encoding is one-to-one, we must somehow fix it so that no symbol in any TM's alphabet receives the same number as any other symbol in any other TM's alphabet. The easiest way to handle this problem is to fix once and for all the set of symbols that can be used by TMs, and to number these symbols at the outset. This is the reason for the following

Convention. We assume from this point on that there are two fixed infinite sets $\mathcal{Q} = \{q_1, q_2, \ldots\}$ and $\mathcal{S} = \{a_1, a_2, \ldots\}$ so that for any Turing machine $T = (Q, \Sigma, \Gamma, q_0, \delta)$, we have $Q \subseteq \mathcal{Q}$ and $\Gamma \subseteq \mathcal{S}$. ■

It should be clear that this assumption about states is not a restriction at all, because the names assigned to the states of a TM are irrelevant. Furthermore, as long as $\mathcal{S}$ contains all the letters, digits, and other symbols we might want in our input alphabets, the other assumption is equally harmless (no more restrictive, for example, than limiting the character set on a computer to 256 characters). Once we have a subscript attached to every possible state and tape symbol, we can represent a state or a symbol by a string of 0's of the appropriate length; 1's are used as separators.

Definition 9.5 The Encoding Function *e*

First we associate to each tape symbol (including Δ), to each state (including h_a and h_r), and to each of the three directions, a string of 0's. Let

$$\begin{aligned} s(\Delta) &= 0 \\ s(a_i) &= 0^{i+1} \quad \text{(for each } a_i \in \mathcal{S}\text{)} \\ s(h_a) &= 0 \\ s(h_r) &= 00 \\ s(q_i) &= 0^{i+2} \quad \text{(for each } q_i \in \mathcal{Q}\text{)} \\ s(\mathrm{S}) &= 0 \\ s(\mathrm{L}) &= 00 \\ s(\mathrm{R}) &= 000 \end{aligned}$$

Each move m of a TM, described by the formula

$$\delta(p, a) = (q, b, D)$$

is encoded by the string

$$e(m) = s(p)1s(a)1s(q)1s(b)1s(D)1$$

and for any TM T, with initial state q, T is encoded by the string

$$e(T) = s(q)1e(m_1)1e(m_2)1 \cdots e(m_k)1$$

where $m_1, m_2, \cdots, m_k$ are the distinct moves of T, arranged in some arbitrary order. Finally, any string $z = z_1 z_2 \cdots z_k$, where each $z_i \in \mathcal{S}$, is encoded by

$$e(z) = 1s(z_1)1s(z_2)1 \cdots s(z_k)1$$

The 1 at the beginning of the string $e(z)$ is included so that in a composite string of the form $e(T)e(z)$, there will be no doubt as to where $e(T)$ stops. Notice that one consequence is that the encoding $s(a)$ of a single symbol $a \in \mathcal{S}$ is different from the encoding $e(a)$ of the one-character string a.

Because the moves of a TM T can appear in the string $e(T)$ in any order, there will in general be many correct encodings of T. However, any string of 0's and 1's can be the encoding of at most one TM.

The Encoding of a Simple TM

EXAMPLE 9.11

Consider the TM illustrated in Figure 9.20, which transforms an input string of a's and b's by changing the leftmost a, if there is one, to b. Let us assume for simplicity that the tape symbols a and b are assigned the numbers 1 and 2, so that $s(a) = 00$ and $s(b) = 000$, and that the states q_0, p, and r are given the numbers 1, 2, and 3, respectively. If we take the six moves in the order they appear, left-to-right, the first move $\delta(q_0, \Delta) = (p, \Delta, \mathrm{R})$ is encoded by the string

$$0^3 1 0^1 1 0^4 1 0^1 1 0^3 1 = 00010100001010001$$

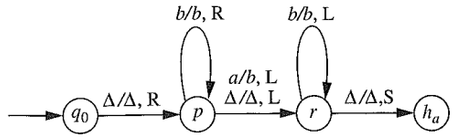


Figure 9.20 |

and the entire TM by the string

0001 000101000010100011 000010001000010001000011 000001001000000100010011
000001010000010100011 000000100010000010001000011 000001010101011

Remember that the first part of the string, in this case 0001, is to identify the initial state of the TM. The individual moves in the remainder of the string are separated by spaces for readability.

The input to the universal TM T_u will consist of a string of the form $e(T)e(z)$, where T is a TM and z is a string over T's input alphabet. In Example 9.11, if the input string to T were baa, the corresponding input string to T_u would consist of the string $e(T)$ given in the example, followed by 10001001001. On any input string of the form $e(T)e(z)$, we want T_u to accept if and only if T accepts input z, and in this case we want the output from T_u to be the encoded form of the output produced by T on input z.

Now we are ready to construct T_u. It will be convenient to give it three tapes. According to our convention for multitape TMs, the first tape will be both the input and output tape. It will initially contain the input string $e(T)e(z)$. The second tape will be the working tape during the simulation of T, and the third tape will contain the encoded form of the state T is currently in.

The first step of T_u is to move the string $e(z)$ (except for the initial 1) from the end of tape 1 to tape 2, beginning in square 3. Since T begins with its leftmost square blank, T_u will write 01 (because $0 = s(\Delta)$) in squares 1 and 2 of tape 2; square 0 is left blank, and the tape head is positioned on square 1. The next step for T_u is to copy the encoded form of T's initial state from the beginning of tape 1 onto tape 3, beginning in square 1, and to delete it from tape 1.

