CHAPTER

# 3

# Regular Languages and Finite Automata

## 3.1 REGULAR LANGUAGES AND REGULAR EXPRESSIONS

Nonnull strings over an alphabet $\Sigma$ are created by concatenating simple strings, those of length 1. Since concatenation can also be thought of as an operation on languages, we may consider the languages obtained by concatenation from the simple languages of the form $\{a\}$, where $a \in \Sigma$. If concatenation is the only operation we allow, however, we can get only single strings or languages that contain single strings. Adding the set operation of union permits languages with several elements, and if we allow the Kleene * operation, which arises naturally from concatenation, we can produce infinite languages as well.

To the simple languages of the form $\{a\}$, we add two more: the empty language $\emptyset$ and the language $\{\Lambda\}$ whose only element is the null string.

A *regular* language over an alphabet $\Sigma$ is one that can be obtained from these basic languages using the operations of union, concatenation, and Kleene *. A regular language can therefore be described by an explicit formula. It is common to simplify the formula slightly, by leaving out the set brackets {} or replacing them with parentheses and by replacing $\cup$ by $+$; the result is called a *regular expression.*

Here are several examples of regular languages over the alphabet {0, 1}, along with the corresponding regular expressions.

| *Language* | *Corresponding Regular Expression* |
|---|---|
| **1.** $\{\Lambda\}$ | $\Lambda$ |
| **2.** $\{0\}$ | 0 |
| **3.** $\{001\}$ (i.e., $\{0\}\{0\}\{1\}$) | 001 |
| **4.** $\{0, 1\}$ (i.e., $\{0\} \cup \{1\}$) | $0 + 1$ |
| **5.** $\{0, 10\}$ (i.e., $\{0\} \cup \{10\}$) | $0 + 10$ |

| *Language* | *Corresponding Regular Expression* |
|---|---|
| **6.** $\{1, \Lambda\}\{001\}$ | $(1 + \Lambda)001$ |
| **7.** $\{110\}^*\{0, 1\}$ | $(110)^*(0 + 1)$ |
| **8.** $\{1\}^*\{10\}$ | $1^*10$ |
| **9.** $\{10, 111, 11010\}^*$ | $(10 + 111 + 11010)^*$ |
| **10.** $\{0, 10\}^*(\{11\}^* \cup \{001, \Lambda\})$ | $(0 + 10)^*((11)^* + 001 + \Lambda)$ |

We think of a regular expression as representing the "most typical string" in the corresponding language. For example, $1^*10$ stands for a string that consists of the substring 10 preceded by any number of 1's.

The phrase we used above, "obtained from these basic languages using the operations of union, concatenation, and Kleene *," should suggest a recursive definition of the type we studied in Section 2.4.2. It will be helpful to complicate the definition a little so that it defines not only the regular languages but also the regular expressions corresponding to them.

**Definition 3.1 Regular Languages and Regular Expressions over $\Sigma$**

The set $R$ of regular languages over $\Sigma$, and the corresponding regular expressions, are defined as follows.

1. $\emptyset$ is an element of $R$, and the corresponding regular expression is $\emptyset$.
2. $\{\Lambda\}$ is an element of $R$, and the corresponding regular expression is $\Lambda$.
3. For each $a \in \Sigma$, $\{a\}$ is an element of $R$, and the corresponding regular expression is $a$.
4. If $L_1$ and $L_2$ are any elements of $R$, and $r_1$ and $r_2$ are the corresponding regular expressions,
   (a) $L_1 \cup L_2$ is an element of $R$, and the corresponding regular expression is $(r_1 + r_2)$;
   (b) $L_1L_2$ is an element of $R$, and the corresponding regular expression is $(r_1r_2)$;
   (c) $L_1^*$ is an element of $R$, and the corresponding regular expression is $(r_1^*)$.

Only those languages that can be obtained by using statements 1–4 are regular languages over $\Sigma$.

The empty language is included in the definition primarily for the sake of consistency. There will be a number of places where we will want to say things like "To every something-or-other, there corresponds a regular language," and without the language $\emptyset$ we would need to make exceptions for trivial special cases.

Our definition of regular expressions is really a little more restrictive in several respects than we need to be in practice. We use notation such as $L^2$ for languages, and it is reasonable to use similar shortcuts in the case of regular expressions. Thus we sometimes write $(r^2)$ to stand for the regular expression $(rr)$, $(r^+)$ to stand for the regular expression $((r^*)r)$, and so forth. You should also note that the regular



expressions we get from the definition are *fully parenthesized*. We will usually relax this requirement, using the same rules that apply to algebraic expressions: The Kleene * operation has the highest precedence and + the lowest, with concatenation in between. This rule allows us to write $a + b^*c$, for example, instead of $(a + ((b^*)c))$. Just as with algebraic expressions, however, there are times when parentheses are necessary. The regular expression $(a + b)^*$ is a simple example, since the languages corresponding to $(a + b)^*$ and $a + b^*$ are not the same.

Let us agree to identify two regular expressions if they correspond to the same language. At the end of the last paragraph, for example, we could simply have said

$$(a + b)^* \neq a + b^*$$

instead of saying that the two expressions correspond to different languages. With this convention we can look at a few examples of rules for simplifying regular expressions over $\{0, 1\}$:

$$1^*(1 + \Lambda) = 1^*$$
$$1^*1^* = 1^*$$
$$0^* + 1^* = 1^* + 0^*$$
$$(0^*1^*)^* = (0 + 1)^*$$
$$(0 + 1)^*01(0 + 1)^* + 1^*0^* = (0 + 1)^*$$

(All five are actually special cases of more general rules. For example, for any two regular expressions $r$ and $s$, $(r^*s^*)^* = (r + s)^*$.) These rules are really statements about languages, which we could have considered in Chapter 1. The last one is probably the least obvious. It says that the language of all strings of 0's and 1's (the right side) can be expressed as the union of two languages, one containing all the strings having the substring 01, the other containing all the others. (Saying that all the 1's precede all the 0's is the same as saying that 01 is not a substring.)

Although there are times when it is helpful to simplify a regular expression as much as possible, we will not attempt a systematic discussion of the algebra of regular expressions. Instead, we consider a few more examples.

### Strings of Even Length — EXAMPLE 3.1

Let $L \subseteq \{0, 1\}^*$ be the language of all strings of even length. (Since 0 is even, $\Lambda \in L$.) Is $L$ regular? If it is, what is a regular expression corresponding to it?

Any string of even length can be obtained by concatenating zero or more strings of length 2. Conversely, any such concatenation has even length. It follows that

$$L = \{00, 01, 10, 11\}^*$$

so that one regular expression corresponding to $L$ is $(00 + 01 + 10 + 11)^*$. Another is $((0 + 1)(0 + 1))^*$.

## EXAMPLE 3.2 Strings with an Odd Number of 1's

Let $L$ be the language of all strings of 0's and 1's containing an odd number of 1's. Any string in $L$ must contain at least one 1, and it must therefore start with a string of the form $0^i10^j$. There is an even number (possibly zero) of additional 1's, each followed by zero or more 0's. This means that the rest of the string is the concatenation of zero or more pieces of the general form $10^m10^n$. One regular expression describing $L$ is therefore

$$0^*10^*(10^*10^*)^*$$

A slightly different expression, which we might have obtained by stopping the initial substring immediately after the 1, is

$$0^*1(0^*10^*1)^*0^*$$

If we had begun by considering the *last* 1 in the string, rather than the first, we might have ended up with

$$(0^*10^*1)^*0^*10^*$$

A more complicated answer that is still correct is

$$0^*(10^*10^*)^*1(0^*10^*1)^*0^*$$

In this case the 1 that is emphasized is somewhere in the middle, with an even number of 1's on either side of it. There are still other ways we could describe a typical element of $L$, depending on which aspect of the structure we wanted to emphasize, and there is not necessarily one that is the simplest or the most natural. The important thing in all these examples is that the regular expression must be general enough to describe every string in the language. One that does not quite work, for example, is

$$(10^*10^*)^*10^*$$

since it does not allow for strings beginning with 0. We could correct this problem by inserting $0^*$ at the beginning, to obtain

$$0^*(10^*10^*)^*10^*$$

This is a way of showing the last 1 in the string explicitly, slightly different from the third regular expression in this example.

## EXAMPLE 3.3 Strings of Length 6 or Less

Let $L$ be the set of all strings over $\{0, 1\}$ of length 6 or less. A simple but inelegant regular expression corresponding to $L$ is

$$\Lambda + 0 + 1 + 00 + 01 + 10 + 11 + 000 + \cdots + 111110 + 111111$$

A regular expression to describe the set of strings of length exactly 6 is

$$(0 + 1)(0 + 1)(0 + 1)(0 + 1)(0 + 1)(0 + 1)$$

or, in our extended notation, $(0 + 1)^6$. To reduce the length, however, we may simply allow some or all of the factors to be $\Lambda$. We may therefore describe $L$ by the regular expression

$$(0 + 1 + \Lambda)^6$$



## EXAMPLE 3.4 Strings Ending in 1 and Not Containing 00

This time we let $L$ be the language

$$L = \{x \in \{0, 1\}^* \mid x \text{ ends with 1 and does not contain the substring 00}\}$$

In order to find a regular expression for $L$, we try stating the defining property of strings in $L$ in other ways. Saying that a string does not contain the substring 00 is the same as saying that no 0 can be followed by 0, or in other words, every 0 either comes at the very end or is followed immediately by 1. Since strings in $L$ cannot have 0 at the end, every 0 must be followed by 1. This means that copies of the strings 01 and 1 can account for the entire string and therefore that every string in $L$ corresponds to the regular expression $(1 + 01)^*$. This regular expression is a little too general, however, since it allows the null string. The definition says that strings in $L$ must end with 1, and this is stronger than saying they cannot end with 0. We cannot fix the problem by adding a 1 at the end, to obtain $(1+01)^*1$ because now our expression is not general enough; it does not allow 01. Allowing this choice at the end, we obtain $(1 + 01)^*(1 + 01)$, or $(1 + 01)^+$.

