CHAPTER

5

Regular and Nonregular Languages

5.1 | A CRITERION FOR REGULARITY

Kleene's theorem (Theorems 4.4 and 4.5) provides a useful characterization of regular languages: A language is regular (describable by a regular expression) if and only if it can be accepted by a finite automaton. In other words, a language can be *generated* in a simple way, from simple primitive languages, if and only if it can be *recognized* in a simple way, by a device with a finite number of states and no auxiliary memory. There is a construction algorithm associated with each half of the theorem, so that if we already have a regular expression, we can find an FA to accept the corresponding language, and if we already have an FA, we can find a regular expression to describe the language it accepts.

Suppose now that we have a language over the alphabet Σ specified in some way that involves neither a regular expression nor an FA. How can we tell whether it is regular? (What inherent property of a language identifies it as being regular?) And, if we suspect that the language is regular, how can we find either a regular expression describing it or an FA accepting it?

We have a partial answer to the first question already. According to Theorem 3.2, if there are infinitely many strings that are "pairwise distinguishable" with respect to L, then L cannot be regular. (To say it another way, if L is regular, then every set that is pairwise distinguishable with respect to L is finite.) It is useful to reformulate this condition slightly, using Definition 3.5. Recall that L/x denotes the set $\{y \in \Sigma^* \mid xy \in L\}$. If we let I_L be the indistinguishability relation on Σ^*, defined by

$$x I_L y \text{ if and only if } L/x = L/y$$

then I_L is an equivalence relation on Σ^* (Exercise 1.33b), and saying that two strings are distinguishable with respect to L means that they are in different equivalence classes of I_L. The statement above can therefore be reformulated as follows: If L is regular, then the set of equivalence classes for the relation I_L is finite. In order to obtain a characterization of regularity using this approach, we need to show that the converse is also true, that if the set of equivalence classes is finite then L is regular.

Once we do this, we will have an answer to the first question above (how can we tell whether L is regular?), in terms of the equivalence classes of the relation I_L. Furthermore, it turns out that if our language L is regular, identifying these equivalence classes will also give us an answer to the second question (how can we find an FA?), because there is a simple way to use the equivalence classes to construct an FA accepting L. This FA is *the* most natural one to accept L, in the sense that it has the fewest possible states; and an interesting by-product of the discussion will be a method for taking any FA known to accept L and simplifying it as much as possible.

We wish to show that if the the set of equivalence classes of I_L is finite, then there is a finite automaton accepting L. The discussion may be easier to understand, however, if we start with a language L known to be regular, and with an FA $M = (Q, \Sigma, q_0, A, \delta)$ recognizing L. If $q \in Q$, then adapting the notation introduced in Section 4.3, we let

$$L_q = \{x \in \Sigma^* \mid \delta^*(q_0, x) = q\}$$

We remember from Chapter 1 that talking about equivalence relations on a set is essentially the same as talking about partitions of the set: An equivalence relation determines a partition (in which the subsets are the equivalence classes), and a partition determines an equivalence relation (in which being equivalent means belonging to the same subset). At this point, there are two natural partitions of Σ^* that we might consider: the one determined by the equivalence relation I_L, and the one formed by all the sets L_q for $q \in Q$. The relationship between them is given by Lemma 3.1. If x and y are in the same L_q (in other words, if $\delta^*(q_0, x) = \delta^*(q_0, y)$), then $L/x = L/y$, so that x and y are in the same equivalence class of I_L. This means that each set L_q must be a subset of a single equivalence class, and therefore that every equivalence class of I_L is the union of one or more of the L_q's (Exercise 1.68). In particular, there can be no fewer of the L_q's than there are equivalence classes of I_L. If the two numbers are the same, then the two partitions are identical, each set L_q is precisely one of the equivalence classes of I_L, and M is an FA with the fewest possible states recognizing L.

These observations suggest how to turn things around and begin from the other end. If we have an FA accepting L, then under certain circumstances, the strings in one of the equivalence classes of I_L are precisely those that correspond to one of the states of the FA. If we are not given an FA to start with, but we know the equivalence classes of I_L, then we might try to construct an FA with exactly this property: Rather than starting with a state q and considering the corresponding set L_q of strings, this time we have the set of strings and we hope to specify a state to which the set will correspond. However, the point is that we do not have to *find* a state like this, we can simply *define* one. A "state" is an abstraction anyway; why not go ahead and say that a state is a set of strings—specifically, one of the equivalence classes of I_L? If there are only a finite number of these equivalence classes, then we have at least the first ingredient of a finite automaton accepting L: a finite set of states.

Once we commit ourselves to this abstraction, filling in the remaining details is surprisingly easy. Because one of the strings that cause an FA to be in the initial state is Λ, we choose for our initial state the equivalence class containing Λ. Because we want the FA to accept L, we choose for our accepting states those equivalence classes containing elements of L. And because in the recognition algorithm we change the current string by concatenating one more input symbol, we compute the value of our transition function by taking a string in our present state (equivalence class) and concatenating it with the input symbol. The resulting string determines the new equivalence class.

Before making our definition official, we need to look a little more closely at this last step. "Taking a string in our present state and concatenating it with the input symbol" means that if we start with an equivalence class q containing a string x, then $\delta(q, a)$ should be the equivalence class containing xa. Writing this symbolically, we should have

$$\delta([x], a) = [xa]$$

where for any string z, $[z]$ denotes the equivalence class containing z. As an assertion about a string x, this is a perfectly straightforward formula, which may be true or false, depending on how $\delta([x], a)$ is defined. If we want the formula to be the *definition* of $\delta([x], a)$, we must consider a potential problem. We are trying to define $\delta(q, a)$, where q is an equivalence class (a set of strings). We have taken a string x in the set q, which allows us to write $q = [x]$. However, there is nothing special about x; the set q could just as easily be written as $[y]$ for any other string $y \in q$. If our definition is to make any sense, it must tell us what $\delta(q, a)$ is, whether we write $q = [x]$ or $q = [y]$. The formula gives us $[xa]$ in one case and $[ya]$ in the other; obviously, unless $[xa] = [ya]$, our definition is nonsense. Fortunately, the next lemma takes care of the potential problem.

Lemma 5.1 I_L is *right invariant* with respect to concatenation. In other words, for any $x, y \in \Sigma^*$ and any $a \in \Sigma$, if $x \; I_L \; y$, then $xa \; I_L \; ya$. Equivalently, if $[x] = [y]$, then $[xa] = [ya]$.

Proof Suppose $x \; I_L \; y$ and $a \in \Sigma$. Then $L/x = L/y$, so that for any $z' \in \Sigma^*$, xz' and yz' are either both in L or both not in L. Therefore, for any $z \in \Sigma^*$, xaz and yaz are either both in L or both not in L (because we can apply the previous statement with $z' = az$), and we conclude that $xa I_L ya$. ■

Theorem 5.1

Let $L \subseteq \Sigma^*$, and let Q_L be the set of equivalence classes of the relation I_L on Σ^*. (Each element of Q, therefore, is a set of strings.) If Q_L is a finite set, then $M_L = (Q_L, \Sigma, q_0, A_L, \delta)$ is a finite automaton accepting L, where $q_0 = [\Lambda]$, $A_L = \{q \in Q_L \mid q \cap L \neq \emptyset\}$, and $\delta : Q_L \times \Sigma \to Q_L$ is defined by the formula $\delta([x], a) = [xa]$. Furthermore, M_L has the fewest states of any FA accepting L.

Proof

According to Lemma 5.1, the formula $\delta([x], a) = [xa]$ is a meaningful definition of a function δ from $Q_L \times \Sigma$ to Q_L, and thus the 5-tuple M_L has all the ingredients of an FA. In order to verify that M_L recognizes L, we need the formula

$$\delta^*([x], y) = [xy]$$

for $x, y \in \Sigma^*$. The proof is by structural induction on y. The basis step is to show that $\delta^*([x], \Lambda) = [x\Lambda]$ for every x. This is easy, since the left side is $[x]$ because of the definition of δ^* in an FA (Definition 3.3), and the right side is $[x]$ because $x\Lambda = x$.

For the induction step, suppose that for some y, $\delta^*([x], y) = [xy]$ for every string x, and consider $\delta^*([x], ya)$ for $a \in \Sigma$:

$$\begin{aligned} \delta^*([x], ya) &= \delta(\delta^*([x], y), a) \quad \text{(by definition of } \delta^*) \\ &= \delta([xy], a) \quad \text{(by the induction hypothesis)} \\ &= [xya] \quad \text{(by the definition of } \delta) \end{aligned}$$

From this formula it follows that $\delta^*(q_0, x) = \delta^*([\Lambda], x) = [x]$. Our definition of A tells us, therefore, that x is accepted by M_L if and only if $[x] \cap L \neq \emptyset$. What we want is that x is accepted if and only if $x \in L$. But in fact the two statements $[x] \cap L \neq \emptyset$ and $x \in L$ are the same. One direction is obvious: If $x \in L$, then $[x] \cap L \neq \emptyset$, since $x \in [x]$. In the other direction, if $[x]$ contains an element y of L, then x must be in L. Otherwise the string Λ would distinguish x and y with respect to L, and x and y could not both be elements of $[x]$. Therefore, M_L accepts L.

Finally, if there are n equivalence classes of I_L, then we can get a set of n strings that are pairwise distinguishable by just choosing one string from each equivalence class. (No two of these strings are equivalent, or any two are distinguishable.) Theorem 3.2 implies that any FA accepting L must have at least n states. Since M_L has exactly n, it has the fewest possible.

Corollary 5.1 L is a regular language if and only if the set of equivalence classes of I_L is finite.

Proof Theorem 5.1 tells us that if the set of equivalence classes is finite, there is an FA accepting L; and Theorem 3.2 says that if the set is infinite, there can be no such FA. ■

Corollary 5.1 was proved by Myhill and Nerode, and it is often called the Myhill-Nerode theorem.

It is interesting to observe that to some extent, the construction of M_L in Theorem 5.1 makes sense even when the language L is not regular. I_L is an equivalence relation for any language L, and we can consider the set Q_L of equivalence classes. Neither

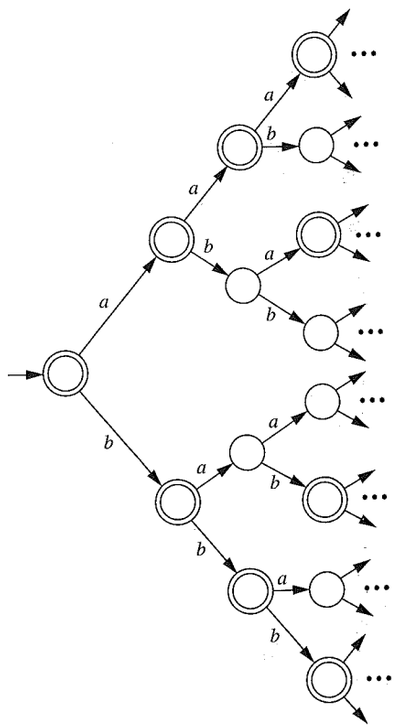


Figure 5.1 |

the definition of $\delta : Q_L \times \Sigma \to Q_L$ nor the proof that M_L accepts L depends on the assumption that Q_L is a finite set. It appears that even in the most general case, we have some sort of "device" that accepts L. If it is not a finite automaton, what is it?