After these initial steps, T_u is ready to begin simulating the action of T (encoded on tape 1) on the input string (encoded on tape 2). As the simulation starts, the three tape heads are all in square 1. The next move of T at any point is determined by T's state (encoded on tape 3) and the current symbol on T's tape, whose encoding starts in the current position on tape 2. In order to simulate this move, T_u must search tape 1 for the 5-tuple whose first two parts match this state-input combination. Abstractly this is a straightforward pattern-matching operation, and a TM that carries it out is shown in Figure 9.21. Since the search operation never changes any tape symbols, we have simplified the labeling slightly, by writing as

$$(a, b, c), (D_1, D_2, D_3)$$

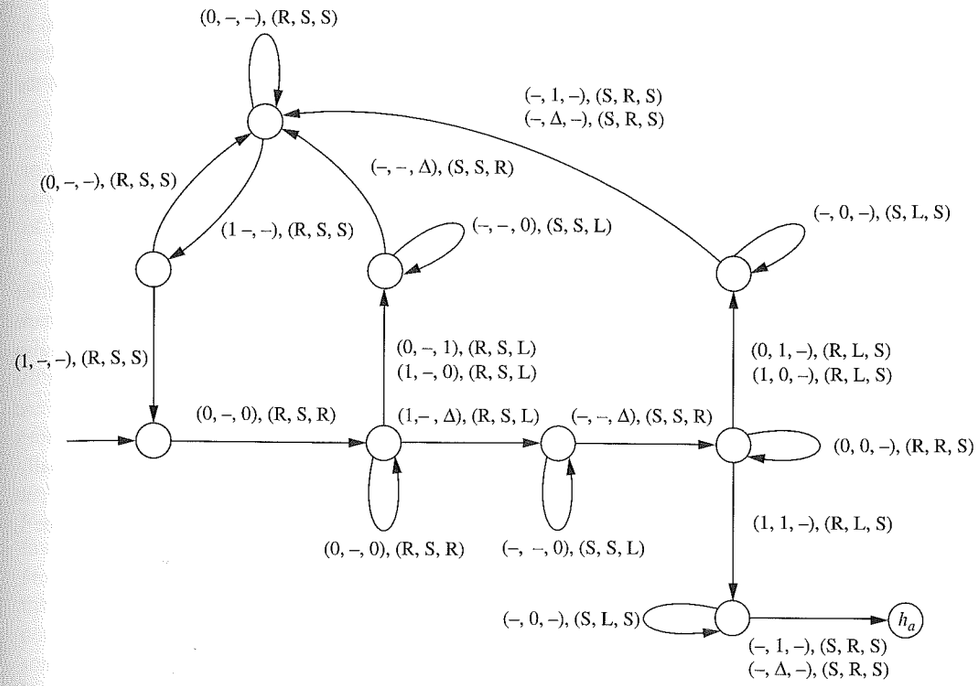


Figure 9.21 |
Finding the right move on tape 1.

what would normally be written as

$$(a, b, c)/(a, b, c), (D_1, D_2, D_3)$$

Once the appropriate 5-tuple is found, the last three parts tell T_u how to simulate the move. To illustrate, suppose that before the search, T_u's three tapes look like this:

Δ0001010000101000110000100100000100010011000 1...
Δ01001 0 010001Δ...
Δ 0 0000Δ...

The corresponding tape of T would be

$$\Delta a \underline{a} b \Delta \ldots$$

assuming that the symbols numbered 1 and 2 are a and b, respectively, and T would be in state 2 (the one with encoding 0000). After the search of tape 1, the tapes look like this:

Δ000101000010100011000010010000010001001100001...
Δ01001 0 010001Δ...
Δ 0 0000Δ...

The 5-tuple on tape 1 specifies that T's current symbol should be changed to b, the head should be moved left, and the state should be changed to state 3. These operations can be carried out by T_u in a fairly straightforward way, and we omit the details. The final result is

$\Delta\underline{0}0010100001010001100001001000001000100110001\ldots$
$\Delta 01\underline{0}0100010001\Delta\ldots$
$\Delta\underline{0}0000\Delta\ldots$

and T_u is now ready to simulate the next move.

There are several ways this process might stop. T might halt abnormally, either because there is no move specified or because the move calls for it to move its tape head off the tape. In the first case, the search operation pictured in Figure 9.21 also halts abnormally (although the move to h_r is not shown explicitly), because after the last 5-tuple on tape 1 has been tried unsuccessfully, the second of the 1's at the end takes the machine back to the initial state, and there is no move from that state with 1 on tape 1. We can easily arrange for T_u to reject in the second case as well. Finally, T may accept. T_u detects this when it processes a 5-tuple on tape 1 whose third part is a single 0. In this case, after T_u has changed tape 2 appropriately, it erases tape 1, copies tape 2 onto tape 1, and accepts.

9.7 MODELS OF COMPUTATION AND THE CHURCH-TURING THESIS

A Turing machine is a model of computation more general than either a finite automaton or a pushdown automaton, and in this chapter we have seen examples of computations that are feasible on a TM but not on the simpler machines. A TM is not the only possible way of extending a PDA, and we might examine some other approaches briefly.