## EXAMPLE 3.5 The Language of C Identifiers

For this example a little more notation will be useful. Let us temporarily use $l$ (for "letter") to denote the regular expression

$$\mathrm{a} + \mathrm{b} + \cdots + \mathrm{z} + \mathrm{A} + \mathrm{B} + \cdots + \mathrm{Z}$$

and $d$ (for "digit") to stand for

$$0 + 1 + 2 + \cdots + 9$$

An identifier in the C programming language is any string of length 1 or more that contains only letters, digits, and underscores (_) and begins with a letter or an underscore. Therefore, a regular expression for the language of all C identifiers is

$$(l + \_)(l + d + \_)^*$$

## EXAMPLE 3.6 Real Literals in Pascal

Suppose we keep the abbreviations $l$ and $d$ as in the previous example and introduce the additional abbreviations $s$ (for "sign") and $p$ (for "point"). The symbol $s$ is shorthand for $\Lambda + a + m$, where $a$ is "plus" and $m$ is "minus." Consider the regular expression

$$sd^+(pd^+ + pd^+\mathrm{E}sd^+ + \mathrm{E}sd^+)$$

(Here E is not an abbreviation, but one of the symbols in the alphabet.) A typical string corresponding to this regular expression has this form: first a sign (plus, minus, or neither); one or more digits; then *either* a decimal point and one or more digits, which may or may not be followed by an E, a sign, and one or more digits, *or* just the E, the sign, and one or more digits. (If nothing else, you should be convinced by now that one regular expression is often worth several lines of prose.) This is precisely the specification for a real "literal," or constant, in the Pascal programming language. If the constant is in exponential format, no decimal point

is needed. If there is a decimal point, there must be at least one digit immediately preceding and following it.

## 3.2 THE MEMORY REQUIRED TO RECOGNIZE A LANGUAGE

When we discuss the problem of *recognizing* a language (deciding whether an arbitrary input string is in the language), we will be following two conventions for the time being. First, we will restrict ourselves to a single pass through the input, from left to right. Although this restriction is somewhat arbitrary, it allows us to consider how much information must be "remembered" during the processing of the input, and this turns out to be a useful criterion for classifying languages. Second, rather than waiting until the end of the input string to reach a decision (and having to assume that the end of the string is marked explicitly), we make a tentative decision after each input symbol. This allows us to process a string the same way, whether it represents the entire input or a prefix of a longer string. The processing produces a sequence of tentative decisions, one for each prefix, and the final answer for the string is simply the last of these.

The question we want to consider is how much information we need to remember at each step, in order to guarantee that our sequence of decisions will always be correct. The two extremes are that we remember everything (that is, exactly what substring we have read) and that we remember nothing. Remembering nothing might be enough! For example, if the language is empty, the algorithm that answers "no" at each step, regardless of the input, is the correct one; if the language is all of $\Sigma^*$, answering "yes" at each step is correct. In both these trivial cases, since the answer we return is always the same, we can continue to return the right answer without remembering what input symbols we have read, or remembering that we have read one substring rather than another.

In any situation other than these two trivial ones, however, the answers in the sequence are not identical. There are two strings $x$ and $y$ for which the answers are different. This means that the information we remember at the point when we have received input string $x$ must be different from what we remember when we have received input string $y$, for otherwise we would have no way to distinguish the two strings. Therefore, in at least one of these two situations we must remember *something*.

### EXAMPLE 3.7 Strings Ending with 0

Let $L$ be the language $\{0, 1\}^*\{0\}$ of all strings in $\{0, 1\}^*$ that end with 0. Then for any nonnull input string $x$, whether or not $x \in L$ depends only on the last symbol. Another way to say this is that there is no need to distinguish between one string ending with 0 and any other string ending with 0, or between one string ending with 1 and any other string ending with 1. Any two strings ending with the same symbol can be treated exactly the same way.



The only string not accounted for is $\Lambda$. However, there is no need to distinguish between $\Lambda$ and a string ending with 1: Neither is in the language, because neither ends with 0, and once we get one more symbol we will not remember enough to distinguish the resulting strings anyway, because they will both end with the same symbol. The conclusion is that there are only two cases (either the string ends with 0 or it does not), and at each step we must remember only which case we have currently.

### EXAMPLE 3.8 Strings with Next-to-Last Symbol 0

Let $L$ be the language of all strings in $\{0, 1\}^*$ with next-to-last symbol 0. Following the last example, we can say that the decision we make for a string depends on its next-to-last symbol and that we must remember at least that much information. Is that enough? For example, is it necessary to distinguish between the strings 01 and 00, both of which have next-to-last symbol 0?

Any algorithm that does not distinguish between these two strings, and treats them exactly the same, is also unable to distinguish the two strings obtained after one more input symbol. Now it is clear that such an algorithm cannot work: If the next input is 0, for example, the two resulting strings are 010 and 000, and only one of these is in the language. For the same reason, any correct algorithm must also distinguish between 11 and 10 because their last symbols are different.

We conclude that, for this language, it is apparently necessary to remember both the last two symbols. For strings of length at least 2, there are four separate cases. Just as in the previous example, we can see that the three input strings of length less than 2 do not require separate cases. Both the strings $\Lambda$ and 1 can be treated exactly like 11, because for either string at least two more input symbols will be required before the current string is in the language, and at that point, the string we had before those two symbols is irrelevant. The string 0 represents the same case as 10: Neither string is in the language, and once another input is received, both current strings will have the same last two symbols. The four cases we must distinguish are these:

a. The string is $\Lambda$ or 1 or ends with 11.
b. The string is 0 or ends with 10.
c. The string ends with 00.
d. The string ends with 01.

### EXAMPLE 3.9 Strings Ending with 11

This time, let $L = \{0, 1\}^*\{11\}$, the language of all strings in $\{0, 1\}^*$ ending with 11. We can easily formulate an algorithm for recognizing $L$ in which we remember only the last two symbols of the input string. This time, in fact, we can get by with even a little less.

First, it is *not* sufficient to remember only whether the current string ends with 11. For example, suppose the algorithm does not distinguish between a string ending in 01 and one ending in 00, on the grounds that neither ends in 11. Then if the next input is 1, the algorithm will not be able to distinguish between the two new strings, which end in 11 and 01, respectively.

This is not correct, since only one of these strings is in $L$. The algorithm must remember enough now to distinguish between 01 and 00, so that it will be able if necessary to distinguish between 011 and 001 one symbol later.

Two strings ending in 00 and 10, however, do not need to be distinguished. Neither string is in $L$, and no matter what the next symbol is, the two resulting strings will have the same last two symbols. For the same reason, the string 1 can be identified with any string ending in 01.

Finally, the two strings 0 and $\Lambda$ can be identified with all the other strings ending in 0: In all these cases, at least two more input symbols are required to produce an element of $L$, and at that point it will be unnecessary to remember anything but the last two symbols.

Any algorithm recognizing $L$ and following the rules we have adopted must distinguish the following three cases, and it is sufficient for the algorithm to remember which of these the current string represents:

a. The string does not end in 1. (Either it is $\Lambda$ or it ends in 0.)
b. The string is 1 or ends in 01.
c. The string ends in 11.

## EXAMPLE 3.10 Strings with an Even Number of 0's and an Odd Number of 1's

Consider the language of strings $x$ in $\{0, 1\}^*$ for which $n_0(x)$ is even and $n_1(x)$ is odd. One way to get by with remembering less than the entire current string would be to remember just the *numbers* of 0's and 1's we have read, ignoring the way the symbols are arranged. For example, it is not necessary to remember whether the current string is 011 or 101. However, remembering this much information would still require us to consider an infinite number of distinct cases, and an algorithm that remembers much less information can still work correctly. There is no need to distinguish between the strings 011 and 0001111, for example: The current answers are both "no," and the answers will continue to be the same, no matter what input symbols we get from now on. In the case of 011 and 001111, the current answers are also both "no"; however, these two strings must be distinguished, since if the next input is 1, the answer should be "no" in the first case but "yes" in the second.

The reason 011 and 0001111 can be treated the same is that both have an odd number of 0's and an even number of 1's. The reason 011 and 001111 must be distinguished is that one has an odd number of 0's, the other an even number. It is essential to remember the parity (i.e., even or odd) of both the number of 0's and the number of 1's, and this is also sufficient. Once again, we have four distinct cases, and the only information about an input string that we must remember is which of these cases it represents.

## EXAMPLE 3.11 A Recognition Algorithm for the Language in Example 3.4

As in Example 3.4, let

$$L = \{x \in \{0, 1\}^* \mid x \text{ ends in 1 and does not contain the substring 00}\}$$

Suppose that in the course of processing an input string, we have seen the string $s$ so far. If $s$ already contains the substring 00, then that fact is all we need to remember; $s$ is not in $L$, and no matter what input we get from here on, the result will never be in $L$. Let us denote this case by the letter $N$.



Consider next two other cases, in both of which 00 has not yet occurred: case 0, in which the last symbol of $s$ is 0, and case 1, in which the last symbol is 1. In the first case, if the next input is 0 we have case $N$, and if the next input is 1 we have case 1. Starting in case 1, the inputs 0 and 1 take us to cases 0 and 1, respectively. These three cases account for all substrings except $\Lambda$. This string, however, must be distinguished from all the others. It would not be correct to say that the null string corresponds to case $N$, because unlike that case there are possible subsequent inputs that would give us a string in $L$. $\Lambda$ does not correspond to case 0, because if the next input is 0 the answers should be different in the two cases; and it does not correspond to case 1, because the *current* answers should already be different.