Instead of inventing a name for something with an infinite number of states, let us draw a (partial) picture of it in a simple case we have studied. Let L be the language *pal* of all palindromes over $\{a, b\}$. As we observed in the proof of Theorem 3.3, any two strings in $\{a, b\}^*$ are distinguishable with respect to L. Not only is the set Q_L infinite, but there are as many equivalence classes as there are strings; each equivalence class contains exactly one string. Even in this most extreme case, there is no difficulty in visualizing M_L, as Figure 5.1 indicates.

The only problem, of course, is that there is no way to complete the picture, and no way to implement M_L as a physical machine. As we have seen in other ways, the crucial aspect of an FA is precisely the finiteness of the set of states.

EXAMPLE 5.1 Applying Theorem 5.1 to $\{0, 1\}^*\{10\}$

Consider the language discussed in Example 3.12,

$$L = \{x \in \{0, 1\}^* \mid x \text{ ends with } 10\}$$

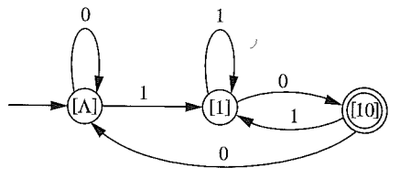


Figure 5.2 |
A minimum-state FA recognizing $\{0, 1\}^*\{10\}$.

and consider the three strings Λ, 1, and 10. We can easily verify that any two of these strings are distinguishable with respect to L: The string Λ distinguishes Λ and 10, and also 1 and 10, while the string 0 distinguishes Λ and 1. Therefore, the three equivalence classes $[\Lambda]$, [1], and [10] are distinct.

However, any string y is equivalent to (indistinguishable from) one of these strings. If y ends in 10, then y is equivalent to 10; if y ends in 1, y is equivalent to 1; otherwise (if $y = \Lambda$, $y = 0$, or y ends with 00), y is equivalent to Λ. Therefore, these three equivalence classes are the only ones.

Let $M_L = (Q_L, \{0, 1\}, [\Lambda], \{[10]\}, \delta)$ be the FA we constructed in Theorem 5.1. Then

$$\delta([\Lambda], 0) = [\Lambda] \quad \text{and} \quad \delta([\Lambda], 1) = [1]$$

since $\Lambda 0$ is equivalent to Λ and $\Lambda 1 = 1$. Similarly,

$$\delta([1], 0) = [10] \quad \text{and} \quad \delta([1], 1) = [1]$$

since 11 is equivalent to 1. Finally,

$$\delta([10], 0) = [\Lambda] \quad \text{and} \quad \delta([10], 1) = [1]$$

since 100 is equivalent to Λ and 101 is equivalent to 1. It follows that the FA M_L is the one shown in Figure 5.2. Not surprisingly, this is the same FA we came up with in Example 3.12, except for the names given to the states. (One reason it is not surprising is that the strings Λ, 1, and 10 were chosen to correspond to the three states in the previous FA!)

We used Theorem 3.2, which is the "only if" part of Theorem 5.1, to show that the language of palindromes over $\{0, 1\}$ is nonregular. We may use the same principle to exhibit a number of other nonregular languages.

EXAMPLE 5.2 The Equivalence Classes of l_L for $L = \{0^n1^n \mid n > 0\}$

Let $L = \{0^n1^n \mid n > 0\}$. The intuitive reason L is not regular is that in trying to recognize elements of L, we must remember how many 0's we have seen, so that when we start seeing 1's we will be able to determine whether the number of 1's is exactly the same. In order to use Theorem 5.1 to show L is not regular, we must show that there are infinitely many distinct equivalence classes of l_L. In this example, at least, let us do even more than that and describe the equivalence classes exactly.

Some strings are not prefixes of any elements of L (examples include 1, 011, and 010), and it is not hard to see that the set of all such strings is an equivalence class (Exercise 5.4). The remaining strings are of three types: strings in L, strings of 0's (0^i, for some $i \geq 0$), and strings of the form $0^i 1^j$ with $0 < j < i$.

The set L is itself an equivalence class of I_L. This is because for any string $x \in L$, Λ is the only string that can follow x so as to produce an element of L.

Saying as we did above that "we must remember how many 0's we have seen" suggests that two distinct strings of 0's should be in different equivalence classes. This is true: If $i \neq j$, the strings 0^i and 0^j are distinguished by the string 1^i, because $0^i 1^i \in L$ and $0^j 1^i \notin L$. We now know that $[0^i] \neq [0^j]$. To see exactly what these sets are, we note that for *any* string x other than 0^i, the string 01^{i+1} distinguishes 0^i and x (because $0^i 01^{i+1} \in L$ and $x01^{i+1} \notin L$). In other words, 0^i is equivalent only to itself, and $[0^i] = \{0^i\}$.

Finally, consider the string 000011, for example. There is exactly one string z for which $000011z \in L$: the string $z = 11$. However, any other string x having the property that $xz \in L$ if and only if $z = 11$ is equivalent to 000011, and these are the strings $0^{j+2}1^j$ for $j > 0$. No string other than one of these can be equivalent to 000011, and we may conclude that the equivalence class [000011] is the set $\{0^{j+2}1^j \mid j > 0\}$. Similarly, for each $k > 0$, the set $\{0^{j+k}1^j \mid j > 0\}$ is an equivalence class.

Let us summarize our conclusions. The set L and the set of all nonprefixes of elements of L are two of the equivalence classes; for each $i \geq 0$, the set with the single element 0^i is an equivalence class; and for each $k > 0$, the infinite set $\{0^{j+k}1^j \mid j > 0\}$ is an equivalence class. Since every string is in one of these equivalence classes, these are the only equivalence classes.

As we expected, we have shown in particular that there are infinitely many distinct equivalence classes, which allows us to conclude that L is not regular.

EXAMPLE 5.3 Simple Algebraic Expressions

Let L be the set of all legal algebraic expressions involving the identifier a, the operator $+$, and left and right parentheses. To show that the relation I_L has infinitely many distinct equivalence classes, we can ignore much of the structure of L. The only fact we use is that the string

$$((\cdots(a)\cdots))$$

is in L if and only if the numbers of left and right parentheses are the same. We may therefore consider the set $S = \{(^n \mid n \geq 0\}$, in the same way that we considered the strings 0^n in the previous example. For $0 \leq m < n$, the string $a)^m$ distinguishes $(^m$ and $(^n$, and so any two elements of S are distinguishable with respect to L (i.e., in different equivalence classes). We conclude from this that L is not regular. Exercise 5.35 asks you to describe the equivalence classes of I_L more precisely.

EXAMPLE 5.4 The Set of Strings of the Form *ww*

For yet another example where the set $S = \{0^n \mid n \geq 0\}$ can be used to prove a language nonregular, take L to be the language

$$\{ww \mid w \in \{0, 1\}^*\}$$

of all even-length strings of 0's and 1's whose first and second halves are identical. This time, for a string z that distinguishes 0^n and 0^m when $m \neq n$, we choose $z = 1^n 0^n 1^n$. The string $0^n z$ is in L, and the string $0^m z$ is not.

Exercise 5.20 asks you to get even a little more mileage out of the set $\{0^n \mid n \geq 0\}$ or some variation of it. We close this section with one more example.

EXAMPLE 5.5 Another Nonregular Language from Theorem 5.1

Let $L = \{0, 011, 011000, 0110001111, \ldots\}$. A string in L consists of groups of 0's alternated with groups of 1's. It begins with a single 0, and each subsequent group of identical symbols is one symbol longer than the previous group. Here we can show that the infinite set L itself is pairwise distinguishable with respect to L, and therefore that L is not regular. Let x and y be two distinct elements of L. Suppose x and y both end with groups of 0's, for example, x with 0^j and y with 0^k. Then $x1^{j+1} \in L$, but $y1^{j+1} \notin L$. It is easy to see that similar arguments also work in the other three cases.

5.2 | MINIMAL FINITE AUTOMATA

Theorem 5.1 and Corollary 5.1 help us to understand a little better what makes a language L regular. In a sense they provide an absolute answer to the question of how much information we need to remember at each step of the recognition algorithm: We can forget everything about the current string *except* which equivalence class of I_L it belongs to. If there are infinitely many of these equivalence classes, then this is more information than any FA can remember, and L cannot be regular. If the set of equivalence classes is finite, and if we can identify them, then we can use them to construct the simplest possible FA accepting L.

It is not clear in general just how these equivalence classes are to be identified or described precisely; in Example 5.1, where we did it by finding three pairwise distinguishable strings, we had a three-state FA recognizing L to start with, so that we obtained little or no new information. In this section we will show that as long as we have *some* FA to start with, we can always find the simplest possible one. We will develop an algorithm for taking an arbitrary FA and modifying it if necessary so that the resulting machine has the fewest possible states (and the states correspond exactly to the equivalence classes of I_L).

Suppose we begin with the finite automaton $M = (Q, \Sigma, q_0, A, \delta)$. We consider again the two partitions of Σ^* that we described in Section 5.1, one in which the subsets are the sets L_q and one in which the subsets are the equivalence classes of I_L. If the two partitions are the same, then we already have the answer we want, and M is already a minimum-state FA. If not, the fact that the first partition is finer than the second (one subset from the second partition might be the union of several from the first) tells us that we do not need to abandon our FA and start over, looking for one with fewer states; we just have to determine which sets L_q we can combine to obtain an equivalence class.

Before we attack this problem, there is one obvious way in which we might be able to reduce the number of states in M without affecting the L_q partition at all. This is to eliminate the states q for which $L_q = \emptyset$. For such a q, there are no strings x satisfying $\delta^*(q_0, x) = q$; in other words, q is unreachable from q_0. It is easy to formulate a recursive definition of the set of reachable states of M and then use that to obtain an algorithm that finds all reachable states. If all the others are eliminated, the resulting FA still recognizes L (Exercise 3.29). For the remainder of this discussion, therefore, we assume that all states of M are reachable from q_0.

It may be helpful at this point to look again at Example 3.12. Figure 5.3*a* shows the original FA we drew for this language; Figure 5.3*b* shows the minimum-state FA we arrived at in Example 5.1; Figure 5.3*c* shows the partition corresponding to the original FA, with seven subsets; and Figure 5.3*d* shows the three equivalence classes of I_L, which are the sets L_q for the minimum-state FA. We obtain the simpler FA from the first one by merging the three states 1, 2, and 4 into the state A, and by merging the states 3, 5, and 7 into B. State 6 becomes state C. Once we have done this, we can easily determine the new transitions. From any of the states 1, 2, and 4, the input symbol 0 takes us to one of those same states. Therefore, the transition from A with input 0 must go to A. From 1, 2, or 4, the input 1 takes us to 3, 5, or 7. Therefore, the transition from A with input 1 goes to B. The other cases are similar.

In general, starting with a finite automaton M, we may describe the problem in terms of identifying the *pairs* (p, q) of states for which L_p and L_q are subsets of the same equivalence class. Let us write this condition as $p \equiv q$. What we will actually do is to solve the opposite problem: identify those pairs (p, q) for which $p \not\equiv q$. The first step is to express the statement $p \equiv q$ in a slightly different way.