Our first example of a non-CFL (Example 8.1) was the language $L = \{a^n b^n c^n \mid n \geq 1\}$. The pumping lemma for CFLs tells us that a finite automaton with a single stack is not sufficient to recognize strings of this form. One stack is sufficient to accept $\{a^n b^n \mid n \geq 1\}$; it is not hard to see that *two* stacks are sufficient for L. Of course, if $\{a^n b^n c^n d^n \mid n \geq 1\}$ turned out to require three stacks, and $\{a^n b^n c^n d^n e^n \mid n \geq 1\}$ four, then this observation would not be useful. The interesting thing is that two are enough to handle all these languages (Exercise 9.49).

In Example 8.2, we considered $L = \{ss \mid s \in \{a, b\}^*\}$. If we ignore the apparent need for nondeterminism by changing the language to $\{scs \mid s \in \{a, b\}^*\}$, then we might consider trying to accept this language using a finite automaton with a *queue* instead of a stack. We could load the first half of the string on the queue, and the "first in, first out" operation of the queue would allow us to compare the first and second halves from left to right.

Although it is not at all obvious from these two simple examples, both approaches lead to families of abstract machines with the same computing power as Turing machines. With appropriate conventions regarding input and output, both these models can be considered as general models of computation and have been studied from this point of view. They are investigated in more detail in Exercises 9.49–9.53.

Still, a Turing machine seems like a more natural approach to a general-purpose computer, perhaps because of Turing's attempt to incorporate into TM moves the primitive steps carried out by a human computer. Even a few examples, such as the TM accepting $\{ss \mid s \in \{a, b\}^*\}$ in Example 9.3 or the one computing the reverse function in Example 9.4, are enough to suggest that TMs have the basic features required to carry out algorithms of arbitrary complexity. The point is not that recognizing strings of the form ss is a particularly complex calculation, but that even algorithms of much greater logical complexity depend ultimately on the same sorts of routine manipulations that appear in these two examples. Designing algorithms to solve problems can of course be difficult; implementing an algorithm on a TM is primarily a matter of organizing the data storage areas, and choosing bookkeeping mechanisms for keeping track of the progress of the algorithm. A simple model of computation such as an FA puts severe restrictions on the *type* of algorithm that can be executed. A TM allows one to design an algorithm without reference to the machine and to have confidence that the result can be implemented.

To say that the Turing machine is a general model of computation is simply to say that any algorithmic procedure that can be carried out at all (by a human, a team of humans, or a computer) can be carried out by a TM. This statement was first formulated by Alonzo Church, a logician, in the 1930s (*American Journal of Mathematics* 58:345–363, 1936), and it is usually referred to as *Church's thesis*, or the *Church-Turing thesis*. It is not a mathematically precise statement because we do not have a precise definition of the term *algorithmic procedure*, and therefore it is not something we can prove. Since the invention of the TM, however, enough evidence has accumulated to cause the Church-Turing thesis to be generally accepted. Here is an informal summary of some of the evidence.

1. The nature of the model makes it seem likely that all the steps that are crucial to human computation can be carried out by a TM. Of course, there are differences in the details of how they are carried out. A human normally works with a two-dimensional sheet of paper, not a one-dimensional tape, and a human is perhaps not restricted to transferring his or her attention to the location immediately adjacent to the current one. However, although working within the constraints imposed by a TM might make certain steps in a computation awkward, it does not appear to limit the types of computation that are possible. For example, if the two-dimensional aspect of the paper really plays a significant role in a computation, the TM tape can be organized so as to simulate two dimensions. This may mean that two locations contiguous on a sheet of paper are not contiguous on the tape; the only consequence is that the TM may require more moves to do what a human could do in one.
2. Various enhancements of the TM model have been suggested in order to make the operation more like that of a human computer, or more convenient, or more efficient. These include the enhancements mentioned in this chapter, such as doubly infinite tapes, multiple tapes, and nondeterminism. In each case, it is possible to show that the computing power of the machine is unchanged.
3. Other theoretical models of computation have been proposed. These include machines such as those mentioned earlier in this section, machines that are

closer to modern computers in their operation, and various notational systems (simple programming-type languages, grammars, and others) that can be used to describe computations. Again, in every case, the model has been shown to be equivalent to the Turing machine.

4. Since the introduction of the TM, no one has suggested any type of computation that ought to be included in the category of "algorithmic procedure" and *cannot* be implemented on a TM.

As we observed earlier, the Church-Turing thesis is not a statement for which a precise proof is possible, because of the imprecision in the term "algorithmic procedure." Once we adopt the thesis, however, we are effectively giving a precise meaning to the term: An algorithm is a procedure that can be executed on a TM. The advantage of having such a definition is that it provides a starting point for a discussion of problems that can be solved algorithmically and problems (if any) that cannot. This discussion begins in Chapter 10 and continues in Chapter 11.

Another way in which the Church-Turing thesis will be used in the rest of the book is that when we want to describe a solution to a problem, we will often be satisfied with a verbal description of the algorithm; translating it into a detailed TM implementation may be tedious but is generally straightforward.

EXERCISES

9.1. Trace the TM in Figure 9.5 (the one accepting the language $\{ss \mid s \in \{a, b\}^*\}$) on the string $aaba$. Show the configuration at each step.