Once again we have managed to divide the set $\{0, 1\}^*$ into four types of strings so that in order to recognize strings in $L$ it is sufficient at each step to remember which of the four types we have so far.

We can summarize Examples 3.7–3.11 by the schematic diagrams in Figure 3.1. A diagram like this can be interpreted as a flowchart for an algorithm recognizing the language. In each diagram, the circles correspond to the distinct cases the algorithm is keeping track of, or the distinct types of strings in our classification. The two circles in Figure 3.1*a* (corresponding to Example 3.7) represent strings that do not end with 0 and strings that do, respectively. In Figures 3.1*b* and 3.1*c*, corresponding to Examples 3.8 and 3.9, the circles represent the cases involving the last two symbols that must be distinguished. In Figure 3.1*d*, the label used in each circle is a description of the parities of the number of 0's and the number of 1's, respectively, in the current string. We have already discussed the labeling scheme in Figure 3.1*e*.

In these diagrams, the short arrow not originating at one of the circles indicates the starting point of the algorithm, the case that includes the null string $\Lambda$. The double circles in each case designate cases in which the current string is actually an element of $L$. This is the way the flowchart indicates the answer the algorithm returns for each string.

The arrows originating at a circle tell us, for each possible next symbol, which case results. As we have already described, this information is all the algorithm needs to remember. In Figure 3.1*e*, for example, if at some point we are in case 0 (i.e., the current substring ends in 0 and does not contain 00) and the next symbol is 1, the arrow labeled 1 allows us to forget everything except the fact that the new substring ends with 1 and does not contain 00.

The last sentence of the preceding paragraph is misleading in one sense. Although in studying the algorithm it is helpful to think of case 1 as meaning "The substring we have received so far ends in 1 and does not contain 00," it is not necessary to think of it this way at all. We could give the four cases arbitrary, meaningless labels, and as long as we are able to keep track of which case we are currently in, we will be able to execute the algorithm correctly. The procedure is purely mechanical and requires no understanding of the significance of the cases. A computer program, or a machine, could do it.

This leads us to another possible interpretation of these diagrams, which is the one we will adopt. We think of Figure 3.1*e*, for example, as specifying an *abstract*

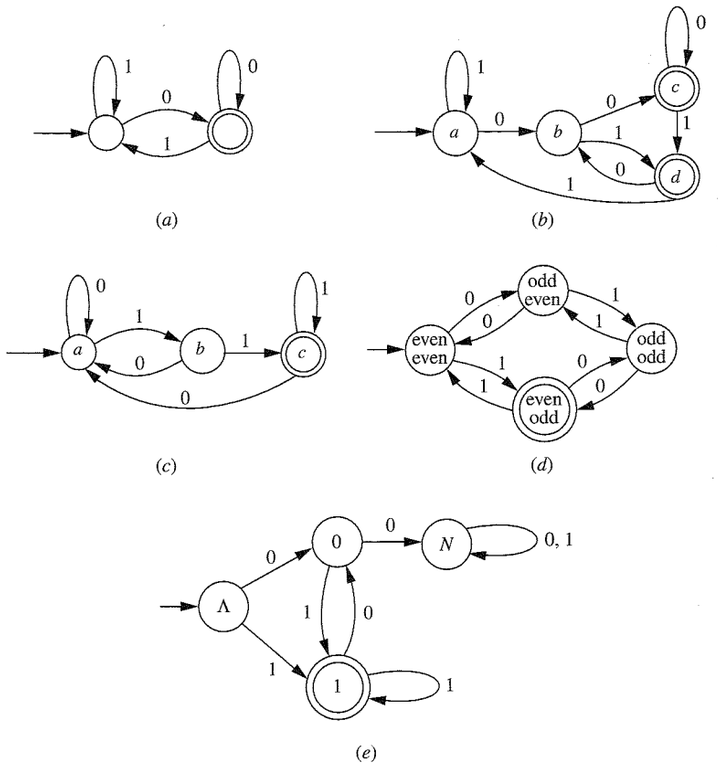



**Figure 3.1 |**
(a) Strings ending in 0; (b) Strings with next-to-last symbol 0; (c) Strings ending with 11; (d) Strings with $n_0$ even and $n_1$ odd; (e) A recognition algorithm for the language in Example 3.4.

*machine* that would work as follows: The machine is at any time in one of four possible *states*, which we have arbitrarily labeled $\Lambda$, 0, 1, and $N$. When it is activated initially, it is in state $\Lambda$. The machine receives successive *inputs* of 0 or 1, and as a result of being in a certain state and receiving a certain input, it moves to the state specified by the corresponding arrow. Finally, certain states are *accepting* states (state 1 is the only one in this example). A string of 0's and 1's is in $L$ if and only if the state the machine is in as a result of processing that string is an accepting state.

It seems reasonable to refer to something of this sort as a "machine," since one can visualize an actual piece of hardware that works according to these rough specifications. The specifications do not say exactly how the hardware works—exactly how the input is transmitted to the machine, for example, or whether a "yes" answer corresponds to a flashing light or a beep. For that matter, the "machine" might exist only in software form, so that the strings are input data to a program. The



phrase *abstract machine* means that it is a specification, in some minimal sense, of the capabilities the machine needs to have. The machine description does not say what physical status the "states" and "inputs" have. The abstraction at the heart of the machine is the *set* of states and the *function* that specifies, for each combination of state and input symbol, the state the machine goes to next. The crucial property is the *finiteness* of the set of states. This is significant because the size of the set puts an absolute limit on the amount of information the machine can (or needs to) remember. Although strings in the language can be arbitrarily long—and will be, unless the language is finite—remembering a fixed amount of information, independent of the size of the input, is sufficient. Being able to distinguish between these states (or between strings that lead to these states) is the only form of memory the machine has.

The more states a machine of this type has, the more complicated a language it will be able to recognize. However, the requirement that the set of states be finite is a significant constraint, and we will be able to find many languages (see Theorem 3.3 for an example) that cannot be recognized by this type of machine, or this type of algorithm. We will show in Chapter 4 that the languages that can be recognized this way are precisely the regular languages. The conclusion, which may not have been obvious from the discussion in Section 3.1, is that regular languages are fairly simple, at least in principle, and there are many languages that are not regular.

## 3.3 | FINITE AUTOMATA

In Section 3.2, we were introduced to a simple type of language-recognizing machine. Now we are ready for the official definition.

**Definition 3.2 Definition of a Finite Automaton**

A *finite automaton*, or *finite-state machine* (abbreviated FA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where

$Q$ is a finite set (whose elements we will think of as *states*);
$\Sigma$ is a finite alphabet of *input symbols*;
$q_0 \in Q$ (the *initial* state);
$A \subseteq Q$ (the set of *accepting* states);
$\delta$ is a function from $Q \times \Sigma$ to $Q$ (the *transition* function).

For any element $q$ of $Q$ and any symbol $a \in \Sigma$, we interpret $\delta(q, a)$ as the state to which the FA moves, if it is in state $q$ and receives the input $a$.

If you have not run into definitions like this before, you might enjoy what the mathematician R. P. Boas had to say about them, in an article in *The American Mathematical Monthly* (88: 727–731, 1981) entitled "Can We Make Mathematics Intelligible?":

> There is a test for identifying some of the future professional mathematicians at an early age. These are students who instantly comprehend a sentence beginning "Let $X$ be an

ordered quintuple $(a, T, \pi, \sigma, \mathcal{B})$, where . . ." They are even more promising if they add, "I never really understood it before."

Whether or not you "instantly comprehend" a definition of this type, you can appreciate the practical advantages. Specifying a finite automaton requires that we specify five things: two sets $Q$ and $\Sigma$, an element $q_0$ and a subset $A$ of $Q$, and a function from $Q \times \Sigma$ to $Q$. Defining a finite automaton to be a 5-tuple may seem strange at first, but it is simply efficient use of notation. It allows us to talk about the five things at once as though we are talking about one "object"; it will allow us to say

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA

instead of

Let $M$ be an FA with state set $Q$, input alphabet $\Sigma$, initial state $q_0$, set of accepting states $A$, and transition function $\delta$.

## EXAMPLE 3.12 Strings Ending with 10

Figure 3.2*a* gives a transition diagram for an FA with seven states recognizing $L = \{0, 1\}^*\{10\}$, the language of all strings in $\{0, 1\}^*$ ending in 10. Until it gets more than two inputs, the FA remembers exactly what it has received (there is a separate state for each possible input string of length 2 or less). After that, it cycles back and forth among four states, "remembering" the last two symbols it has received. Figure 3.2*b* describes this machine in tabular form, by giving the values of the transition function for each state-input pair.

We would expect from Example 3.9 that this FA is more complicated than it needs to be. In particular, the rows for the three states 1, 01, and 11 in the transition table are exactly the same. If $x$, $y$, and $z$ are strings causing the FA to be in the three states, then $x$, $y$, and $z$ do not need to be distinguished now, since none of the three states is an accepting state; and the three strings that will result one input symbol later *cannot* be distinguished. The conclusion is that the three states do not represent cases that need to be distinguished. We could merge them into one and call it state $B$.

The rows for the three states 0, 00, and 10 are also identical. These three states cannot all be merged into one, because 10 is an accepting state and the other two are not. The two nonaccepting states can, and we call the new state $A$.