Lemma 5.2 Suppose $p, q \in Q$, and x and y are strings with $x \in L_p$ and $y \in L_q$ (in other words, $\delta^*(q_0, x) = p$ and $\delta^*(q_0, y) = q$). Then these three statements are all equivalent:

1. $p \equiv q$.
2. $L/x = L/y$ (i.e., $x I_L y$, or x and y are indistinguishable with respect to L).
3. For any $z \in \Sigma^*$, $\delta^*(p, z) \in A \Leftrightarrow \delta^*(q, z) \in A$ (i.e., $\delta^*(p, z)$ and $\delta^*(q, z)$ are either both in A or both not in A).

Proof To see that statements 2 and 3 are equivalent, we begin with the formulas

$$\delta^*(p, z) = \delta^*(\delta^*(q_0, x), z) = \delta^*(q_0, xz)$$
$$\delta^*(q, z) = \delta^*(\delta^*(q_0, y), z) = \delta^*(q_0, yz)$$

Saying that $L/x = L/y$ means that a string z is in one set if and only if it is in the other, or that $xz \in L$ if and only if $yz \in L$; since M accepts L, this is exactly the same as statement 3.

Now if statement 1 is true, then L_p and L_q are both subsets of the same equivalence class. This means that x and y are equivalent, which is statement 2. The converse is also true, because we know that if L_p and L_q are not both subsets of the same equivalence class, then they are subsets of different equivalence classes, so that statement 2 does not hold. ■

Let us now consider how it can happen that $p \not\equiv q$. According to the lemma, this means that for some z, exactly one of the two states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A.

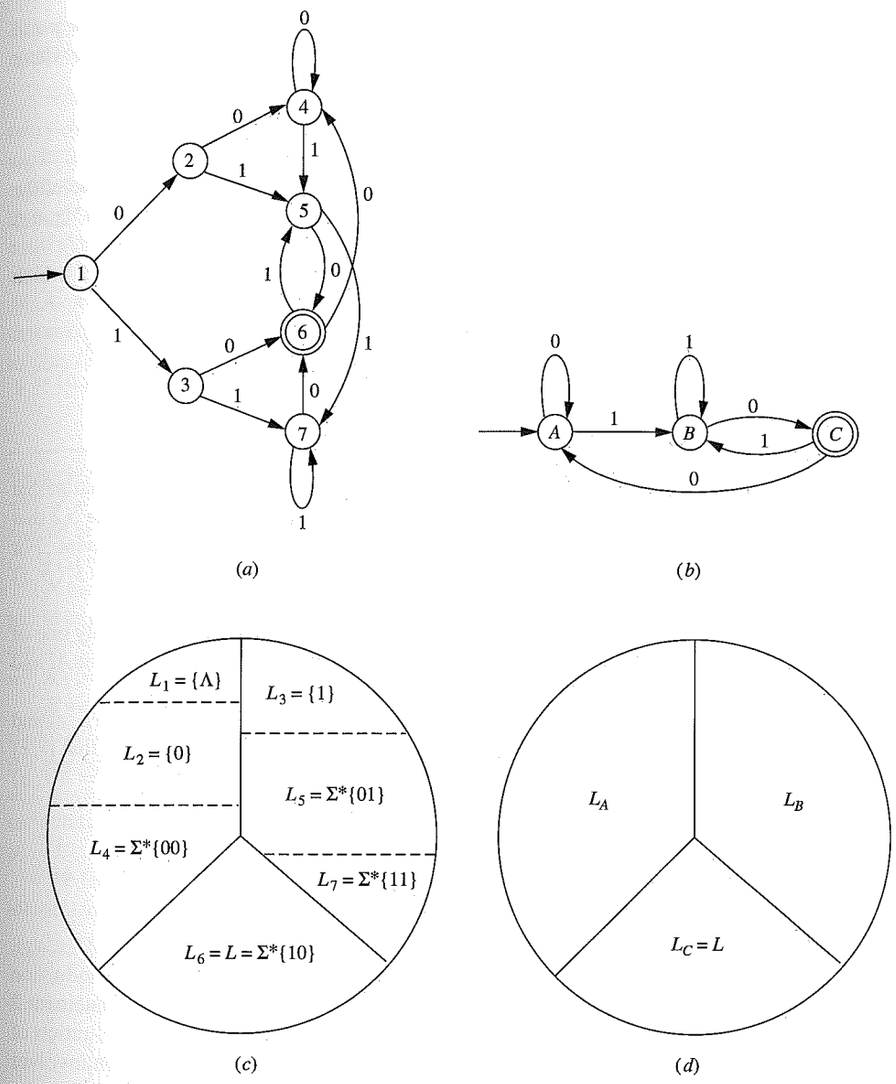


Figure 5.3 |
Two FAs for $\{0, 1\}^*\{10\}$ and the corresponding partitions of $\{0, 1\}^*$.

The simplest way this can happen is with $z = \Lambda$, so that only one of the states p and q is in A. Once we have one pair (p, q) with $p \not\equiv q$, we consider the situation where $r, s \in Q$, and for some $a \in \Sigma$, $\delta(r, a) = p$ and $\delta(s, a) = q$. We may write

$$\delta^*(r, az) = \delta^*(\delta^*(r, a), z) = \delta^*(\delta(r, a), z) = \delta^*(p, z)$$

and similarly, $\delta^*(s, az) = \delta^*(q, z)$. Since $p \not\equiv q$, then for some z, exactly one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A; therefore, exactly one of $\delta^*(r, az)$ and $\delta^*(s, az)$ is in A, and $r \not\equiv s$.

These observations suggest the following recursive definition of a set S, which will turn out to be the set of all pairs (p, q) with $p \not\equiv q$.

1. For any p and q for which exactly one of p and q is in A, (p, q) is in S.
2. For any pair $(p, q) \in S$, if (r, s) is a pair for which $\delta(r, a) = p$ and $\delta(s, a) = q$ for some $a \in \Sigma$, then (r, s) is in S.
3. No other pairs are in S.

It is not difficult to see from the comments preceding the recursive definition that for any pair $(p, q) \in S$, $p \not\equiv q$. On the other hand, it follows from Lemma 5.2 that we can show S contains all such pairs by establishing the following statement: For any string $z \in \Sigma^*$, every pair of states (p, q) for which only one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A is an element of S.

We do this by using structural induction on z. For the basis step, if only one of $\delta^*(p, \Lambda)$ and $\delta^*(q, \Lambda)$ is in A, then only one of the two states p and q is in A, and $(p, q) \in S$ because of statement 1 of the definition.

Now suppose that for some z, all pairs (p, q) for which only one of $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A are in S. Consider the string az, where $a \in \Sigma$, and suppose that (r, s) is a pair for which only one of $\delta^*(r, az)$ and $\delta^*(s, az)$ is in A. If we let $p = \delta(r, a)$ and $q = \delta(s, a)$, then we have

$$\delta^*(r, az) = \delta^*(\delta(r, a), z) = \delta^*(p, z)$$
$$\delta^*(s, az) = \delta^*(\delta(s, a), z) = \delta^*(q, z)$$

Our assumption on r and s is that only one of the states $\delta^*(r, az)$ and $\delta^*(s, az)$, and therefore only one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$, is in A. Our induction hypothesis therefore implies that $(p, q) \in S$, and it then follows from statement 2 in the recursive definition that $(r, s) \in S$. (Note that in the recursive definition of Σ^* implicit in this structural induction, the recursive step of the definition involves strings of the form az rather than za.)

Now it is a simple matter to convert this recursive definition into an algorithm to identify all the pairs (p, q) for which $p \not\equiv q$.

Algorithm 5.1 (For Identifying the Pairs (p, q) with $p \not\equiv q$) List all (unordered) pairs of states (p, q) for which $p \neq q$. Make a sequence of passes through these pairs. On the first pass, mark each pair of which exactly one element is in A. On each subsequent pass, mark any pair (r, s) if there is an $a \in \Sigma$ for which $\delta(r, a) = p$, $\delta(s, a) = q$, and (p, q) is already marked. After a pass in which no new pairs are marked, stop. The marked pairs (p, q) are precisely those for which $p \not\equiv q$. ■

When the algorithm terminates, any pair (p, q) that remains unmarked represents two states in our FA that can be merged into one, since the corresponding sets of strings are both subsets of the same equivalence class. In order to find the total number of equivalence classes, or the minimum number of states, we can make one final pass through the states of M; the first state of M to be considered corresponds to one equivalence class; for each subsequent state q of M, q represents a new equivalence class only if the pair (p, q) was marked by Algorithm 5.1 for every previous state p of M. As we have seen in our example, once we have the states in the minimum-state FA, determining the transitions is straightforward. We return once more to Example 5.1 to illustrate the algorithm.

Minimizing the FA in Figure 5.3a — EXAMPLE 5.6

We apply Algorithm 5.1 to the FA in Figure 5.3*a*. Figure 5.4*a* shows all unordered pairs (p, q) with $p \neq q$. The pairs marked 1 are those of which exactly one element is in A; they are marked on pass 1. The pairs marked 2 are those marked on the second pass. For example, $(2, 5)$ is one of these, since $\delta(2, 0) = 4$, $\delta(5, 0) = 6$, and the pair $(4, 6)$ was marked on pass 1.

A third pass produces no new marked pairs. Suppose for example that $(1, 2)$ is the first pair to be tested on the third pass. We calculate $\delta(1, 0) = 2$ and $\delta(2, 0) = 4$, and $(2, 4)$ is not marked. Similarly, $\delta(1, 1) = 3$ and $\delta(2, 1) = 5$, and $(3, 5)$ is not marked. It follows that $(1, 2)$ will not be marked on this pass.

If we now go through the seven states in numerical order, we see that there is an equivalence class containing state 1; state 2 is in the same class, since the pair $(1, 2)$ is unmarked; state 3 is in a new equivalence class, since $(1, 3)$ and $(2, 3)$ are both marked; state 4 is in the same class as 2, since $(2, 4)$ is unmarked; 5 is in the same class as 3; 6 is in a new class; and 7 is in the same class as 3. We conclude that the three equivalence classes of I_L are $p_1 = L_1 \cup L_2 \cup L_4$, $p_2 = L_3 \cup L_5 \cup L_7$, and $p_3 = L_6$.

We found the transitions earlier. The resulting FA is shown in Figure 5.4*b*. Again, it is identical to the one obtained in Example 3.12 except for the names of the states.

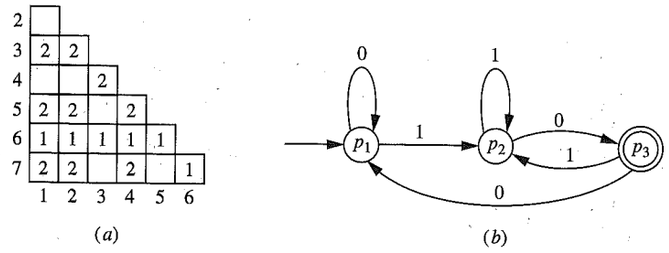


Figure 5.4 |
Applying Algorithm 5.1 to the FA in Figure 5.3*a*.

5.3 THE PUMPING LEMMA FOR REGULAR LANGUAGES

Every regular language can be accepted by a finite automaton, a recognizing device with a finite set of states and no auxiliary memory. We can use the finiteness of this set to derive another property shared by all regular languages. Showing that a language does not have this property will then be another way, in addition to using Corollary 5.1, of showing that the language is not regular. One reason this is useful is that the method we come up with can be adapted for use with more general languages, as we will see in Chapter 8.

Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA recognizing a language L. The property we are interested in has to do with paths through M that contain "loops." An input string $x \in L$ requiring M to enter some state twice corresponds to a path that starts at q_0, ends at some accepting state q_f, and contains a loop. (See Figure 5.5.) Any other path obtained from this one by changing the number of traversals of the loop will then also correspond to an element of L, different from x in that it contains a different number of occurrences of the substring corresponding to the loop. This simple observation will lead to the property we want.

Suppose that the set Q has n elements. For any string x in L with length at least n, if we write $x = a_1a_2 \cdots a_n y$, then the sequence of $n + 1$ states

$$\begin{aligned} q_0 &= \delta^*(q_0, \Lambda) \\ q_1 &= \delta^*(q_0, a_1) \\ q_2 &= \delta^*(q_0, a_1a_2) \\ &\cdots \\ q_n &= \delta^*(q_0, a_1a_2 \cdots a_n) \end{aligned}$$

must contain some state at least twice, by the pigeonhole principle (Exercise 2.44). This is where our loop comes from. Suppose $q_i = q_{i+p}$, where $0 \le i < i + p \le n$. Then

$$\begin{aligned} \delta^*(q_0, a_1a_2 \cdots a_i) &= q_i \\ \delta^*(q_i, a_{i+1}a_{i+2} \cdots a_{i+p}) &= q_i \\ \delta^*(q_i, a_{i+p+1}a_{i+p+2} \cdots a_n y) &= q_f \in A \end{aligned}$$

To simplify the notation, let

$$\begin{aligned} u &= a_1a_2 \cdots a_i \\ v &= a_{i+1}a_{i+2} \cdots a_{i+p} \end{aligned}$$

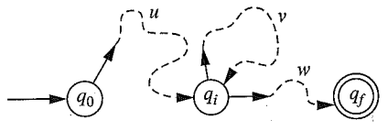


Figure 5.5

$$w = a_{i+p+1}a_{i+p+2} \cdots a_n y$$

(See Figure 5.5.) The string u is interpreted to be Λ if $i = 0$, and w is interpreted to be y if $i + p = n$.

Since $\delta^*(q_i, v) = q_i$, we have $\delta^*(q_i, v^m) = q_i$ for every $m \ge 0$, and it follows that $\delta^*(q_0, uv^m w) = q_f$ for every $m \ge 0$. Since $p > 0$ and $i + p \le n$, we have proved the following result.

Theorem 5.2

Suppose L is a regular language recognized by a finite automaton with n states. For any $x \in L$ with $|x| \ge n$, x may be written as $x = uvw$ for some strings u, v, and w satisfying

$$|uv| \le n$$
$$|v| > 0$$
$$\text{for any } m \ge 0,\ uv^m w \in L$$

This result is often referred to as the Pumping Lemma for Regular Languages, since we can think of it as saying that for an arbitrary string in L, provided it is sufficiently long, a portion of it can be "pumped up," introducing additional copies of the substring v, so as to obtain many more distinct elements of L.

The proof of the result was easy, but the result itself is complicated enough in its logical structure that applying it correctly requires some care. It may be helpful first to weaken it slightly by leaving out some information (where the integer n comes from). Theorem 5.2a clarifies the essential feature and is sufficient for most applications.

Theorem 5.2a The Pumping Lemma for Regular Languages

Suppose L is a regular language. Then there is an integer n so that for any $x \in L$ with $|x| \ge n$, there are strings u, v, and w so that

$$x = uvw \quad (5.1)$$
$$|uv| \le n \quad (5.2)$$
$$|v| > 0 \quad (5.3)$$
$$\text{for any } m \ge 0,\ uv^m w \in L \quad (5.4)$$

In order to use the pumping lemma to show that a language L is not regular, we must show that L fails to have the property described in the lemma. We do this by assuming that the property is satisfied and deriving a contradiction.

The statement is of the form "There is an n so that for any $x \in L$ with $|x| \ge n, \ldots$" We assume, therefore, that we have such an n, although we do not know what it is. We try to find a specific string x with $|x| \ge n$ so that the statements involving x in the theorem will lead to a contradiction. (The theorem says that under the assumption that L is regular, *any* $x \in L$ with $|x| \ge n$ satisfies certain conditions; therefore,

our specific x satisfies these conditions; this leads to a contradiction; therefore, the assumption leads to a contradiction; therefore, L is not regular.)

Remember, however, that we do not know what n is. In effect, therefore, we must show that for *any* n, we can find an $x \in L$ with $|x| \geq n$ so that the statements about x in the theorem lead to a contradiction. It may be that we have to choose x carefully in order to obtain a contradiction. We are free to pick any x we like, as long as $|x| \geq n$—but since we do not know what n is, the choice of x must involve n.

Once we have chosen x, we are *not* free to choose the strings u, v, and w into which the theorem says x can be decomposed. What we know is that there is *some* way to write x as uvw so that equations (5.2)–(5.4) are true. Because we must guarantee that a contradiction is produced, we must show once we have chosen x that *any* choice of u, v, and w satisfying equations (5.1)–(5.4) produces a contradiction. Let us use as our first illustration one of the languages that we already know is not regular.

EXAMPLE 5.7 Application of the Pumping Lemma

Let $L = \{0^i 1^i \mid i \geq 0\}$. Suppose that L is regular, and let n be the integer in Theorem 5.2a. We can now choose any x with $|x| \geq n$; a reasonable choice is $x = 0^n 1^n$. The theorem says that $x = uvw$ for some u, v, and w satisfying equations (5.2)–(5.4). No matter what u, v, and w are, the fact that (5.2) is true implies that $uv = 0^k$ for some k, and it follows from (5.3) that $v = 0^j$ for some $j > 0$. Equation (5.4) says that $uv^m w \in L$ for every $m \geq 0$. However, we can obtain a contradiction by considering $m = 2$. The string uv^2w contains j extra 0's in the first part ($uv^2w = 0^{n+j}1^n$), and cannot be in L because $j > 0$. This contradiction allows us to conclude that L cannot be regular.

Let us look a little more closely at the way we chose x in this example (which for this language just means the way we chose $|x|$). In the statement of the pumping lemma, the only condition x needs to satisfy is $|x| \geq n$; with a little more effort, we can obtain a contradiction by starting with $x = 0^m 1^m$, for any $m \geq n/2$. However, now we can no longer assert that $uv = 0^k$. There are two other possibilities to consider. In each case, however, looking at $m = 2$ is enough to obtain a contradiction. If v contains both 0's and 1's, then $v = 0^i 1^j$, so that uv^2w contains the substring 10 and is therefore not in L. If v contains only 1's, then $v = 1^j$, and $uv^2w = 0^n 1^{n+j}$, also not in L.

Again, the point is that when we use the pumping lemma to show L is nonregular, we are free to choose x any way we wish, as long as $|x| \geq n$ and as long as it will allow us to derive a contradiction. We try to choose x so that getting the contradiction is as simple as possible. Once we have chosen x, we must be careful to show that a contradiction follows inevitably. Unless we can get a contradiction in every conceivable case, we have not accomplished anything.

Another feature of this example is that it allows us to prove more than we originally set out to. We started with the string $x = 0^n 1^n \in L$ and observed that for the strings u, v, and w, uv^2w fails to be an element, not only of L but of the larger language $L_1 = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$. Therefore, our proof also allows us to conclude that L_1 is not regular.

However, with this larger language it is worth looking one more time at the initial choice of x, because specifying a length no longer determines the string, and not all strings of the same length are equally suitable. First we observe that choosing $x = 0^{n/2}1^{n/2}$, which would have worked for the language L (at least if n is even), no longer works for L_1. The reason this string works for L is that even if v happens to contain both 0's and 1's, uv^2w is not of the form $0^i 1^i$. (The contradiction is obtained, not by looking at the *numbers* of 0's and 1's, but by looking at the order of the symbols.) The reason it does not work for L_1 is that if v contains *equal* numbers of 0's and 1's, then $uv^m w$ also has equal numbers of 0's and 1's, no matter what m we use, and there is no contradiction.

If we had set out originally to show that L_1 was not regular, we might have chosen an x in L_1 but not in L. An example of an inappropriate choice is the string $x = (01)^n$. Although this string is in L_1 and its length is at least n, look what happens when we try to produce a contradiction. If $x = uvw$, we have these possibilities:

1. $v = (01)^j$ for some $j > 0$
2. $v = 1(01)^j$ for some $j \geq 0$
3. $v = 1(01)^j 0$ for some $j \geq 0$
4. $v = (01)^j 0$ for some $j \geq 0$

Unfortunately, none of the conditions (5.2)–(5.4) gives us any more information about v, except for some upper bounds on j. In cases 2 and 4 we can obtain a contradiction because the string v that is being pumped has unequal numbers of 0's and 1's. In the other two cases, however, there is no contradiction, because $uv^m w$ has equal numbers of 0's and 1's for any m. We cannot guarantee that one of these cases does not occur, and therefore we are unable to finish the proof using this choice of x.

EXAMPLE 5.8 Another Application of the Pumping Lemma

Consider the language

$$L = \{0^i x \mid i \geq 0, x \in \{0, 1\}^* \text{ and } |x| \leq i\}$$

Another description of L is that it is the set of all strings of 0's and 1's so that at least the first half of x consists of 0's. The proof that L is not regular starts the same way as in the previous example. Assume that L is regular, and let n be the integer in Theorem 5.2a. We obviously should not try to start with a string x of all 0's, because then no string obtained from x by pumping could have any 1's, and there would be no chance of a contradiction. Suppose we try $x = 0^n 1^n$, just as in the previous example. Then if Equations (5.1)–(5.4) hold, it follows as before that $v = 0^j$ for some $j > 0$. In this example the term *pumping* is a little misleading. We cannot obtain a contradiction by looking at strings with additional copies of v, because initial 0's account for an even larger fraction of these strings than in x. However, Equation (5.4) also says that $uv^0 w \in L$. This does give us our contradiction, because $uv^0 w = uw = 0^{n-j}1^n \notin L$. Therefore, L is not regular.

EXAMPLE 5.9 Application of the Pumping Lemma to *pal*

Let L be *pal*, the languages of palindromes over $\{0, 1\}$. We know from Theorem 3.3 that L is not regular, and now we can also use the pumping lemma to prove this. Suppose that L is regular, and let n be the integer in the statement of the pumping lemma. We must choose x to be a palindrome of length at least n that will produce a contradiction; let us try $x = 0^n 10^n$.

Then just as in the two previous examples, if Equations (5.1)–(5.4) are true, the string v is a substring of the form 0^j (with $j > 0$) from the first part of x. We can obtain a contradiction using either $m = 0$ or $m > 1$. In the first case, $uv^m w = 0^{n-j}10^n$, and in the second case, if $m = 2$ for example, $uv^m w = 0^{n+j}10^n$. Neither of these is a palindrome, and it follows that L cannot be regular.

It is often possible to get by with a weakened form of the pumping lemma. Here are two versions that leave out many of the conclusions of Theorem 5.2a but are still strong enough to show that certain languages are not regular.