9.2. Below is a transition table for a TM.

q	σ	$\delta(q, \sigma)$	q	σ	$\delta(q, \sigma)$	q	σ	$\delta(q, \sigma)$
q_0	Δ	(q_1, Δ, R)	q_2	Δ	(h_a, Δ, R)	q_6	a	(q_6, a, R)
q_1	a	(q_1, a, R)	q_3	Δ	(q_4, a, R)	q_6	b	(q_6, b, R)
q_1	b	(q_1, b, R)	q_4	a	(q_4, a, R)	q_6	Δ	(q_7, b, L)
q_1	Δ	(q_2, Δ, L)	q_4	b	(q_4, b, R)	q_7	a	(q_7, a, L)
q_2	a	(q_3, Δ, R)	q_4	Δ	(q_7, a, L)	q_7	b	(q_7, b, L)
q_2	b	(q_5, Δ, R)	q_5	Δ	(q_6, b, R)	q_7	Δ	(q_2, Δ, L)

a. What is the final configuration if the input is ab?

b. What is the final configuration if the input is baa?

c. Describe what the TM does for an arbitrary input string in $\{a, b\}^*$.

9.3. Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a TM, and let s and t be the sizes of the sets Q and Γ, respectively. How many distinct configurations of T could there possibly be in which all tape squares past square n are blank and T's tape head is on or to the left of square n? (The tape squares are numbered beginning with 0.)

9.4. The TM shown in Figure 9.2*b* (obtained from the FA in Figure 9.2*a*) accepts a string as soon as it finds the substring aba. Draw another TM accepting the same language that is more similar to the FA in that it accepts a string only after it has read all the symbols of the string.

9.5. Figures 9.2 and 9.3 show two examples of converting an FA to a TM accepting the same language. Describe precisely how this can be done for an arbitrary FA.

9.6. Draw a transition diagram for a Turing machine accepting each of the following languages.

a. $\{a^i b^j \mid i < j\}$

b. $\{a^n b^n c^n \mid n \geq 0\}$

c. $\{x \in \{a, b, c\}^* \mid n_a(x) = n_b(x) = n_c(x)\}$

d. The language of balanced strings of parentheses

e. The language of all nonpalindromes over $\{a, b\}$

f. $\{www \mid w \in \{a, b\}^*\}$

9.7. Describe the language (a subset of $\{1\}^*$) accepted by the TM in Figure 9.22.

9.8. We do not define Λ-transitions for a TM. Why not? What features of a TM make it unnecessary or inappropriate to talk about Λ-transitions?

9.9. Suppose T_1 and T_2 are TMs accepting languages L_1 and L_2 (both subsets of Σ^*), respectively. If we were following as closely as possible the method used in the case of finite automata to accept the language $L_1 L_2$, we might form the composite TM $T_1 T_2$. (See the construction of M_c in the proof of Theorem 4.4.) Explain why this approach, or any obvious modification of it, will not work.

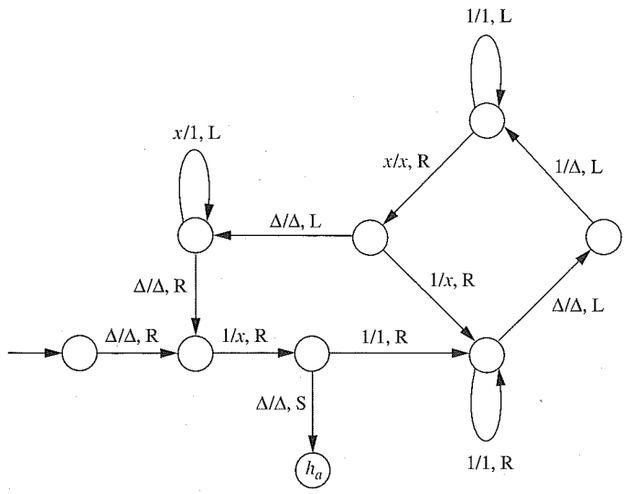


Figure 9.22 |

9.10. Given TMs $T_1 = (Q_1, \Sigma_1, \Gamma_1, q_1, \delta_1)$ and $T_2 = (Q_2, \Sigma_2, \Gamma_2, q_2, \delta_2)$, with $\Gamma_1 \subseteq \Sigma_2$, give a precise definition of the TM $T_1T_2 = (Q, \Sigma, \Gamma, q_0, \delta)$. Say precisely what Q, Σ, Γ, q_0, and δ are.

9.11. Suppose T is a TM accepting a language L. Describe how you would modify T to obtain another TM accepting L that never halts in the reject state h_r.

9.12. Suppose T is a TM that accepts every input. We would like to construct a TM R_T so that for every input string x, R_T halts in the accepting state with exactly the same tape contents as when T halts on input x, but with the tape head positioned at the rightmost nonblank symbol on the tape. (One reason this is useful is that we might want to use T in a larger composite machine, but to erase the tape after T has halted.)

a. Show that there is no fixed TM T_0 so that $R_T = TT_0$ for every T. (In other words, there is no TM capable of executing the instruction "move the tape head to the rightmost nonblank tape symbol" in every possible situation.)

b. Describe a general method for constructing R_T, given T.

9.13. Draw the *Insert*(σ) TM, which changes the tape contents from $y\underline{z}$ to $y\underline{\sigma}z$. Here $y \in (\Sigma \cup \{\Delta\})^*$, $\sigma \in \Sigma \cup \{\Delta\}$, and $z \in \Sigma^*$. You may assume that $\Sigma = \{a, b\}$.