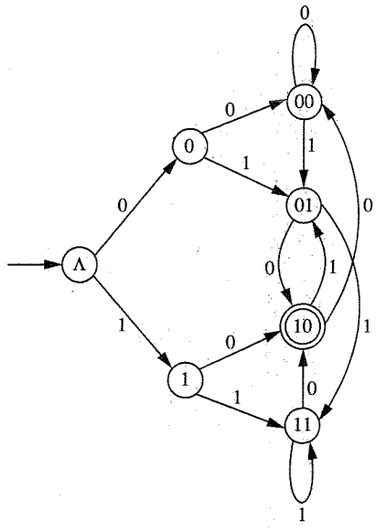
At this point, the number of states has been reduced to 4, and we have the transition table

| | | Input | |
|---|---|---|---|
| | | **0** | **1** |
| **State** | $\Lambda$ | $A$ | $B$ |
| | $A$ | $A$ | $B$ |
| | $B$ | 10 | $B$ |
| | 10 | $A$ | $B$ |

We can go one step further, using the same reasoning as before. In this new FA, the rows for states $A$ and $\Lambda$ are the same, and neither $\Lambda$ nor $A$ is an accepting state. We can therefore include $\Lambda$ in the state $A$ as well. The three resulting states are $A$, $B$, and 10, and it is easy to see that the number cannot be reduced any more. The final result and the corresponding transition table are pictured in Figure 3.3.

We might describe state $A$ as representing "no progress toward 11," meaning that the machine has received either no input at all, or an input string ending with 0 but not 10. $B$ stands for "halfway there," or last symbol 1. As we observed after Example 3.11, however, these descriptions of the states are not needed to specify the abstract machine completely.



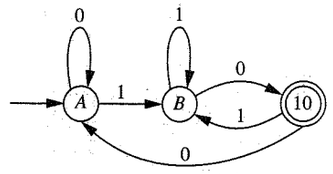



(*a*)

| | | input | |
|---|---|---|---|
| | | **0** | **1** |
| | Λ | 0 | 1 |
| | 0 | 00 | 01 |
| | 1 | 10 | 11 |
| **state** | 00 | 00 | 01 |
| | 01 | 10 | 11 |
| | 10 | 00 | 01 |
| | 11 | 10 | 11 |

(*b*)

**Figure 3.2 |**
A finite automaton recognizing $\{0, 1\}^*\{10\}$. (a) Transition diagram; (b) Transition table.




(*a*)

| | | input | |
|---|---|---|---|
| | | **0** | **1** |
| | $A$ | $A$ | $B$ |
| **state** | $B$ | 10 | $B$ |
| | 10 | $A$ | $B$ |

(*b*)

**Figure 3.3 |**
A simplified finite automaton recognizing $\{0, 1\}^*\{10\}$: (*a*) Transition diagram; (*b*) Transition table.

The analysis that led to the simplification of the FA in this example can be turned into a systematic procedure for minimizing the number of states in a given FA. See Section 5.2 for more details.

For an arbitrary FA $M = (Q, \Sigma, q_0, A, \delta)$, the expression $\delta(q, a)$ is a concise way of writing "the state to which the FA goes, if it is in state $q$ and receives input $a$." The next step is to extend the notation so that we can describe equally concisely "the state in which the FA ends up, if it begins in state $q$ and receives the string $x$ of input symbols." Let us write this as $\delta^*(q, x)$. The function $\delta^*$, then, is an extension of the transition function $\delta$ from the set $Q \times \Sigma$ to the larger set $Q \times \Sigma^*$. How do we define the function $\delta^*$ precisely? The idea behind the function is simple enough. If $x$ is the string $a_1a_2 \ldots a_n$, we want to obtain $\delta^*(q, x)$ by first going to the state $q_1$ to which $M$ goes from state $q$ on input $a_1$; then going to the state $q_2$ to which $M$ goes from $q_1$ on input $a_2$; ...; and finally, going to the state $q_n$ to which $M$ goes from $q_{n-1}$ on input $a_n$. Unfortunately, so far this does not sound either particularly precise or particularly concise. Although we can replace many of the phrases by mathematical formulas (for example, "the state to which $M$ goes from state $q$ on input $a_1$" is simply $\delta(q, a_1)$), we still have the problem of the ellipses. At this point, we might be reminded of the discussion in Example 2.14.

The easiest way to define $\delta^*$ precisely is to give a recursive definition. For a particular state $q$, we are trying to define the expression $\delta^*(q, x)$ for each string $x$ in $\Sigma^*$. Using the recursive definition of $\Sigma^*$ (Example 2.15), we can proceed as follows: Define $\delta^*(q, \Lambda)$; then, assuming we know what $\delta^*(q, y)$ is, define $\delta^*(q, ya)$ for an element $a$ of $\Sigma$.

The "basis" part of the definition is not hard to figure out. We do not expect the state of the FA to change as a result of getting the input string $\Lambda$, and we define $\delta^*(q, \Lambda) = q$, for every $q \in Q$. Now, $\delta^*(q, ya)$ is to be the state that results when $M$ begins in state $q$ and receives first the input string $y$, then the single additional symbol $a$. The state $M$ is in after getting $y$ is $\delta^*(q, y)$; and from any state $p$, the state to which $M$ moves from $p$ on the single input symbol $a$ is $\delta(p, a)$. This means that the recursive part of the definition should define $\delta^*(q, ya)$ to be $\delta(\delta^*(q, y), a)$.

**Definition 3.3 The Extended Transition Function $\delta^*$**

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. We define the function

$$\delta^* : Q \times \Sigma^* \to Q$$

as follows:

1. For any $q \in Q$, $\delta^*(q, \Lambda) = q$
2. For any $q \in Q$, $y \in \Sigma^*$, and $a \in \Sigma$,

$$\delta^*(q, ya) = \delta(\delta^*(q, y), a)$$

It is important to understand that, in adopting the recursive definition of $\delta^*$, we have not abandoned the intuitive idea with which we first approached the definition.



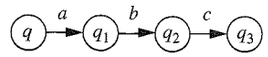



**Figure 3.4 |**

The point is that the recursive definition is the best way to capture this intuitive idea in a formal definition. Using this definition to calculate $\delta^*(q, x)$ amounts to just what you would expect, given what we wanted the function $\delta^*$ to represent: processing the symbols of $x$, one at a time, and seeing where the transition function $\delta$ takes us at each step. Suppose for example that $M$ contains the transitions shown in Figure 3.4.

Let us use Definition 3.3 to calculate $\delta^*(q, abc)$:

$$\begin{aligned}
\delta^*(q, abc) &= \delta(\delta^*(q, ab), c) \\
&= \delta(\delta(\delta^*(q, a), b), c) \\
&= \delta(\delta(\delta^*(q, \Lambda a), b), c) \\
&= \delta(\delta(\delta(\delta^*(q, \Lambda), a), b), c) \\
&= \delta(\delta(\delta(q, a), b), c) \\
&= \delta(\delta(q_1, b), c) \\
&= \delta(q_2, c) \\
&= q_3
\end{aligned}$$

Note that in the calculation above, it was necessary to calculate $\delta^*(q, a)$ by using the recursive part of the definition, since the basis part involves $\delta^*(q, \Lambda)$. Fortunately, $\delta^*(q, a)$ turned out to be $\delta(q, a)$; for strings of length 1 (i.e., elements of $\Sigma$), $\delta$ and $\delta^*$ can be used interchangeably. For a string $x$ with $|x| \neq 1$, however, writing $\delta(q, x)$ is incorrect, because the pair $(q, x)$ does not belong to the domain of $\delta$.

Other properties you would expect $\delta^*$ to satisfy can be derived from our definition. For example, a natural generalization of statement 2 of the definition is the formula

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

which should be true for any $q \in Q$ and any two strings $x$ and $y$. The proof is by mathematical induction, and the details are left to you in Exercise 3.22.

Now we can state more concisely what it means for an FA to accept a string and what it means for an FA to accept a language.

**Definition 3.4 Acceptance by an FA**

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. A string $x \in \Sigma^*$ is *accepted* by $M$ if $\delta^*(q_0, x) \in A$. If a string is not accepted, we say it is *rejected* by $M$. The *language* accepted by $M$, or the language recognized by $M$, is the set

$$L(M) = \{x \in \Sigma^* \mid x \text{ is accepted by } M\}$$

If $L$ is any language over $\Sigma$, $L$ is accepted, or recognized, by $M$ if and only if $L = L(M)$.


**Figure 3.5 |**

Notice what the last statement in the definition does *not* say. It does not say that $L$ is accepted by $M$ if every string in $L$ is accepted by $M$. If it did, we could use the FA in Figure 3.5 to accept any language, no matter how complex. The power of a machine does not lie in the number of strings it accepts, but in its ability to discriminate—to accept some and reject others. In order to accept a language $L$, an FA has to accept all the strings in $L$ *and* reject all the strings in $L'$.

The terminology introduced in Definition 3.4 allows us to record officially the following fact, which we have mentioned already but will not prove until Chapter 4.

**Theorem 3.1**
A language $L$ over the alphabet $\Sigma$ is regular if and only if there is an FA with input alphabet $\Sigma$ that accepts $L$.

This theorem says on the one hand that if $M$ is any FA, there is a regular expression corresponding to the language $L(M)$; and on the other hand, that given a regular expression, there is an FA that accepts the corresponding language. The proofs in Chapter 4 will actually give us ways of constructing both these things. Until then, many examples are simple enough that we can get by without a formal algorithm.

**EXAMPLE 3.13** Finding a Regular Expression Corresponding to an FA

Let us try to describe by a regular expression the language $L$ accepted by the FA in Figure 3.6. The state labeled $A$ is both the initial state and an accepting state; this tells us that $\Lambda \in L$. More generally, even-length strings of 0's (of which $\Lambda$ is one) are in $L$. These are the only strings $x$ for which $\delta^*(A, x) = A$, because the only arrow to $A$ from another state is the one from state 0, and the only arrow to that state is the one from $A$. These strings correspond to the regular expression $(00)^*$.