Theorem 5.3 Weak Form of Pumping Lemma
Suppose L is an infinite regular language. Then there are strings u, v, and w so that $|v| > 0$ and $uv^m w \in L$ for every $m \geq 0$.

Proof
This follows immediately from Theorem 5.2a. No matter how big the integer n in the statement of that theorem is, L must contain a string at least that long, because L has infinitely many elements.

Theorem 5.3 would be sufficient for Example 5.7, as you are asked to show in Exercise 5.21, but it is not enough to take care of Examples 5.8 or 5.9.

Theorem 5.4 Even Weaker Form of Pumping Lemma
Suppose L is an infinite regular language. There are integers p and q, with $q > 0$, so that for every $m \geq 0$, L contains a string of length $p + mq$. In other words, the set of integers

$$lengths(L) = \{|x| \mid x \in L\}$$

contains the "arithmetic progression" of all integers $p + mq$ (where $m \geq 0$).

Proof
This follows from Theorem 5.3, by taking $p = |u| + |w|$ and $q = |v|$.

Theorem 5.4 would not be enough to show that the language in Example 5.7 is not regular. The next example shows a language for which it might be used.

EXAMPLE 5.10 An Application of Theorem 5.4

Let

$$L = \{0^n \mid n \text{ is prime}\} = \{0^2, 0^3, 0^5, 0^7, 0^{11}, \ldots\}$$

According to Theorem 5.4, in order to show that L is not regular we just need to show that the set of primes cannot contain an infinite arithmetic progression of the form $\{p + mq \mid m \geq 0\}$; in other words, for any $p \geq 0$ and any $q > 0$, there is an integer m so that $p + mq$ is not prime.

The phrase *not prime* means factorable into factors 2 or bigger. We could choose $m = p$, which would give

$$p + mq = p + pq = p(1 + q)$$

except that we are not certain that $p \geq 2$. Instead let $m = p + 2q + 2$. Then

$$\begin{aligned} p + mq &= p + (p + 2q + 2)q \\ &= (p + 2q) + (p + 2q)q \\ &= (p + 2q)(1 + q) \end{aligned}$$

and this is clearly not prime.

This example has a different flavor from the preceding ones and seems to have more to do with arithmetic, or number theory, than with languages. Yet it illustrates the fact, which will become even more obvious in the later parts of this book, that many statements about computation can be formulated as statements about languages. What we have found in this example is that a finite automaton is not a powerful enough device (it does not have enough memory) to solve the problem of determining, for an arbitrary integer, whether it is prime.

Corollary 5.1, in the first part of this chapter, gives a condition involving a language that is necessary *and* sufficient for the language to be regular. Theorem 5.2a gives a necessary condition. One might hope that it is also sufficient. This result (the converse of Theorem 5.2a) would imply that for any nonregular language L, the pumping lemma could be used to prove L nonregular; constructing the proof would just be a matter of making the right choice for x. The next example shows that this is not correct: Showing that the conclusions of Theorem 5.2a hold (i.e., showing that there is no choice of x that produces a contradiction) is not enough to show that the language is regular.

EXAMPLE 5.11 The Pumping Lemma Cannot Show a Language Is Regular

Let

$$L = \{a^i b^j c^j \mid i \geq 1 \text{ and } j \geq 0\} \cup \{b^j c^k \mid j, k \geq 0\}$$

Let us show first that the conclusions of Theorem 5.2a hold. Take n to be 1, and suppose that $x \in L$ and $|x| \geq n$. There are two cases to consider. If $x = a^i b^j c^j$, where $i > 0$, then define

$$u = \Lambda \qquad v = a \qquad w = a^{i-1} b^j c^j$$

Any string of the form $uv^m w$ is still of the form $a^l b^j c^j$ and is therefore an element of L (whether or not l is 0). If $x = b^j c^j$, then again let $u = \Lambda$ and let v be the first symbol in x. It is still true that $uv^m w \in L$ for every $m \geq 0$.

However, L is not regular, as you can show using Corollary 5.1. The details are almost identical to those in Example 5.7 and are left to Exercise 5.22.

5.4 | DECISION PROBLEMS

A finite automaton is a rudimentary computer. It receives input, and in response to that input produces the output "yes" or "no," in the sense that it does or does not end up in an accepting state. The computational problems that a finite automaton can solve are therefore limited to *decision* problems: problems that can be answered yes or no, like "Given a string x of a's and b's, does x contain an occurrence of the substring baa?" or "Given a regular expression r and a string x, does x belong to the language corresponding to r?" A decision problem of this type consists of a set of specific *instances*, or specific cases in which we want the answer. An instance of the first problem is a string x of a's and b's, and the set of possible instances is the entire set $\{a, b\}^*$. An instance of the second is a pair (r, x), where r is a regular expression and x is a string. In general, if the problem takes the form "Given x, is it true that ...?", then an instance is a particular value of x.

There are other possible formulations of a finite automaton, in which the machine operates essentially the same way but can produce more general outputs, perhaps in the form of strings over the input alphabet. What makes the finite automaton only a primitive model of computation is not that it is limited to solving decision problems, but that it can handle only *simple* decision problems. An FA cannot remember more than a fixed amount of information, and it is incapable of solving a decision problem if some instances of the problem would require the machine to remember more than this amount.

The generic decision problem that can be solved by a particular finite automaton is the *membership problem* for the corresponding regular language L: Given a string x, is x an element of L? An instance of this problem is a string x. We might step up one level and formulate the *membership problem for regular languages:* Given a finite automaton M and a string x, is x accepted by M? (Or, equivalently, given a regular language specified by the finite automaton M, and a string x, is x an element of the language?) Now an instance of the problem is a pair (M, x), where M is an FA and x is a string. The problem has an easy solution—informally, it is simply to give the string x to the FA M as input and see what happens! If M ends up in an accepting state as a result of processing x, the answer is yes; otherwise the answer is no. The reason this approach is acceptable as an algorithm is that M behaves deterministically (that is, its specifications determine exactly what steps it will follow in processing x) and is guaranteed to produce an answer after $|x|$ steps.

In addition to the membership problem, we can formulate a number of other decision problems having to do with finite automata and regular languages, and some of them we already have decision algorithms to answer. Here is a list that is not by any means exhaustive. (The first problem on the list is one of the two mentioned above.)

1. Given a regular expression r and a string x, does x belong to the language corresponding to r?
2. Given a finite automaton M, is there a string that it accepts? (Alternatively, given an FA M, is $L(M) = \emptyset$)?
3. Given an FA M, is $L(M)$ finite?

4. Given two finite automata M_1 and M_2, are there any strings that are accepted by both?
5. Given two FAs M_1 and M_2, do they accept the same language? In other words, is $L(M_1) = L(M_2)$?
6. Given two FAs M_1 and M_2, is $L(M_1)$ a subset of $L(M_2)$?
7. Given two regular expressions r_1 and r_2, do they correspond to the same language?
8. Given an FA M, is it a minimum-state FA accepting the language $L(M)$?

Problem 1 is a version of the membership problem for regular languages, except that we start with a regular expression rather than a finite automaton. Because we have an algorithm from Chapter 4 to take an arbitrary regular expression and produce an FA accepting the corresponding language, we can reduce problem 1 to the version of the membership problem previously mentioned.

Section 5.2 gives a decision algorithm for problem 8: Apply the minimization algorithm to M, and see if the number of states is reduced. Of the remaining problems, some are closely related to others. In fact, if we had an algorithm to solve problem 2, we could construct algorithms to solve problems 4 through 7. For problem 4, we could first use the algorithm presented in Section 3.5 to construct a finite automaton M recognizing $L(M_1) \cap L(M_2)$, and then apply to M the algorithm for problem 2. Problem 6 could be solved the same way, with $L(M_1) \cap L(M_2)$ replaced by $L(M_1) - L(M_2)$, because $L(M_1) \subseteq L(M_2)$ if and only if $L(M_1) - L(M_2) = \emptyset$. Problem 5 can be reduced to problem 6, since two sets are equal precisely when each is a subset of the other. Finally, a solution to problem 6 would give us one to problem 7, because of our algorithm for finding a finite automaton corresponding to a given regular expression.

Problems 2 and 3 remain. With regard to problem 2, one might ask how a finite automaton could fail to accept any strings. A trivial way is for it to have no accepting states. Even if M does have accepting states, however, it fails to accept anything if none of its accepting states is reachable from the initial state. We can determine whether this is true by calculating T_k, the set of states that can be reached from q_0 by using strings of length k or less, as follows:

$$T_k = \begin{cases} \{q_0\} & \text{if } k = 0 \\ T_{k-1} \cup \{\delta(q, a) \mid q \in T_{k-1} \text{ and } a \in \Sigma\} & \text{if } k > 0 \end{cases}$$

(T_k contains, in addition to the elements of T_{k-1}, the states that can be reached in one step from the elements of T_{k-1}.)

Decision Algorithm for Problem 2 (Given an FA M, is $L(M) = \emptyset$?) Compute the set T_k for each $k \geq 0$, until either T_k contains an accepting state or until $k > 0$ and $T_k = T_{k-1}$. In the first case $L(M) \neq \emptyset$, and in the second case $L(M) = \emptyset$. ■

If n is the number of states of M, then one of the two outcomes of the algorithm must occur by the time T_n has been computed. This implies that the following algorithm would also work.

Begin testing all input strings, in nondecreasing order of length, for acceptance by M. If no strings of length n or less are accepted, then $L(M) = \emptyset$.

Note, however, that this approach is likely to be much less efficient. For example, if we test the string 0101100 and later the string 01011000, all but the last step of the second test is duplicated effort.

The idea of testing individual strings in order to decide whether an FA accepts something is naturally tempting, but useless as an algorithm without some way to stop if the individual tests continue to fail. Only the fact that we can stop after testing strings of length n makes the approach feasible. Theorem 5.2, the original form of the pumping lemma, is another way to see that this is possible. The pumping lemma implies that if x is any string in L of length at least n, then there is a shorter string in L (the one obtained by deleting the middle portion v). Therefore, it is impossible for the shortest string in the language to have length n or greater.

Perhaps surprisingly, the pumping lemma allows us to use a similar approach with problem 3. If the FA M has n states, and x is any string in L of length at least n, then there is a string y in L that is shorter than x but not too much shorter: There exist u, v, and w with $0 < |v| \leq n$ so that $x = uvw \in L$ and $y = uw \in L$, so that the difference in length between x and y is at most n. Now consider strings in L whose length is at least n. If there are any at all, then the pumping lemma implies that L must be infinite (because there are infinitely many strings of the form $uv^i w$); in particular, if there is a string $x \in L$ with $n \leq |x| < 2n$, then L is infinite. On the other hand, if there are strings in L of length at least n, it is impossible for the shortest such string x to have length $2n$ or greater—because as we have seen, there would then have to be a shorter string $y \in L$ close enough in length to x so that $|y| \geq n$. Therefore, if L is infinite, there must be a string $x \in L$ with $n \leq |x| < 2n$. We have therefore established that the following algorithm is a solution for problem 3.