9.14. Does every TM compute a partial function? Explain.

9.15. In each case, draw a TM that computes the indicated function. In the first five parts, the function is from $\mathcal{N}$ to $\mathcal{N}$. In each of these parts, assume that the TM uses unary notation—that is, the natural number n is represented by the string 1^n.

a. $f(x) = x + 2$

b. $f(x) = 2x$

c. $f(x) = x^2$

d. $f(x) = x/2$ ("/" means integer division.)

e. $f(x) =$ the smallest integer greater than or equal to $\log_2(x + 1)$ (i.e., $f(0) = 0$, $f(1) = 1$, $f(2) = f(3) = 2$, $f(4) = \cdots = f(7) = 3$, and so on.)

f. $f : \{a, b\}^* \times \{a, b\}^* \to \{0, 1\}$ defined by $f(x, y) = 1$ if $x = y$, $f(x, y) = 0$ otherwise.

g. $f : \{a, b\}^* \times \{a, b\}^* \to \{0, 1\}$ defined by $f(x, y) = 1$ if $x < y$, $f(x, y) = 0$ otherwise. Here $<$ means with respect to "lexicographic," or alphabetical, order. For example, $a < aa$, $abab < abb$, etc.

h. f is the same as in the previous part, except that this time $<$ refers to canonical order. That is, a shorter string precedes a longer one, and the order of two strings of the same length is alphabetical.

i. $f : \{a, b\}^* \to \{a, b\}^*$ defined by $f(x) = a^{n_a(x)}b^{n_b(x)}$ (i.e., $f(x)$ has the same symbols as x but with all the a's at the beginning).

9.16. The TM shown in Figure 9.23 computes a function from $\{a, b\}^*$ to $\{a, b\}^*$. For any string $x \in \{a, b\}^*$, describe the string $f(x)$.

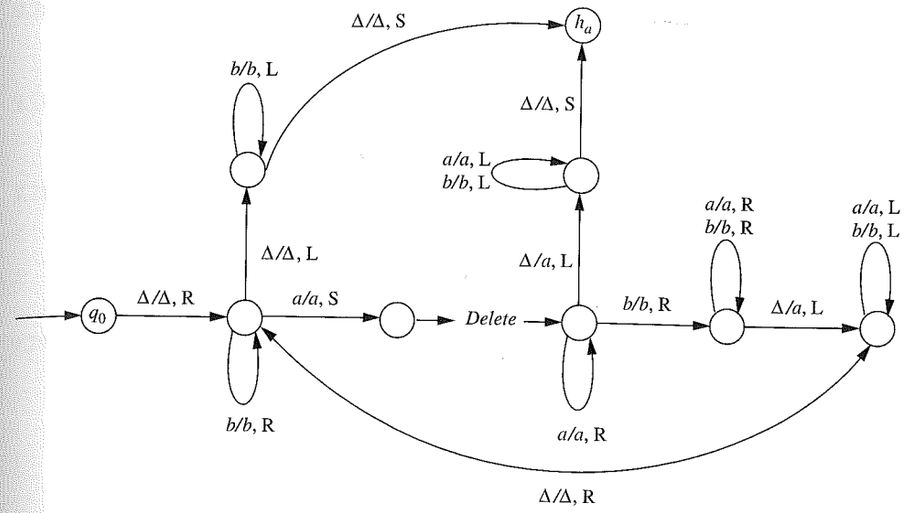


Figure 9.23 |

9.17. Suppose TMs T_1 and T_2 compute the functions f_1 and f_2 from $\mathcal{N}$ to $\mathcal{N}$, respectively. Describe how to construct a TM to compute each of these functions.

a. $f_1 + f_2$

b. the minimum of f_1 and f_2

c. $f_1 \circ f_2$

9.18. Draw a TM that takes as input a string of 0's and 1's, interprets it as the binary representation of a nonnegative integer, and leaves as output the unary representation of that integer (i.e., a string of that many 1's).

9.19. Draw a TM that does the reverse of the previous problem: accepts a string of n 1's as input and leaves as output the binary representation of n.

9.20. In Figure 9.24 is a TM accepting the language $\{scs \mid s \in \{a, b\}^*\}$. Modify it so as to obtain a TM that computes the characteristic function of the same language.

9.21. In Example 9.3, a TM is given that accepts the language $\{ss \mid s \in \{a, b\}^*\}$. Draw a TM with tape alphabet $\{a, b\}$ that accepts this language.

9.22. In Section 9.5 we mentioned a variation of TMs in which the transition function δ takes values in $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{L,R}\}$, so that the tape head must move either to the left or to the right on each move. It is not difficult to show that any ordinary TM can be simulated by one of these. Explain how the move $\delta(p, a) = (q, b, S)$ could be simulated by such a restricted TM.