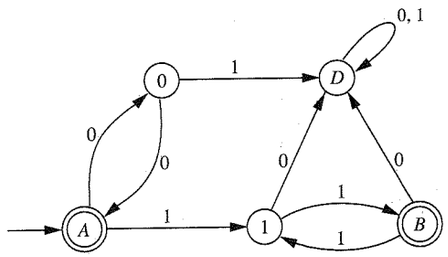



**Figure 3.6 |**
A finite automaton accepting $\{00\}^*\{11\}^*$.



The state labeled $D$ serves the same purpose in this example that $N$ did in Example 3.11. Once the FA reaches $D$, it stays in that state; a string $x$ for which $\delta^*(A, x) = D$ cannot be a prefix of any element of $L$.

The easiest way to reach the other accepting state $B$ from $A$ is with the string 11. Once the FA is in state $B$, any even-length string of 1's returns it to $B$, and these are the only strings that do this. Therefore, if $x$ is a string that causes the FA to go from $A$ to $B$ without revisiting $A$, $x$ must be of the form $11(11)^k$ for some $k \geq 0$. The most general type of string that causes the FA to reach state $B$ is a string of this type preceded by one of the form $(00)^j$, and a corresponding regular expression is $(00)^*11(11)^*$.

By combining the two cases (the strings $x$ for which $\delta^*(A, x) = A$ and those for which $\delta^*(A, x) = B$), we conclude that the language $L$ corresponds to the regular expression $(00)^* + (00)^*11(11)^*$, which can be simplified to

$$(00)^*(11)^*$$

**EXAMPLE 3.14** Another Example of a Regular Expression Corresponding to an FA

Next we consider the FA $M$ in Figure 3.7, with input alphabet $\{a, b\}$. One noteworthy feature of this FA is the fact that every arrow labeled $b$ takes the machine to state $B$. As a result, every string ending in $b$ causes $M$ to be in state $B$; in other words, for any string $x$, $\delta^*(A, xb) = B$. Therefore, $\delta^*(A, xbaaa) = \delta^*(\delta^*(A, xb), aaa) = \delta^*(B, aaa) = E$, which means that any string ending in $baaa$ is accepted by $M$. (The first of these equalities uses the formula in Exercise 3.22.)

On the other hand, we can see from the diagram that the only way to get to state $E$ is to reach $D$ first and then receive input $a$; the only way to reach $D$ is to reach $C$ and then receive $a$; the only way to reach $C$ is to reach $B$ and then receive $a$; and the only way to reach $B$ is to receive input $b$, although this could happen in any state. Therefore, it is also true that any

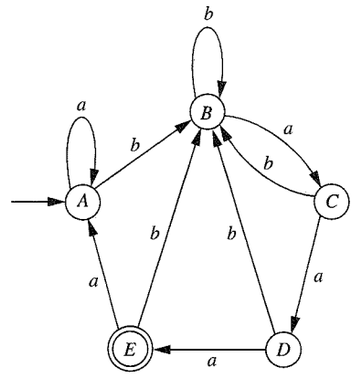



**Figure 3.7 |**
A finite automaton $M$ accepting $\{a, b\}^*\{baaa\}$.

string accepted by $M$ must end in $baaa$. The language $L(M)$ is the set of all strings ending in $baaa$, and a regular expression corresponding to $L(M)$ is $(a + b)^*baaa$.

This is a roundabout way of arriving at a regular expression. It depends on our noticing certain distinctive features of the FA, and it is not clear that the approach will be useful for other machines. An approach that might seem more direct is to start at state $A$ and try to build a regular expression as we move toward $E$. Since $\delta(A, a) = A$, we begin the regular expression with $a^*$. Since $\delta(A, b) = B$ and $\delta(B, b) = B$, we might write $a^*bb^*$ next. Now the symbol $a$ takes us to $C$, and we try $a^*bb^*a$. At this point it starts to get complicated, however, because we can now go back to $B$, loop some more with input $b$, then return to $C$—and we can repeat this any number of times. This might suggest $a^*bb^*a(bb^*a)^*$. As we get closer to $E$, there are more loops, and loops within loops, to take into account, and the formulas quickly become unwieldy. We might or might not be able to carry this to completion, and even if we can, the resulting formula will be complicated. We emphasize again that we do not yet have a systematic way to solve these problems, and there is no need to worry at this stage about complicated examples.

## EXAMPLE 3.15 Strings Containing Either *ab* or *bba*

In this example we consider the language $L$ of all strings in $\{a, b\}^*$ that contain at least one of the two substrings $ab$ and $bba$ ($L$ corresponds to the regular expression $(a+b)^*(ab+bba)(a+b)^*$), and try to construct a finite automaton accepting $L$.

We start with two observations. These two strings themselves should be accepted by our FA; and if $x$ is any string that is accepted, then any other string obtained by adding symbols to the end of $x$ should also be accepted. Figure 3.8a shows a first attempt at an FA (obviously uncompleted) incorporating these features. (We might have started with two separate accepting states, but the transitions from both would have been the same, and one is enough.)

In order to continue, we need transitions labeled $a$ and $b$ from each of the states $p$, $r$, and $s$, and we may need additional states. It will help to think at this point about what each of the

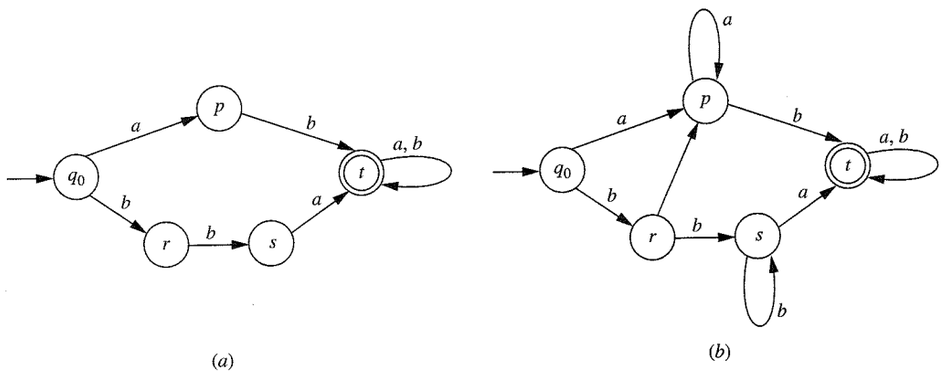



**Figure 3.8 |**
Strings containing either $ab$ or $bba$.



nonaccepting states is supposed to represent. First, being in a nonaccepting state means that we have not yet received one of the two desired strings. Being in the initial state $q_0$ should presumably mean that we have made no progress at all toward getting one of these two strings. It seems as though any input symbol at all represents some progress, however: An $a$ is at least potentially the first symbol in the substring $ab$, and $b$ might be the first symbol in $bba$. This suggests that once we have at least one input symbol, we should never need to return to the initial state.

It is not correct to say that $p$ is the state the FA should be in if the last input symbol received was an $a$, because there are arrows labeled $a$ that go to state $t$. It should be possible, however, to let $p$ represent the state in which the last input symbol was $a$ and we have not yet seen either $ab$ or $bba$. If we are already in this state and the next input symbol we receive is $a$, then nothing has really changed; in other words, $\delta(p, a)$ should be $p$.

We can describe the states $r$ and $s$ similarly. If the last input symbol was $b$, and it was not preceded by a $b$, and we have not yet arrived in the accepting state, we should be in state $r$. We should be in state $s$ if we have just received two consecutive $b$'s but have not yet reached the accepting state. What should $\delta(r, a)$ be? The $b$ that got us to state $r$ is no longer doing us any good, since it was not followed by $b$. In other words, it looked briefly as though we were making progress toward getting the string $bba$, but now it appears that we are not. We do not have to start over, however, because at least we have an $a$. We conclude that $\delta(r, a) = p$. We can also see that $\delta(s, b) = s$: If we are in state $s$ and get input $b$, we have not made any further progress toward getting $bba$, but neither have we lost any ground.

At this point, we have managed to define the missing transitions in Figure 3.8$a$ without adding any more states, and thus we have an FA that accepts the language $L$. The transition diagram is shown in Figure 3.8$b$.

## Another Example of an FA Corresponding to a Regular Expression EXAMPLE 3.16

We consider another regular expression,

$$r = (11 + 110)^*0$$

and try to construct an FA accepting the corresponding language $L$. In the previous example, our preliminary guess at the structure of the FA turned out to provide all the states we needed. Here it will not be so straightforward. We will just proceed one symbol at a time, and for each new transition try to determine whether it can go to a state that is already present or whether it will require a new state.

The null string is not in $L$, which tells us that the initial state $q_0$ should not be an accepting state. The string 0, however, is in $L$, so that from $q_0$ the input symbol 0 must take us to an accepting state. The string 1 is not in $L$; furthermore, 1 must be distinguished from $\Lambda$, because the subsequent input string 110 should take us to an accepting state in one case and not in the other (i.e., $110 \in L$ and $1110 \notin L$.) At this point, we have determined that we need at least the states in Figure 3.9$a$.

The language $L$ contains 0 but no other strings beginning with 0. It also contains no strings beginning with 10. It is appropriate, therefore, to introduce a state $s$ that represents all the strings that fail for either reason to be a prefix of an element of $L$ (Figure 3.9$b$). Once our FA reaches the state $s$, which is not an accepting state, it never leaves this state.