Decision Algorithm for Problem 3 (Given an FA M, is $L(M)$ finite?) Test input strings beginning with those of length n (where n is the number of states of M), in nondecreasing order of length. If there is a string x with $n \leq |x| < 2n$ that is accepted, then $L(M)$ is infinite; otherwise, $L(M)$ is finite. ■

There are at least two reasons for discussing, and trying to solve, decision problems like the ones in our list. One is the obvious fact that solutions may be useful. For a not entirely frivolous example, picture your hard-working instructor grading an exam question that asks for an FA recognizing a specific language. He or she knows a solution, but one student's paper shows a different FA. The instructor must then try to determine whether the two are equivalent, and this means answering an instance of problem 5. If the answer to a specific instance of the problem is the primary concern, then whether there is an efficient, or feasible, solution is at least as important as whether there is a solution in principle. The solution sketched above for problem 5 involves solving problem 2, and the second version of the decision algorithm given for problem 2 would not help much in the case of machines with a hundred states, even if a computer program and a fast computer were available.

Aside from the question of finding efficient algorithms, however, there is another reason for considering these decision problems, a reason that will assume greater significance later in the book. It is simply that not all decision problems can be solved.

An example of an easy-to-state problem that cannot be solved by any decision algorithm was formulated in the 1930s by the mathematician Alan Turing. He described a type of abstract machine, now called a Turing machine, more general than a finite automaton. These machines can recognize certain languages in the same way that FAs can recognize regular languages, and Turing's original unsolvable problem is simply the membership problem for this more general class of languages: Given a Turing machine M and a string x, does M accept x? (see Section 11.2). Turing machines are involved in this discussion in another way as well, because such a machine turns out to be a general model of computation. This is what allows us to formulate the idea of an algorithm precisely and to say exactly what "unsolvable" means.

Showing the existence of unsolvable problems—particularly ones that arise naturally and are easy to state—was a significant development in the theory of computation. The conclusion is that there are definite theoretical limits on what it is possible to compute. These limits have nothing to do with how smart we are, or how good at designing software; and they are not simply practical limits having to do with efficiency and the amount of time available, or physical considerations like the number of atoms available for constructing memory devices. Rather, they are fundamental limits inherent in the rules of logic and the nature of computation. We will be investigating these matters later in the book; for the moment, it is reassuring to find that many of the natural problems involving regular languages do have algorithmic solutions.

5.5 REGULAR LANGUAGES AND COMPUTERS

Now that we have introduced the first class of languages we will be studying, we can ask what the relationship is between these simple languages and the familiar ones people use in computer science, programming languages such as C, Java, and Pascal. The answer is almost obvious: Programming languages are not regular. In the C language, for example, the string `main() {`^{m}`}`^{n} is a valid program if and only if $m = n$, and this allows us to prove easily that the set of valid programs is not regular, using either Corollary 5.1 or the pumping lemma.

Although regular languages are not rich enough in structure to be programming languages themselves, however, we have seen in Examples 3.5 and 3.6 some of the ways they occur within programming languages. As a general rule, the *tokens* of a programming language, which include identifiers, literals, operators, reserved words, and punctuation, can be described by a regular expression. The first phase in compiling a program written in a high-level programming language is *lexical analysis*: identifying and classifying the tokens into which the individual characters are grouped. There are programs called lexical-analyzer generators. The input provided

to such a program is a set of regular expressions specifying the structure of tokens, and the output produced by the program is a software version of an FA that can be incorporated as a token-recognizing module in a compiler. One of the most widely used of these is a program called `lex`, which is a tool provided in the Unix operating system. Although `lex` can be used in many situations that require the processing of structured input, it is used most often in conjunction with `yacc`, another Unix tool. The lexical analyzer produced by `lex` creates a string of tokens; and the *parser* produced by `yacc`, on the basis of grammar rules provided as input, is able to determine the syntactical structure of the token string. (`yacc` stands for *yet another compiler compiler*.) Regular expressions come up in Unix in other ways as well. The Unix text editor allows the user to specify a regular expression and searches for patterns in the text that match it. Other commands such as `grep` (global regular expression print) and `egrep` (extended global regular expression print) cause a specified file to be searched for lines containing strings that match a specified regular expression.

If regular languages cannot be programming languages, it would seem that finite automata are even less equipped to be computers. There are a number of obvious differences, some more significant than others, having to do with memory, output capabilities, programmability, and so on. We have seen several examples of languages, such as $\{0^n 1^n \mid n \geq 0\}$, that no FA can recognize but for which a recognition program could be written by any programmer and run on just about any computer.

Well, yes and no; as obvious as this conclusion seems, it has to be qualified at least a little. Any physical computer is a finite device; it has, for some integer n, n total bits of internal memory and disk space. (It may be connected to a larger network, but in that case we may think of the "computer" as the entire network and simply use a larger value of n.) We can describe the complete *state* of the machine by specifying the status of each bit of memory, each pixel on the screen, and so forth. The number of states is huge but still finite, and in this sense our computer is in fact an FA, where the inputs can be thought of as keystrokes, or perhaps bits in some external file being read. (In particular, the computer cannot *actually* recognize the language $\{0^j 1^j\}$, because there is an integer j so large that if the computer has read exactly j 0's, it will not be able to remember this.)

As a way of understanding a computer, however, this observation is not helpful; there is hardly any practical difference between this finite number of possible states and infinity. Finite automata are simple machines, but having to think about a computer as an FA would complicate working with computers considerably. The situation is similar with regard to languages. One might argue that programming languages are effectively regular because the set of programs that one can physically write, or enter as input to a computer, is finite. However, though finite languages are simpler in some ways than infinite languages (in particular, they are always regular), restricting ourselves to finite languages would by no means simplify the discussion. Finite languages can be called *simple* because there is no need to consider any underlying structure—they are just sets of strings. However, with no underlying principle to impose some logical organization (some complexity!), a large set becomes unwieldy and complicated to deal with.

The advantage of a theoretical approach to computation is that we do not need to get bogged down in issues like memory size. Obviously there are many languages, including some regular ones, that no physical computer will ever be able to recognize; however, it still makes sense to distinguish between the logical problems that arise in recognizing some of these and the problems that arise in recognizing others. Finite automata and computers are different in principle, and what we are studying is what each is capable of in principle, not what a specific computer can do in practice. (Most people would probably agree that in principle, a computer *can* recognize the language $\{0^n 1^n \mid n \geq 0\}$.) As we progress further in the book, we will introduce abstract models that resemble a computer more closely. There will never be a perfect physical realization of any of them. Studying the conceptual model, however, is still the best way to understand both the potential and the limitations of the physical machines that approximate the model.

EXERCISES

5.1. For which languages $L \subseteq \{0, 1\}^*$ is there only one equivalence class with respect to the relation I_L?

5.2. Let x be an arbitrary string in $\{0, 1\}^*$, and let $L = \{x\}$. How many equivalence classes are there for the relation I_L? Describe them.

5.3. Find a language $L \subseteq \{0, 1\}^*$ for which every equivalence class of I_L has exactly one element.

5.4. Show that for any language $L \subseteq \Sigma^*$, the set

$$S = \{x \in \Sigma^* \mid x \text{ is not a prefix of any element of } L\}$$

is one equivalence class of I_L, provided it is not empty.

5.5. Let $L \subseteq \Sigma^*$ be any language. Show that if $[\Lambda]$ (the equivalence class of I_L containing Λ) is not $\{\Lambda\}$, then it is infinite.

5.6. Show that if $L \subseteq \Sigma^*$ is a language, $x \in \Sigma^*$, and $[x]$ (the equivalence class of I_L containing x) is finite, then x is a prefix of an element of L.

5.7. For a certain language $L \subseteq \{a, b\}^*$, I_L has exactly four equivalence classes. They are $[\Lambda]$, $[a]$, $[ab]$, and $[b]$. It is also true that the three strings a, aa, and abb are all equivalent, and that the two strings b and aba are equivalent. Finally, $ab \in L$, but Λ and a are not in L, and b is not even a prefix of any element of L. Draw an FA accepting L.

5.8. Suppose there is a 3-state FA accepting $L \subseteq \{a, b\}^*$. Suppose $\Lambda \notin L$, $b \notin L$, and $ba \in L$. Suppose also that $a I_L b$, $\Lambda I_L bab$, $a I_L aaa$, and $b I_L bb$. Draw an FA accepting L.

5.9. Suppose there is a 3-state FA accepting $L \subseteq \{a, b\}^*$. Suppose $\Lambda \notin L$, $b \in L$, $ba \notin L$, and $baba \in L$, and that $\Lambda I_L a$ and $a I_L bb$. Draw an FA accepting L.

5.10. Find all possible languages $L \subseteq \{a, b\}^*$ for which I_L has these three equivalence classes: the set of all strings ending in b, the set of all strings ending in ba, and the set of all strings ending in neither b nor ba.

5.11. Find all possible languages $L \subseteq \{a, b\}^*$ for which I_L has three equivalence classes, corresponding to the regular expressions $((a + b)a^*b)^*$, $((a + b)a^*b)^*aa^*$, and $((a + b)a^*b)^*ba^*$, respectively.

5.12. In Example 5.2, if the language is changed to $\{0^n1^n \mid n \geq 0\}$ (i.e., Λ is added to the original language), are there any changes in the partition of $\{0, 1\}^*$ corresponding to I_L? Explain.

5.13. Consider the language $L = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$ (where $n_0(x)$ and $n_1(x)$ are the number of 0's and the number of 1's, respectively, in x).

a. Show that if $n_0(x) - n_1(x) = n_0(y) - n_1(y)$, then $x \; I_L \; y$.

b. Show that if $n_0(x) - n_1(x) \neq n_0(y) - n_1(y)$, then x and y are distinguishable with respect to x.

c. Describe all the equivalence classes of I_L.

5.14. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA, and suppose that Q_1 is a subset of Q such that $\delta(q, a) \in Q_1$ for every $q \in Q_1$ and every $a \in \Sigma$.

a. Show that if $Q_1 \cap A = \emptyset$, then for any p and q in Q_1, $p \equiv q$.

b. Show that if $Q_1 \subseteq A$, then for any p and q in Q_1, $p \equiv q$.

5.15. For a language L over Σ, and two strings x and y in Σ^* that are distinguishable with respect to L, let

$$d_{L,x,y} = \min\{|z| \mid z \text{ distinguishes } x \text{ and } y \text{ with respect to } L\}$$

a. For the language $L = \{x \in \{0, 1\}^* \mid x \text{ ends in } 010\}$, find the maximum of the numbers $d_{L,x,y}$ over all possible pairs of distinguishable strings x and y.

b. If L is the language of balanced strings of parentheses, if $|x| = m$ and $|y| = n$, find an upper bound involving m and n on the numbers $d_{L,x,y}$.

5.16. For each of the FAs pictured in Figure 5.6, use the minimization algorithm described in Algorithm 5.1 and illustrated in Example 5.6 to find a minimum-state FA recognizing the same language. (It's possible that the given FA may already be minimal.)

5.17. Find a minimum-state FA recognizing the language corresponding to each of these regular expressions.

a. $(0^*10 + 1^*0)(01)^*$

b. $(010)^*1 + (1^*0)^*$

5.18. Suppose that in applying Algorithm 5.1, we establish some fixed order in which to process the pairs, and we follow the same order on each pass.

a. What is the maximum number of passes that might be required? Describe an FA, and an ordering of the pairs, that would require this number.

b. Is there always a fixed order (depending on M) that would guarantee no pairs are marked after the second pass, so that the algorithm terminates after three passes?