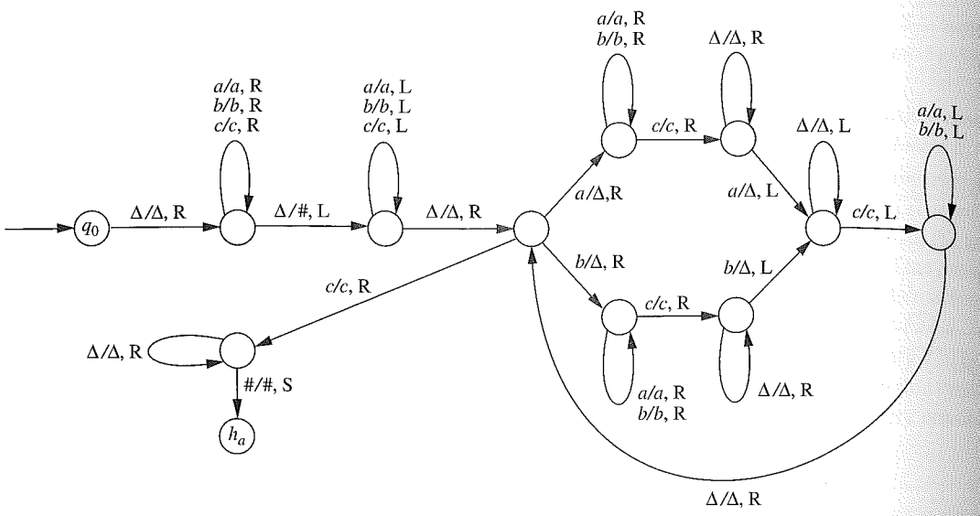


Figure 9.24 |

9.23. An ordinary TM can also be simulated by one in which δ takes values in $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\} \cup \{\text{L,R}\})$, so that writing a symbol and moving the tape head are not both allowed on the same move. Explain how the move $\delta(p, a) = (q, b, \text{R})$ of an ordinary TM could be simulated by one of these restricted TMs.

Exercises 9.24–9.25 involve a Turing machine with a doubly infinite tape. The tape squares on such a machine can be thought of as numbered left to right, as in an ordinary TM, but now the numbers include all negative integers as well as nonnegative. A configuration can still be described by a pair $(q, x\underline{a}y)$. There is no assumption about which square the string x begins in; in other words, two configurations that are identical except for the square in which x begins are considered the same. For this reason, we may adopt the same convention about the string x as about y: when we specify a configuration as $(q, x\underline{a}y)$, we may assume that x does not begin with a blank.

9.24. Construct a TM with a doubly-infinite tape that does the following: If it begins with the tape blank except for a single a somewhere on it, it halts in the accepting state with the head scanning the square with the a.

9.25. Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a TM. Show that there is a TM $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$ with a doubly-infinite tape, with $\Gamma \subseteq \Gamma_1$, satisfying these two conditions:

a. For any $x \in \Sigma^*$, T_1 accepts input x if and only if T does.

b. For any $x \in \Sigma^*$, if $(q_0, \underline{\Delta}x) \vdash_T^* (h_a, y\underline{a}z)$, then $(q_1, \underline{\Delta}x) \vdash_{T_1}^* (h_a, y\underline{a}z)$.

9.26. In defining a multitape TM, another option is to specify a single tape head that scans the same position on all tapes simultaneously. Show that a machine of this type is equivalent to the multitape TM defined in Section 9.5.

9.27. Draw the portion of the transition diagram for the one-tape TM M_2 embodying the six steps shown in the proof of Theorem 9.1 corresponding to the move $\delta_1(p, a_1, a_2) = (q, b_1, b_2, \text{R,L})$ of M_1.

9.28. Draw a transition diagram for a three-tape TM that works as follows: starting in the configuration $(q_0, \underline{\Delta}x, \underline{\Delta}y, \underline{\Delta})$, where x and y are strings of 0's and 1's of the same length, it halts in the configuration $(h_a, \underline{\Delta}x, \underline{\Delta}y, \underline{\Delta}z)$, where z is the string obtained by interpreting x and y as binary representations and adding them.

9.29. What is the effect of the nondeterministic TM with input alphabet {0, 1} whose transition table is shown below, assuming it starts with a blank tape? (Assuming that it halts, where is the tape head when it halts, and what strings might be on the tape?)

q	σ	$\delta(q, \sigma)$
q_0	Δ	$\{(q_1, \Delta, \text{R})\}$
q_1	Δ	$\{(q_1, 0, \text{R}), (q_1, 1, \text{R}), (q_2, \Delta, \text{L})\}$
q_2	0	$\{(q_2, 0, \text{L})\}$
q_2	1	$\{(q_2, 1, \text{L})\}$
q_2	Δ	$\{(h_a, \Delta, \text{S})\}$

9.30. Call the NTM in the previous exercise G. Let *Copy* be the TM in Example 9.4, which transforms $\underline{\Delta}x$ to $\underline{\Delta}x\Delta x$ for an arbitrary string $x \in \{0, 1\}^*$. Finally, let *Equal* be a TM that works as follows: starting with the tape $\underline{\Delta}x\Delta y$, it accepts if and only if $x = y$. Consider the NTM shown in Figure 9.25. (It is nondeterministic because G is.) What language does it accept?

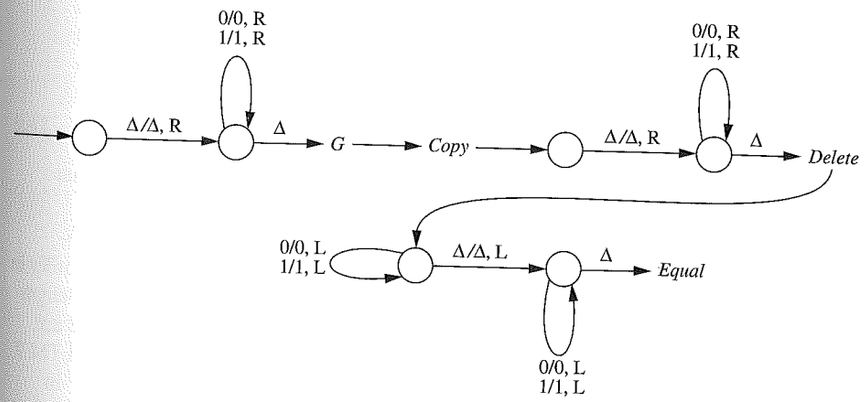


Figure 9.25 |

9.31. Using the idea in the previous exercise, draw a transition diagram for an NTM that accepts the language $\{1^n \mid n = k^2 \text{ for some } k \geq 0\}$.