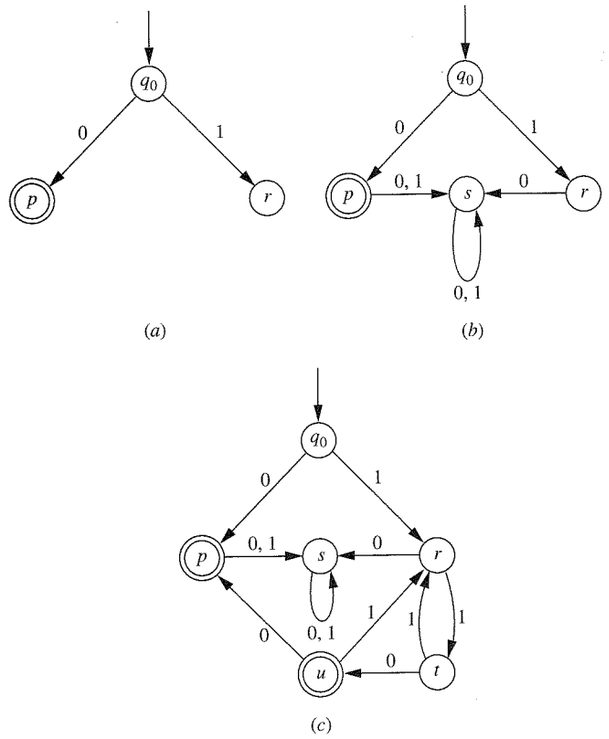



**Figure 3.9 |**
A finite automaton accepting $L_3$.

We must now consider the situation in which the FA is in state $r$ and receives the input 1. It should not stay in state $r$, because the strings 1 and 11 need to be distinguished (for example, $110 \in L$, but $1110 \notin L$). It should not return to the initial state, because $\Lambda$ and 11 need to be distinguished. Therefore, we need a new state $t$. From $t$, the input 0 must lead to an accepting state, since $110 \in L$. This accepting state cannot be $p$, because 110 is a prefix of a longer string in $L$ and 0 is not. Let $u$ be the new accepting state. If the FA receives a 0 in state $u$, then we have the same situation as an initial 0: The string 1100 is in $L$ but is not a prefix of any longer string in $L$. So we may let $\delta(u, 0) = p$. We have yet to define $\delta(t, 1)$ and $\delta(u, 1)$. States $t$ and $u$ can be thought of as "the end of one of the strings 11 and 110." (The reason $u$ is accepting is that 110 can also be viewed as 11 followed by 0.) In either case, if the next symbol is 1, we should think of it as the first symbol in *another* occurrence of one of these two strings. This means that it is appropriate to define $\delta(t, 1) = \delta(u, 1) = r$, and we arrive at the FA shown in Figure 3.9*c*.

The procedure we have followed here may seem hit-or-miss. We continued to add states as long as it was necessary, stopping only when all transitions from every state had been drawn



and went to states that were already present. Theorem 3.1 is the reason we can be sure that the process *will* eventually stop. If we used the same approach for a language that was not regular, we would never be able to stop: No matter how many states we created, defining the transitions from those states would require yet more states. The step that is least obvious and most laborious in our procedure is determining whether a given transition needs to go to a new state, and if not, which existing state is the right one. The algorithm that we develop in Chapter 4 uses a different approach that avoids this difficulty.

## 3.4 | DISTINGUISHING ONE STRING FROM ANOTHER

Using a finite automaton to recognize a language $L$ depends on the fact that there are groups of strings so that strings within the same group do not need to be distinguished from each other by the machine. In other words, it is not necessary for the machine to remember exactly which string within the group it has read so far; remembering which group the string belongs to is enough. The number of distinct states the FA needs in order to recognize a language is related to the number of distinct strings that must be distinguished from each other. The following definition specifies precisely the circumstances under which an FA recognizing $L$ must distinguish between two strings $x$ and $y$, and the lemma that follows spells out explicitly how such an FA accomplishes this. (It says simply that the FA is in different states as a result of processing the two strings).

**Definition 3.5 Distinguishable Strings with Respect to *L***

Let $L$ be a language in $\Sigma^*$, and $x$ any string in $\Sigma^*$. The set $L/x$ is defined as follows:

$$L/x = \{z \in \Sigma^* \mid xz \in L\}$$

Two strings $x$ and $y$ are said to be *distinguishable with respect to* $L$ if $L/x \neq L/y$. Any string $z$ that is in one of the two sets but not the other (i.e., for which $xz \in L$ and $yz \notin L$, or vice versa) is said to distinguish $x$ and $y$ with respect to $L$. If $L/x = L/y$, $x$ and $y$ are indistinguishable with respect to $L$.

In order to show that two strings $x$ and $y$ are distinguishable with respect to a language $L$, it is sufficient to find one string $z$ so that either $xz \in L$ and $yz \notin L$, or $xz \notin L$ and $yz \in L$ (in other words, so that $z$ is in one of the two sets $L/x$ and $L/y$ but not the other). For example, if $L$ is the language in Example 3.9, the set of all strings in $\{0, 1\}^*$ that end in 10, we observed that 00 and 01 are distinguishable with respect to $L$, because we can choose $z$ to be the string 0; that is, $000 \notin L$ and

$010 \in L$. The two strings 0 and 00 are indistinguishable with respect to $L$, because the two sets $L/0$ and $L/00$ are equal; each is just the set $L$ itself. (The only way $0z$ or $00z$ can end in 10 is for $z$ to have this property.)

**Lemma 3.1** Suppose $L \subseteq \Sigma^*$ and $M = (Q, \Sigma, q_0, A, \delta)$ is an FA recognizing $L$. If $x$ and $y$ are two strings in $\Sigma^*$ that are distinguishable with respect to $L$, then $\delta^*(q_0, x) \neq \delta^*(q_0, y)$.

***Proof*** The assumption that $x$ and $y$ are distinguishable with respect to $L$ means that there is a string $z$ in one of the two sets $L/x$ and $L/y$ but not the other. In other words, one of the two strings $xz$ and $yz$ is in $L$ and the other is not. Because we are also assuming that $M$ accepts $L$, it follows that one of the two states $\delta^*(q_0, xz)$ and $\delta^*(q_0, yz)$ is an accepting state and the other is not. In particular, therefore,

$$\delta^*(q_0, xz) \neq \delta^*(q_0, yz)$$

According to Exercise 3.22,

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

Because the left sides of these two equations are unequal, the right sides must be, and therefore $\delta^*(q_0, x) \neq \delta^*(q_0, y)$. ■

**Theorem 3.2**
Suppose $L \subseteq \Sigma^*$ and, for some positive integer $n$, there are $n$ strings in $\Sigma^*$, any two of which are distinguishable with respect to $L$. Then every FA recognizing $L$ must have at least $n$ states.

***Proof***
Suppose $x_1, x_2, \ldots, x_n$ are $n$ strings, any two of which are distinguishable with respect to $L$. If $M = (Q, \Sigma, q_0, A, \delta)$ is any FA with fewer than $n$ states, then by the pigeonhole principle (Exercise 2.44), the states $\delta^*(q_0, x_1), \delta^*(q_0, x_2), \ldots, \delta^*(q_0, x_n)$ are not all distinct, and so for some $i \neq j$, $\delta^*(q_0, x_i) = \delta^*(q_0, x_j)$. Since $x_i$ and $x_j$ are distinguishable with respect to $L$, it follows from the lemma that $M$ cannot recognize $L$.

We interpret Theorem 3.2 as putting a lower bound on the memory requirements of any FA that is capable of recognizing $L$. To make this interpretation more concrete, we might think of the states as being numbered from 1 through $n$ and assume that there is a register in the machine that stores the number of the current state. The binary representation of the number $n$ has approximately $\log_2(n)$ binary digits, and thus the register needs to be able to hold approximately $\log_2(n)$ bits of information.



## The Language $L_n$ EXAMPLE 3.17

Suppose $n \geq 1$, and let

$$L_n = \{x \in \{0, 1\}^* \mid |x| \geq n \text{ and the } n\text{th symbol from the right in } x \text{ is } 1\}$$

There is a straightforward way to construct an FA recognizing $L_n$, by creating a distinct state for every possible substring of length $n$ or less, just as we did in Example 3.12. In this way the FA will be able to remember the last $n$ symbols of the current input string. For each $i$, the number of strings of length exactly $i$ is $2^i$. If we add these numbers for the values of $i$ from 0 to $n$, we obtain $2^{n+1} - 1$ (Exercise 2.13); therefore, this is the total number of states. Figure 3.10 illustrates this FA in the case $n = 3$.

The eight states representing strings of length 3 are at the right. Not all the transitions from these states are shown, but the rule is simply

$$\delta(abc, d) = bcd$$

The accepting states are the four for which the third symbol from the end is 1.