5.19. For each of the NFA-Λs pictured in Figure 5.7, find a minimum-state FA accepting the same language.

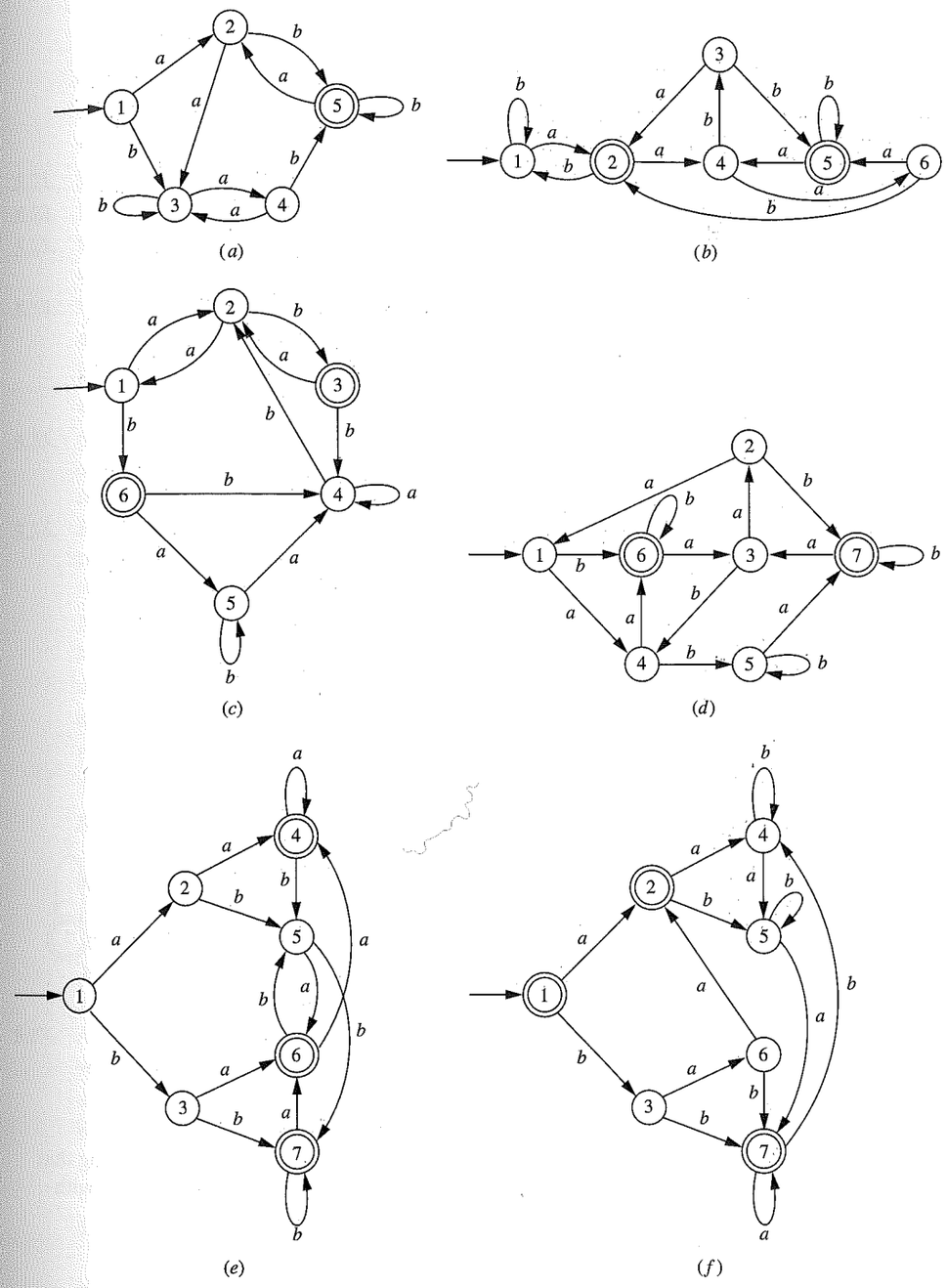
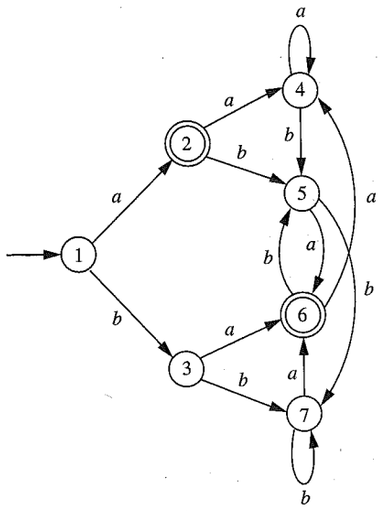
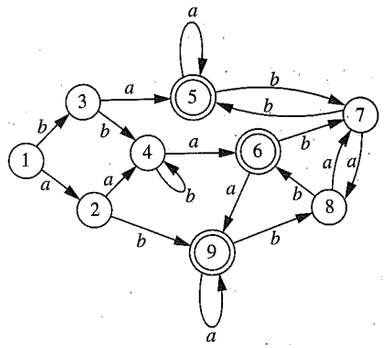


Figure 5.6 |

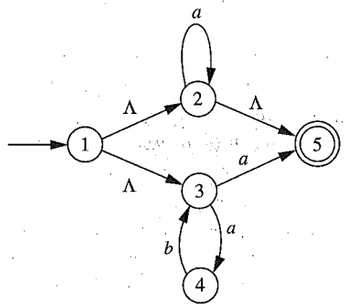


(g)

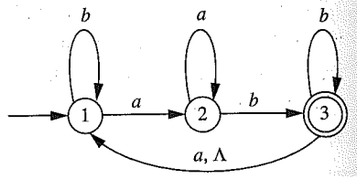


(h)

Figure 5.6 |
Continued



(a)



(b)

Figure 5.7 |

5.20. In each of the following cases, prove that L is nonregular by showing that any two elements of the infinite set $\{0^n \mid n \geq 0\}$ are distinguishable with respect to L.

a. $L = \{0^n 10^{2n} \mid n \geq 0\}$

b. $L = \{0^i 1^j 0^k \mid k > i + j\}$

c. $L = \{0^i 1^j \mid j = i \text{ or } j = 2i\}$

d. $L = \{0^i 1^j \mid j \text{ is a multiple of } i\}$

e. $L = \{x \in \{0, 1\}^* \mid n_0(x) < 2n_1(x)\}$

f. $L = \{x \in \{0, 1\}^* \mid \text{no prefix of } x \text{ has more 1's than 0's}\}$

5.21. Use Theorem 5.3 to show that $\{0^n 1^n \mid n \geq 0\}$ is not regular.

5.22. Use Corollary 5.1 to show that the language in Example 5.11 is not regular.

5.23. In each part of Exercise 5.20, use the pumping lemma for regular languages to show that the language is not regular.

5.24. Use the pumping lemma to show that each of these languages is not regular:

a. $L = \{ww \mid w \in \{0, 1\}^*\}$

b. $L = \{xy \mid x, y \in \{0, 1\}^* \text{ and } y \text{ is either } x \text{ or } x^r\}$

c. The language of algebraic expressions in Example 5.3.

5.25. Suppose L is a language over $\{0, 1\}$, and there is a fixed integer k so that for every $x \in \Sigma^*$, $xz \in L$ for some string z with $|z| \leq k$. Does it follow that L is regular? Why or why not?

5.26. For each statement below, decide whether it is true or false. If it is true, prove it. If not, give a counterexample. All parts refer to languages over the alphabet $\{0, 1\}$.

a. If $L_1 \subseteq L_2$ and L_1 is not regular, then L_2 is not regular.

b. If $L_1 \subseteq L_2$ and L_2 is not regular, then L_1 is not regular.

c. If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ is nonregular.

d. If L_1 and L_2 are nonregular, then $L_1 \cap L_2$ is nonregular.

e. If L is nonregular, then L' is nonregular.

f. If L_1 is regular and L_2 is nonregular, then $L_1 \cup L_2$ is nonregular.

g. If L_1 is regular, L_2 is nonregular, and $L_1 \cap L_2$ is regular, then $L_1 \cup L_2$ is nonregular.

h. If L_1 is regular, L_2 is nonregular, and $L_1 \cap L_2$ is nonregular, then $L_1 \cup L_2$ is nonregular.

i. If $L_1, L_2, L_3, \ldots$ are all regular, then $\cup_{n=1}^{\infty} L_n$ is regular.

j. If $L_1, L_2, L_3, \ldots$ are all nonregular and $L_i \subseteq L_{i+1}$ for each i, then $\cup_{n=1}^{\infty} L_n$ is nonregular.

5.27. A number of languages over $\{0, 1\}$ are given in (a)–(h). In each case, decide whether the language is regular or not, and prove that your answer is correct.

a. The set of all strings x beginning with a nonnull string of the form ww.

b. The set of all strings x containing some nonnull substring of the form ww.

c. The set of odd-length strings over $\{0, 1\}$ with middle symbol 0.

d. The set of even-length strings over $\{0, 1\}$ with the two middle symbols equal.

e. The set of strings over $\{0, 1\}$ of the form xyx for some x with $|x| \geq 1$.

f. The set of nonpalindromes.

g. The set of strings beginning with a palindrome of length at least 3.

h. The set of strings in which the number of 0's is a perfect square.

5.28. Describe decision algorithms to answer each of these questions.

a. Given two FAs M_1 and M_2, are there any strings that are accepted by neither?

b. Given a regular expression r and an FA M, are the corresponding languages the same?

c. Given an FA $M = (Q, \Sigma, q_0, A, \delta)$ and a state $q \in Q$, is there an x with $|x| > 0$ so that $\delta^*(q, x) = q$?

d. Given an NFA-Λ M and a string x, does M accept x?

e. Given two NFA-Λs, do they accept the same language?

f. Given an NFA-Λ M and a string x, is there more than one sequence of transitions corresponding to x that causes M to accept x?

g. Given an FA M accepting a language L, and given two strings x and y, are x and y distinguishable with respect to L?

h. Given an FA M accepting a language L, and a string x, is x a prefix of an element of L?

i. Given an FA M accepting a language L, and a string x, is x a suffix of an element of L?

j. Given an FA M accepting a language L, and a string x, is x a substring of an element of L?

5.29. Find an example of a language $L \subseteq \{0, 1\}^*$ so that L^* is not regular.

5.30. Find an example of a nonregular language $L \subseteq \{0, 1\}^*$ so that L^* is regular.

MORE CHALLENGING PROBLEMS

5.31. Let $L \subseteq \Sigma^*$ be a language, and let L_1 be the set of prefixes of elements of L. What is the relationship, if any, between the two partitions of Σ^* corresponding to the equivalence relations I_L and I_{L_1}, respectively? Explain.

5.32. a. List all the subsets A of $\{0, 1\}^*$ having the property that for some language $L \subseteq \{0, 1\}^*$ for which I_L has exactly two equivalence classes, $A = [\Lambda]$.

b. For each set A that is one of your answers to (a), how many distinct languages L are there so that I_L has two equivalence classes and $[\Lambda]$ is A?