9.32. Using the same general technique, draw a transition diagram for an NTM that accepts the language $\{1^n \mid n \text{ is a composite integer } \geq 4\}$.

9.33. Suppose L is accepted by a TM T. Describe how you could construct a nondeterministic TM to accept each of the following languages.

a. The set of all prefixes of elements of L

b. The set of all suffixes of elements of L

c. The set of all substrings of elements of L

9.34. Figure 9.18*b* shows the portion of the *Execute* TM corresponding to the portion of M_1 shown in Figure 9.18*a*. Consider the portion of M_1 shown in Figure 9.26. Assume as before that the maximum number of choices at any point in M_1 is 2, and that the moves shown are the only ones from state r. Draw the corresponding portion of *Execute*.

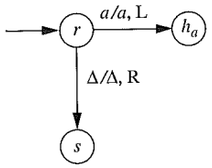


Figure 9.26 |

9.35. Assuming the same encoding method discussed in Section 9.7, and assuming that $s(0) = 00$ and $s(1) = 000$, draw the TM that is encoded by the string

0001 000101000010100011 00001001000010010011 00001000100001000100011
0000101000001010011 00000100100000010010011 00000100010000000100010001
000000101000000010010011 00000001010000000100010011
0000000010010000000010010011 000000001000100000000100010011
000000001010101011

9.36. Draw the portion of the universal TM T_u that is responsible for changing the tape symbol and moving the tape head after the search operation has identified the correct 5-tuple on tape 1. For example, the configuration

Δ000101000010100011000010010000010001001100001...
Δ01001$\underline{0}$010001Δ...
Δ$\underline{0}$000Δ...

would be transformed to

Δ$\underline{0}$00010100001010001100001001000001000100110001...
Δ01$\underline{0}$0100010001Δ...
Δ$\underline{0}$0000Δ...

9.37. Table 7.2 describes a PDA accepting the language *pal*. Draw a TM that accepts this language by simulating the PDA. You can make the TM nondeterministic, and you can use a second tape to represent the stack.

9.38. Suppose we define the *canonical order* of strings in $\{0, 1\}^*$ to be the order in which a string precedes any longer string and the order of two equal-length strings is numerical. For example, the strings 1, 01, 10, 000, 011, 100 are listed here in canonical order. Describe informally how to construct a TM T that enumerates the set of palindromes over $\{0, 1\}$ in canonical order. In other words, T loops forever, and for every positive integer n, there is some

point at which the initial portion of T's tape contains the string

$$\Delta\Delta 0\Delta 1\Delta 00\Delta 11\Delta 000\Delta \cdots \Delta x_n$$

where x_n is the nth palindrome in canonical order, and this portion of the tape is never subsequently changed.

MORE CHALLENGING PROBLEMS

9.39. Suppose you are watching a TM processing an input string, and that at each step you can see the configuration of the TM.

a. Suppose that for some n, the tape head does not move past square n while you are watching. If the pattern continues, will you be able to conclude at some point that the TM is in an infinite loop? If so, what is the longest you might need to watch in order to draw this conclusion?

b. Suppose that in each move you observe, the tape head moves right. If the pattern continues, will you be able to conclude at some point that the TM is in an infinite loop? If so, what is the longest you might need to watch in order to draw this conclusion?

9.40. In each of the following cases, show that the language accepted by the TM T is regular.

a. There is an integer n so that no matter what the input string is, T never moves its tape head to the right of square n.

b. For any $n \geq 0$ and any input of length n, T begins by making $n + 1$ moves in which the tape head is moved right each time, and thereafter T does not move the tape head to the left of square $n + 1$.

9.41. Suppose T is a TM. For each integer $i \geq 0$, denote by $n_i(T)$ the number of the rightmost square to which T has moved its tape head within the first i moves. (For example, if T moves its tape head right in the first five moves and left in the next three, then $n_i(T) = i$ for $i \leq 5$ and $n_i(T) = 5$ for $6 \leq i \leq 10$.) Suppose there is an integer k so that no matter what the input string is, $n_i(T) \geq i - k$ for every $i \geq 0$. Does it follow that $L(T)$ is regular? Give reasons for your answer.

9.42. Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a TM with a doubly-infinite tape (see the comments preceding Exercise 9.24). Show that there is an ordinary TM $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$, with $\Gamma \subseteq \Gamma_1$, satisfying these two conditions:

a. $L(T_1) = L(T)$.

b. For any $x \in \Sigma^*$, if $(q_0, \underline{\Delta}x) \vdash_T^* (h_a, y\underline{a}z)$, then $(q_1, \underline{\Delta}x) \vdash_{T_1}^* (h_a, y\underline{a}z)$.

The proof requires constructing an ordinary TM that can simulate the action of a TM having a doubly-infinite tape. There are several ways you might do this. One is to allow an ordinary tape to "look like" a *folded* doubly-infinite tape, using a technique similar to that in the proof of Theorem 9.1. Another would use even-numbered squares to represent squares indexed by

nonnegative numbers (i.e., the right half of the tape) and odd-numbered squares to represent the remaining squares.