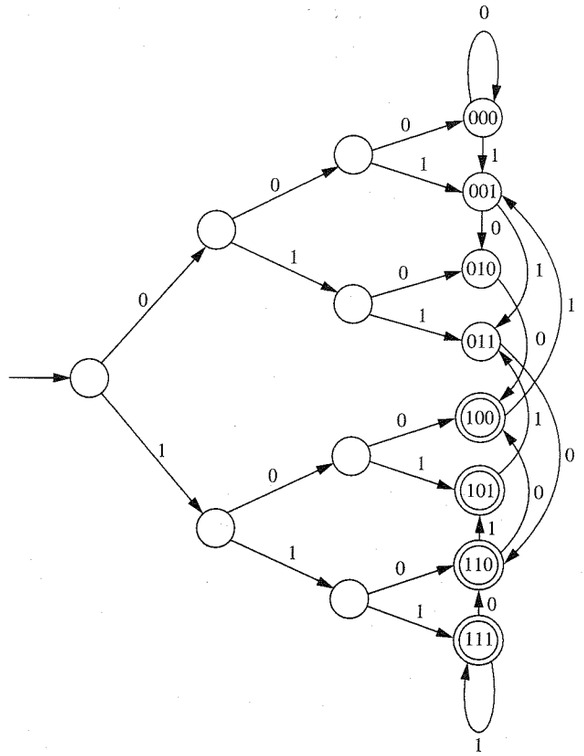



**Figure 3.10 |**
A finite automaton accepting $L_3$.

We might ask whether there is some simpler FA, perhaps using a completely different approach, that would cut the number of states to a number more manageable than $2^{n+1} - 1$. Although the number can be reduced just as in Example 3.12, we can see from Theorem 3.2 that any FA recognizing $L_n$ must have at least $2^n$ states. To do this, we show that any two strings of length $n$ (of which there are $2^n$ in all) are distinguishable with respect to $L_n$.

Let $x$ and $y$ be two distinct strings of length $n$. They must differ in the $i$th symbol (from the left), for some $i$ with $1 \leq i \leq n$. For the string $z$ that we will use to distinguish these two strings with respect to $L_n$, we can choose any string of length $i - 1$. Then $xz$ and $yz$ still differ in the $i$th symbol, and now the $i$th position is precisely the $n$th from the right. In other words, one of the strings $xz$ and $yz$ is in $L$ and the other is not, which implies that $L/x \neq L/y$. Therefore, $x$ and $y$ are distinguishable with respect to $L_n$.

Theorem 3.2 is potentially a way of showing that some languages cannot be accepted by any FA. According to the theorem, if there is a large set of strings, any two of which are distinguishable with respect to $L$, then any FA accepting $L$ must have a large number of states. What if there is a *really* large (i.e., infinite) set $S$ of strings with this property? Then no matter what $n$ is, we can choose a subset of $S$ with $n$ elements, which will be "pairwise distinguishable" (i.e., any two elements are distinguishable). Therefore, no matter what $n$ is, any FA accepting $L$ must have at least $n$ states. If this is true, the only way an FA can recognize $L$ at all is for it to have an infinite set of states, which is exactly what a *finite* automaton is not allowed to have. Such a language $L$ cannot be recognized by any abstract machine, or algorithm, of the type we have described, because no matter how much memory we provide, it will not be enough.

The following theorem provides our first example of such a language. In the proof, we show not only that there is an infinite "pairwise distinguishable" set of strings, but an even stronger statement, that the set $\Sigma^*$ itself has this property: *Any* two strings are distinguishable with respect to the language. This means that in attempting to recognize the language, following the conventions we have adopted, we cannot afford to forget anything. No matter what input string $x$ we have received so far, we must remember that it is $x$ and not some other string. The second conclusion of the theorem is a result of Theorem 3.1.

**Theorem 3.3**

The language *pal* of palindromes over the alphabet $\{0, 1\}$ cannot be accepted by any finite automaton, and it is therefore not regular.

***Proof***

As we described above, we will show that for any two distinct strings $x$ and $y$ in $\{0, 1\}^*$, $x$ and $y$ are distinguishable with respect to *pal*. To show this, we consider first the case when $|x| = |y|$, and we let $z = x^r$. Then $xz = xx^r$, which is in *pal*, and $yz$ is not. If $|x| \neq |y|$, we may as well assume that $|x| < |y|$, and we let $y = y_1y_2$, where $|y_1| = |x|$. Again, we look for a string $z$ so that $xz \in pal$ and $yz \notin pal$. Any $z$ of the form $z = ww^rx^r$ satisfies $xz \in pal$. In order to guarantee that $yz \notin pal$, we can choose $w$ to be any string different from $y_2$ but of the same length. Then

$$yz = y_1y_2z = y_1y_2ww^rx^r$$

Of the five substrings, the two on the ends both have length $|x|$, and all three of the others have length $|y_2|$. In order for $yz$ to be a palindrome, therefore, the fourth substring, $w^r$, must be the reverse of the second, $y_2$. This is impossible, however, since $w \neq y_2$. In either case, we have shown that $L/x \neq L/y$ (where $L = pal$).



In Chapter 5 we will consider other nonregular languages and find other methods for demonstrating that a language is nonregular. Definition 3.5 will also come up again in Chapter 5; the indistinguishability relation can be used in an elegant description of a "minimum-state" FA recognizing a given regular language.

## 3.5 UNIONS, INTERSECTIONS, AND COMPLEMENTS

Suppose $L_1$ and $L_2$ are both regular languages over an alphabet $\Sigma$. There are FAs $M_1$ and $M_2$ accepting $L_1$ and $L_2$, respectively (Theorem 3.1); on the other hand, the languages $L_1 \cup L_2$, $L_1L_2$, and $L_1^*$ are also regular (Definition 3.1) and can therefore be accepted by FAs. It makes sense to ask whether there are natural ways to obtain machines for these three languages from the two machines $M_1$ and $M_2$.

The language $L_1 \cup L_2$ is different in one important respect from the other two: Whether a string $x$ belongs to $L_1 \cup L_2$ depends only on whether $x \in L_1$ and whether $x \in L_2$. As a result, not only is there a simple solution to the problem in this case, but with only minor changes the same method also works for the two languages $L_1 \cap L_2$ and $L_1 - L_2$. We will wait until Chapter 4 to consider the two remaining languages $L_1L_2$ and $L_1^*$.

If as we receive input symbols we execute two algorithms simultaneously, one to determine whether the current string $x$ is in $L_1$, the other to determine whether $x$ is in $L_2$, we will be able to say at each step whether $x$ is in $L_1 \cup L_2$. If $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are FAs recognizing $L_1$ and $L_2$, respectively, a finite automaton $M$ should be able to recognize $L_1 \cup L_2$ if it can remember at each step both the information that $M_1$ remembers and the information $M_2$ remembers. Abstractly, this amounts to "remembering" the ordered pair $(p, q)$, where $p$ and $q$ are the current states of $M_1$ and $M_2$. Accordingly, we can construct $M$ by taking our set of states to be the set of all possible ordered pairs, $Q_1 \times Q_2$. The initial state will be the pair $(q_1, q_2)$ of initial states. If $M$ is in the state $(p, q)$ (which means that $p$ and $q$ are the current states of $M_1$ and $M_2$) and receives input symbol $a$, it should move to the state $(\delta_1(p, a), \delta_2(q, a))$, since the two components of this pair are the states to which the individual machines would move.

What we have done so far is independent of which of the three languages ($L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 - L_2$) we wish to accept. All that remains is to specify the set of

accepting states so that the strings accepted are the ones we want. For the language $L_1 \cup L_2$, for example, $x$ should be accepted if it is in either $L_1$ or $L_2$; this means that the state $(p, q)$ should be an accepting state if either $p$ or $q$ is an accepting state of its respective FA. For the languages $L_1 \cap L_2$ and $L_1 - L_2$, the accepting states of the machine are defined similarly.

**Theorem 3.4**
Suppose $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ accept languages $L_1$ and $L_2$, respectively. Let $M$ be an FA defined by $M = (Q, \Sigma, q_0, A, \delta)$, where

$$Q = Q_1 \times Q_2$$
$$q_0 = (q_1, q_2)$$

and the transition function $\delta$ is defined by the formula

$$\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$$

(for any $p \in Q_1$, $q \in Q_2$, and $a \in \Sigma$). Then

1. If $A = \{(p, q) \mid p \in A_1 \text{ or } q \in A_2\}$, $M$ accepts the language $L_1 \cup L_2$;
2. If $A = \{(p, q) \mid p \in A_1 \text{ and } q \in A_2\}$, $M$ accepts the language $L_1 \cap L_2$;
3. If $A = \{(p, q) \mid p \in A_1 \text{ and } q \notin A_2\}$, $M$ accepts the language $L_1 - L_2$.

***Proof***
We have already sketched the main idea. Since acceptance by $M_1$ and $M_2$ is defined in terms of the functions $\delta_1^*$ and $\delta_2^*$, respectively, and acceptance by $M$ in terms of $\delta^*$, we need the formula

$$\delta^*((p, q), x) = (\delta_1^*(p, x), \delta_2^*(q, x))$$

which holds for any $x \in \Sigma^*$ and any $(p, q) \in Q$, and can be verified easily by using mathematical induction (Exercise 3.32). A string $x$ is accepted by $M$ if and only if $\delta^*((q_1, q_2), x) \in A$. By our formula, this is true if and only if

$$(\delta_1^*(q_1, x), \delta_2^*(q_2, x)) \in A$$

If the set $A$ is defined as in case 1, this is the same as saying that $\delta_1^*(q_1, x) \in A_1$ or $\delta_2^*(q_2, x) \in A_2$, or in other words, that $x \in L_1 \cup L_2$. Cases 2 and 3 are similar.

We may consider the special case in which $L_1$ is all of $\Sigma^*$. $L_1 - L_2$ is therefore $L_2'$, the complement of $L_2$. The construction in the theorem can be used, where $M_1$ is the trivial FA with only the state $q_1$, which is an accepting state. However, this description of $M$ is unnecessarily complicated. Except for the names of the states, it is the same as

$$M_2' = (Q_2, \Sigma, q_2, Q_2 - A_2, \delta_2)$$

which we obtain from $M_2$ by just reversing accepting and nonaccepting states.



It often happens, as in the following example, that the FA we need for one of these three languages is even simpler than the construction in Theorem 3.4 would seem to indicate.