5.33. Let $L = \{ww \mid w \in \{0, 1\}^*\}$. Describe all the equivalence classes of I_L.

5.34. Let L be the language of "balanced" strings of parentheses—that is, all strings that are the strings of parentheses in legal algebraic expressions. For example, Λ, ()(), and ((()())) are in L, (() and))(() are not. Describe all the equivalence classes of I_L.

5.35. a. Let L be the language of all fully parenthesized algebraic expressions involving the operator $+$ and the identifier i. (L can be defined recursively by saying $i \in L$, $(x + y) \in L$ for every x and y in L, and nothing else is in L.) Describe all the equivalence classes of I_L.

b. Answer the same question for the language L in Example 5.3, defined by saying $a \in L$, $x + y \in L$ for every x and y in L, and $(x) \in L$ for every $x \in L$.

5.36. For an arbitrary string $x \in \{0, 1\}^*$, denote by $x^\sim$ the string obtained by replacing all 0's by 1's and vice versa. For example, $\Lambda^\sim = \Lambda$ and $(011)^\sim = 100$.

a. Define

$$L = \{xx^\sim \mid x \in \{0, 1\}^*\}$$

Determine the equivalence classes of I_L.

b. Define

$$L_1 = \{xy \mid x \in \{0, 1\}^* \text{ and } y \text{ is either } x \text{ or } x^\sim\}$$

Determine the equivalence classes of I_{L_1}.

5.37. Let $L = \{x \in \{0, 1\}^* \mid n_1(x) \text{ is a multiple of } n_0(x)\}$. Determine the equivalence classes of I_L.

5.38. Let L be a language over Σ. We know that I_L is a *right invariant* equivalence relation (i.e., for any x and y in Σ^* and any $a \in \Sigma$, if $x \; I_L \; y$, then $xa \; I_L \; ya$). By the Myhill-Nerode theorem (Corollary 5.1), we know that if the set of equivalence classes of I_L is finite, then L is regular, and in this case L is the union of some (zero or more) of these finitely many equivalence classes. Show that if R is *any* right invariant equivalence relation such that the set of equivalence classes of R is finite and L is the union of some of the equivalence classes of R, then L is regular.

5.39. If P is a partition of $\{0, 1\}^*$ (i.e., a collection of pairwise disjoint subsets whose union is $\{0, 1\}^*$), then there is an equivalence relation R on $\{0, 1\}^*$ whose equivalence classes are precisely the subsets in P. Let us say that P is right invariant if the resulting equivalence relation is.

a. Show that for a subset S of $\{0, 1\}^*$, S is one of the subsets of some right invariant partition of $\{0, 1\}^*$ (not necessarily a finite partition) if and only if the following condition is satisfied: for any $x, y \in S$, and any $z \in \{0, 1\}^*$, xz and yz are either both in S or both not in S.

b. To what simpler condition does this one reduce in the case where S is a finite set?

c. Show that if a finite set S satisfies this condition, then there is a finite right invariant partition having S as one of its subsets.

d. For an arbitrary set S satisfying the condition in part (a), there may be no finite right invariant partition having S as one of its subsets. Characterize those sets S for which there is.

5.40. For two languages L_1 and L_2 over Σ, we define the *quotient* of L_1 and L_2 to be the language

$$L_1/L_2 = \{x \mid \text{ for some } y \in L_2, xy \in L_1\}$$

Show that if L_1 is regular and L_2 is any language, then L_1/L_2 is regular.

5.41. Suppose L is a language over Σ, and $x_1, x_2, \ldots, x_n$ are strings that are pairwise distinguishable with respect to L; that is, for any $i \neq j$, x_i and x_j are distinguishable. How many distinct strings are necessary in order to distinguish between the x_i's? In other words, what is the smallest number k so that for some set $\{z_1, z_2, \ldots, z_k\}$, any two distinct x_i's are distinguished, relative to L, by some z_l? Prove your answer. (Here is a way of thinking about the question that may make it easier. Think of the x_i's as points on a piece of paper, and think of the z_l's as cans of paint, each z_l representing a different primary color. Saying that z_l distinguishes x_i and x_j means that one of those two points is colored with that primary color and the other isn't. We allow a single point to have more than one primary color applied to it, and we assume that two distinct combinations of primary colors produce different resulting colors. Then the question is, how many different primary colors are needed in order to color the points so that no two points end up the same color?)

5.42. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting L. We know (Lemma 5.2) that if $p, q \in Q$ and $p \not\equiv q$, then there is a string z so that exactly one of the two states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A. Find an integer n (depending only on M) so that for any p and q with $p \not\equiv q$, there is such a z with $|z| \leq n$.

5.43. Show that L is regular if and only if there is an integer n so that any two strings distinguishable with respect to L can be distinguished by a string of length $\leq n$. (Use the two previous exercises.)

5.44. Suppose that $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are both FAs accepting the language L, and both have as few states as possible. Show that M_1 and M_2 are isomorphic (see Exercise 3.55). Note that in both cases, the sets L_q forming the partition of Σ^* are precisely the equivalence classes of I_L. This tells you how to come up with a bijection from Q_1 to Q_2. What you must do next is to show that the other conditions of an isomorphism are satisfied.

5.45. Use the preceding exercise to describe another decision algorithm to answer the question "Given two FAs, do they accept the same language?"

5.46. Suppose L and L_1 are both languages over Σ, and M is an FA with alphabet Σ. Let us say that M accepts L *relative to* L_1 if M accepts every string in the set $L \cap L_1$ and rejects every string in the set $L_1 - L$. Note that this is not in general the same as saying that M accepts the language $L \cap L_1$.

Now suppose $L_1, L_2, \ldots$ are regular languages over Σ, $L_i \subseteq L_{i+1}$ for each i, and $\cup_{i=1}^{\infty} L_i = \Sigma^*$. For each i, let n_i be the minimum number of states required to accept L relative to L_i. If there is no FA accepting L relative to L_i, we say n_i is ∞.

a. Show that for each i, $n_i \leq n_{i+1}$.

b. Show that if the sequence n_i is bounded (i.e., for some constant C, $n_i \leq C$ for every i), then L is regular. (It follows in particular that if there is some fixed FA that accepts L relative to L_i for every i, then L is regular.)

5.47. Prove the following generalization of the pumping lemma, which can sometimes make it unnecessary to break the proof into cases. If L is a regular language, then there is an integer n so that for any $x \in L$, and any way of writing x as $x = x_1x_2x_3$ with $|x_2| = n$, there are strings u, v, and w so that

$$x_2 = uvw$$

$$|v| > 0$$

$$\text{for any } m \geq 0,\ x_1uv^mwx_2 \in L$$

5.48. Can you find a language $L \subseteq \{0, 1\}^*$ so that in order to prove L nonregular, the pumping lemma is not sufficient but the statement in the preceding problem is?

5.49. Describe decision algorithms to answer each of these questions.

a. Given a regular expression r, is there a simpler regular expression (i.e., one involving fewer operations) that is equivalent to r?

b. Given two FAs M_1 and M_2, is $L(M_1)$ a subset of $L(M_2)$?

c. Given two FAs M_1 and M_2, is every element of $L(M_1)$ a prefix of an element of $L(M_2)$?

d. Given two FAs $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and M_2, and two states $p, q \in Q_1$, is there a string $x \in L(M_2)$ so that $\delta_1^*(p, x) = q$?

5.50. Below are a number of languages over $\{0, 1\}$. In each case, decide whether the language is regular or not, and prove that your answer is correct.

a. The set of all strings x having some nonnull substring of the form www. (You may assume the following fact: There are arbitrarily long strings in $\{0, 1\}^*$ that do not contain any nonnull substring of the form www. In fact, such strings can be obtained using the construction in Exercise 2.43.)

b. The set of strings having the property that in every prefix, the number of 0's and the number of 1's differ by no more than 2.

c. The set of strings having the property that in some prefix, the number of 0's is 3 more than the number of 1's.

d. The set of strings in which the number of 0's and the number of 1's are both divisible by 5.

e. The set of strings x for which there is an integer $k > 1$ (possibly depending on x) so that the number of 0's in x and the number of 1's in x are both divisible by k.

f. (Assuming L is a regular language), $Max(L) = \{x \in L \mid$ there is no nonnull string y so that $xy \in L\}$.

g. (Assuming L is a regular language), $Min(L) = \{x \in L \mid$ no prefix of x other than x itself is in $L\}$.

5.51. A set S of nonnegative integers is an *arithmetic progression* if for some integers n and p,

$$S = \{n + ip \mid i \geq 0\}$$

Let A be a subset of $\{0\}^*$, and let $S = \{|x| \mid x \in A\}$.

a. Show that if S is an arithmetic progression, then A is regular.

b. Show that if A is regular, then S is the union of a finite number of arithmetic progressions.

5.52. This exercise involves languages of the form

$$L = \{x \in \{a, b\}^* \mid n_a(x) = f(n_b(x))\}$$

for some function f from the set of natural numbers to itself. Example 5.7 shows that if f is the function defined by $f(n) = n$, then L is nonregular. If f is any constant function (e.g., $f(n) = 4$), L is regular. One might ask whether L can still be regular when f is not restricted quite so severely.

a. Show that if L is regular, the function f must be bounded—that is, there must be some integer B so that $f(n) \leq B$ for every n. (Suggestion: suppose not, and apply the pumping lemma to strings of the form $a^{f(n)}b^n$.)

b. Show that if $f(n) = n \bmod 2$, then L is regular.

c. $n \bmod 2$ is an *eventually periodic* function; that is, there are integers n_0 and p, with $p > 0$, so that for any $n \geq n_0$, $f(n) = f(n + p)$. Show that if f is any eventually periodic function, L is regular.

d. Show that if L is regular, then f must be eventually periodic. (Suggestion: as in part (a), find a class of strings to which you can apply the pumping lemma.)

5.53. Find an example of a nonregular language $L \subseteq \{0, 1\}^*$ so that L^2 is regular.

5.54. Show that if L is any language over a one-symbol alphabet, then L^* is regular.

PART

3

Context-Free Languages and Pushdown Automata

A *context-free grammar* is a simple recursive method of specifying grammar rules by which strings in a language can be generated. All the regular languages can be generated this way, and there are also simple examples of context-free grammars generating nonregular languages. Grammar rules of this type permit syntax of more variety and sophistication than is possible with regular languages. To a large extent, they are capable of specifying the syntax of high-level programming languages and other formal languages.

A model of computation that corresponds to context-free languages, in the same way that finite automata correspond to regular languages, can be obtained by starting with the finite-state model and adding an auxiliary memory. Although the memory will be potentially infinite, it is sufficient to impose upon it a very simple organization, that of a *stack*. It is necessary, however, to retain the element of nondeterminism in these *pushdown automata*; otherwise, not every context-free language can be accepted this way. For any context-free grammar, there is a simple way to get a nondeterministic pushdown automaton accepting the language so that a sequence of moves by which a string is accepted simulates a derivation of the string in the grammar. For certain classes of grammars, this feature can be retained even when the nondeterminism is removed, so that the result is a parser for the grammar; we study this problem briefly.

The class of context-free languages is still not general enough to include all interesting or useful formal languages. Techniques similar to those in Chapter 5 can be used to exhibit simple non-context-free languages, and these techniques can also be used to find algorithms for certain decision problems associated with context-free languages. ■