9.43. In Figure 9.27 is a transition diagram for a TM M with a doubly-infinite tape. First, trace the moves it makes on the input string abb. Then, for the ordinary TM M_1 that you constructed in the previous exercise to simulate M, trace the moves that M_1 makes in simulating M on the same input.

9.44. Suppose M_1 is a two-tape TM, and M_2 is the ordinary TM constructed in Theorem 9.1 to simulate M_1. If M_1 requires n moves to process an input string x, give an upper bound on the number of moves M_2 requires in order to simulate the processing of x. Note that the number of moves M_1 has made places a limit on the position of its tape head. Try to make your upper bound as sharp as possible.

9.45. Show that if there is a TM T computing the function $f : \mathcal{N} \to \mathcal{N}$, then there is another one, T', whose tape alphabet is $\{1\}$. Suggestion: suppose T has tape alphabet $\Gamma = \{a_1, a_2, \ldots, a_n\}$. Encode Δ and each of the a_i's by a string of 1's and Δ's of length $n + 1$ (for example, encode Δ by $n + 1$ blanks, and a_i by $1^i \Delta^{n+1-i}$). Have T' simulate T, but using blocks of $n + 1$ tape squares instead of single squares.

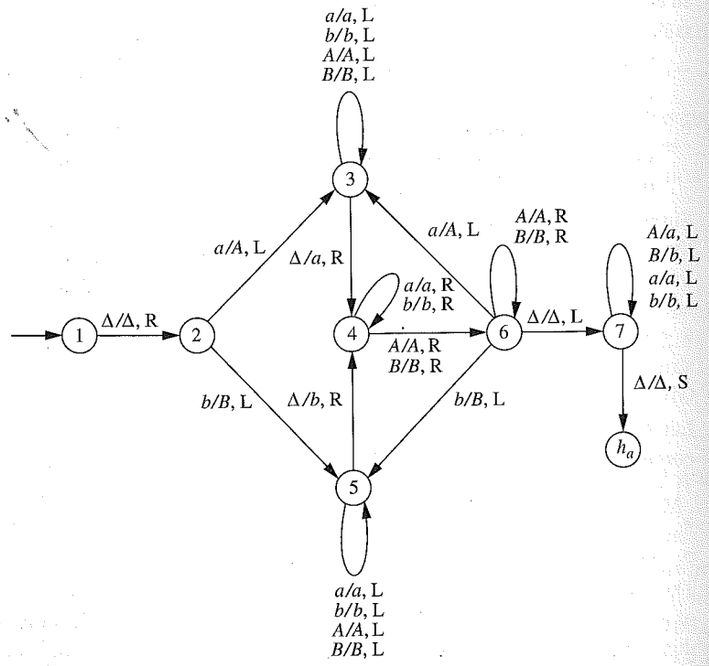


Figure 9.27 |

9.46. Describe how you could construct a TM T_0 that would accept input strings of 0's and 1's and would determine whether the input was a string of the form $e(T)$ for some TM T. ("Determine" means compute the characteristic function of the set of such encodings.)

9.47. Modify the construction in the proof of Theorem 9.2 so that if the NTM halts on every possible sequence of moves, the TM constructed to simulate it halts on every input.

9.48. Beginning with a nondeterministic Turing machine T_1, the proof of Theorem 9.2 shows how to construct an ordinary TM T_2 that accepts the same language. Suppose $|x| = n$, T_1 never has more than two choices of moves, and there is a sequence of n_x moves by which T_1 accepts x. Estimate as precisely as possible the number of moves that might be required for T_2 to accept x.

9.49. Formulate a precise definition of a *two-stack automaton*, which is like a PDA, except that it is deterministic and a move takes into account the symbols on top of both stacks and can replace either or both of them. Describe informally how you might construct a machine of this type accepting $\{a^i b^i c^i \mid i \geq 0\}$. Do it in a way that could be generalized to $\{a^i b^i c^i d^i \mid i \geq 0\}$, $\{a^i b^i c^i d^i e^i \mid i \geq 0\}$, etc.

9.50. Describe how a Turing machine can simulate a two-stack automaton; specifically, show that any language that can be accepted by a two-stack machine can be accepted by a TM.

9.51. A *Post machine* is similar to a PDA, but with the following differences. It is deterministic; it has an auxiliary queue instead of a stack, and the input is assumed to have been previously loaded onto the queue. For example, if the input string is abb, then the symbol currently at the front of the queue is a. Items can be added only to the rear of the queue, and deleted only from the front. Assume that there is a marker Z_0 initially on the queue following the input string (so that in the case of null input Z_0 is at the front). The machine can be defined as a 7-tuple $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, like a PDA. A single move depends on the state and the symbol currently at the front of the queue; and the move has three components: the resulting state, an indication of whether or not to remove the current symbol from the front of the queue, and what to add to the rear of the queue (a string, possibly null, of symbols from the queue alphabet).

Construct a Post machine to accept the language $\{a^n b^n c^n \mid n \geq 0\}$.

9.52. We can specify a configuration of a Post machine (see the previous exercise) by specifying the state and the contents of the queue. If the original marker Z_0 is currently in the queue, so that the string in the queue is of the form $\alpha Z_0 \beta$, then the queue can be thought of as representing the tape of a Turing machine, as follows. The marker Z_0 is thought of, not as an actual tape symbol, but as marking the right end of the string on the tape; the string β is at the beginning of the tape, followed by the string α; and the tape head is currently centered on the first symbol of α—or, if $\alpha = \Lambda$, on the first blank