## An FA Accepting $L_1 - L_2$ — EXAMPLE 3.18

Suppose $L_1$ and $L_2$ are the subsets

$$L_1 = \{x \mid 00 \text{ is not a substring of } x\}$$
$$L_2 = \{x \mid x \text{ ends with } 01\}$$

of $\{0, 1\}^*$. The languages $L_1$ and $L_2$ are recognized by the FAs in Figure 3.11*a*.

The construction in the theorem, for any of the three cases, produces an FA with nine states. In order to draw the transition diagram, we begin with the initial state $(A, P)$. Since $\delta_1(A, 0) = B$ and $\delta_2(P, 0) = Q$, we have $\delta((A, P), 0) = (B, Q)$. Similarly, $\delta((A, P), 1) = (A, P)$. Next we calculate $\delta((B, Q), 0)$ and $\delta((B, Q), 1)$. As we continue this process, as

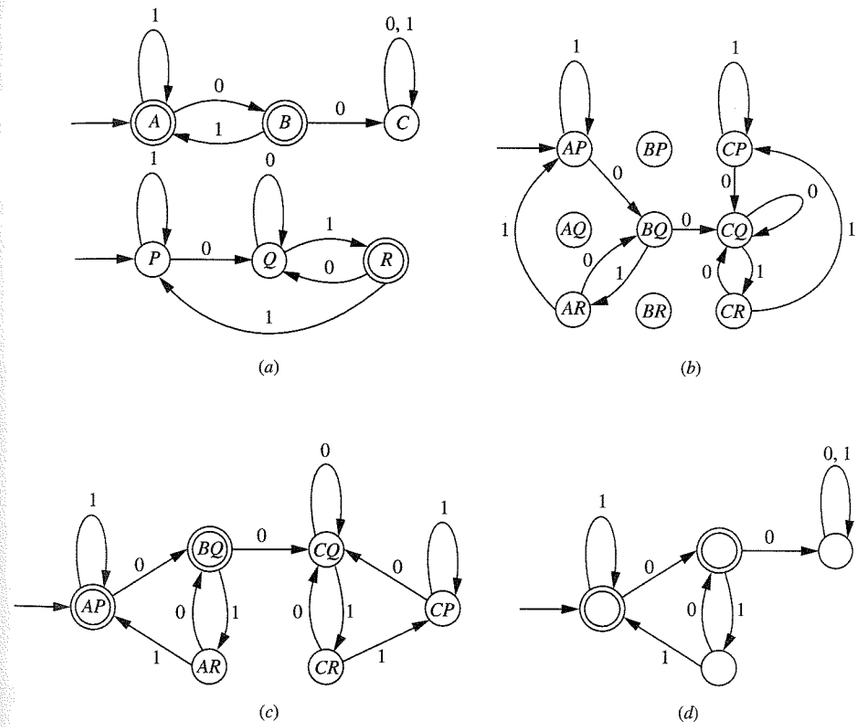



**Figure 3.11 |**
Constructing a finite automaton to accept $L_1 - L_2$.

soon as a new state is introduced, we calculate the transitions from this state. After a few steps we obtain the partial diagram in Figure 3.11*b*. We now have six states; each of them can be reached from $(A, P)$ as a result of some input string, and every transition from one of these six goes to one of these six. We conclude that the other three states are not reachable from the initial state, and therefore that they can be left out of our FA (Exercise 3.29).

Suppose now that we want our FA to recognize the language $L_1 - L_2$. Then we designate as our accepting states those states $(X, Y)$ from among the six for which $X$ is either $A$ or $B$ and $Y$ is not $R$. These are $(A, P)$ and $(B, Q)$, and the resulting FA is shown in Figure 3.11*c*.

In fact, we can simplify this FA even further. None of the states $(C, P)$, $(C, Q)$, or $(C, R)$ is an accepting state, and once the machine enters one of these states, it remains in one of them. Therefore, we may replace all of them with a single state, obtaining the FA shown in Figure 3.11*d*.

## EXERCISES

**3.1.** In each case, find a string of minimum length in $\{0, 1\}^*$ *not* in the language corresponding to the given regular expression.

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

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

c. $0^*(100^*)^*1^*$

d. $1^*(0 + 10)^*1^*$

**3.2.** Consider the two regular expressions

$$r = 0^* + 1^* \qquad s = 01^* + 10^* + 1^*0 + (0^*1)^*$$

a. Find a string corresponding to $r$ but not to $s$.

b. Find a string corresponding to $s$ but not to $r$.

c. Find a string corresponding to both $r$ and $s$.

d. Find a string in $\{0, 1\}^*$ corresponding to neither $r$ nor $s$.

**3.3.** Let $r$ and $s$ be arbitrary regular expressions over the alphabet $\Sigma$. In each case, find a simpler regular expression corresponding to the same language as the given one.

a. $(r + s + rs + sr)^*$

b. $(r(r + s)^*)^+$

c. $r(r^*r + r^*) + r^*$

d. $(r + \Lambda)^*$

e. $(r + s)^*rs(r + s)^* + s^*r^*$

**3.4.** Prove the formula

$$(111^*)^* = (11 + 111)^*$$



**3.5.** Prove the formula

$$(aa^*bb^*)^* = \Lambda + a(a + b)^*b$$

**3.6.** In the definition of regular languages, Definition 3.1, statement 2 can be omitted without changing the set of regular languages. Why?

**3.7.** The set of regular languages over $\Sigma$ is the smallest set that contains all the languages $\emptyset$, $\{\Lambda\}$, and $\{a\}$ (for every $a \in \Sigma$) and is closed under the operations of union, concatenation, and Kleene *. In each case below, describe the smallest set of languages that contains all these "basic" languages and is closed under the specified operations.

a. union

b. concatenation

c. Kleene *

d. union and concatenation

e. union and Kleene *

**3.8.** Find regular expressions corresponding to each of the languages defined recursively below.

a. $\Lambda \in L$; if $x \in L$, then $001x$ and $x11$ are elements of $L$; nothing is in $L$ unless it can be obtained from these two statements.

b. $0 \in L$; if $x \in L$, then $001x$, $x001$, and $x11$ are elements of $L$; nothing is in $L$ unless it can be obtained from these two statements.

c. $\Lambda \in L$; $0 \in L$; if $x \in L$, then $001x$ and $11x$ are in $L$; nothing is in $L$ unless it can be obtained from these three statements.

**3.9.** Find a regular expression corresponding to each of the following subsets of $\{0, 1\}^*$.

a. The language of all strings containing exactly two 0's.

b. The language of all strings containing at least two 0's.

c. The language of all strings that do not end with 01.

d. The language of all strings that begin or end with 00 or 11.

e. The language of all strings not containing the substring 00.

f. The language of all strings in which the number of 0's is even.

g. The language of all strings containing no more than one occurrence of the string 00. (The string 000 should be viewed as containing two occurrences of 00.)

h. The language of all strings in which every 0 is followed immediately by 11.

i. The language of all strings containing both 11 and 010 as substrings.

**3.10.** Describe as simply as possible the language corresponding to each of the following regular expressions.

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

b. $((0+1)^3)^*(\Lambda+0+1)$

c. $(1+01)^*(0+01)^*$

d. $(0+1)^*(0^+1^+0^+ + 1^+0^+1^+)(0+1)^*$ (Give an answer of the form: all strings containing both the substring _____ and the substring _____.)

**3.11.** Show that if $L$ is a regular language, then the language $L^n$ is regular for every $n \geq 0$.

**3.12.** Show that every finite language is regular.

**3.13.** The function $rev : \Sigma^* \to \Sigma^*$ is defined in Example 2.24. For a language $L$, let $L^r$ denote the language $\{rev(x) \mid x \in L\} = \{x^r \mid x \in L\}$.

a. If $e$ is the regular expression $(001 + 11010)^*1010$, and $L_e$ is the corresponding language, give a regular expression corresponding to $L_e^r$.

b. Taking this example as a model, give a recursive definition of a function *rrev* from the set of regular expressions over $\Sigma$ to itself, so that for any regular expression $r$ over $\Sigma$, the language corresponding to $rrev(r)$ is the reverse of the language corresponding to $r$. Give a proof that your function has this property.

c. Show that if $L$ is a regular language, then $L^r$ is regular.

**3.14.** In the C programming language, all the following expressions represent valid numerical "literals":

```
3          13.       .328       41.16     +45.80
+0         -01       -14.4      1e12      +1.4e6
-2.e+7     01E-06    0.2E-20    -.4E-7    00e0
```

The letter e or E refers to an exponent, and if it appears, the number following it is an integer. Based on these examples, write a regular expression representing the language of numerical literals. You can use the same shorthand as in Example 3.4: $l$ for "letter," $d$ for "digit," $a$ for '+', $m$ for '−', and $p$ for "point." Assume that there are no limits on the number of consecutive digits in any part of the expression.

**3.15.** The *star height* of a regular expression $r$ over $\Sigma$, denoted by $sh(r)$, is defined as follows:

(i) $sh(\emptyset) = 0$.

(ii) $sh(\Lambda) = 0$.

(iii) $sh(a) = 0$ for every $a \in \Sigma$.

(iv) $sh((rs)) = sh((r+s)) = \max(sh(r), sh(s))$.

(v) $sh((r^*)) = sh(r) + 1$.

Find the star heights of the following regular expressions.

a. $(a(a + a^*aa) + aaa)^*$

b. $(((a + a^*aa)aa)^* + aaaaaa^*)^*$

**3.16.** For both the regular expressions in the previous exercise, find an equivalent regular expression of star height 1.



**3.17.** For each of the FAs pictured in Figure 3.12, describe, either in words or by writing a regular expression, the strings that cause the FA to be in each state.

**3.18.** Let $x$ be a string in $\{0, 1\}^*$ of length $n$. Describe an FA that accepts the string $x$ and no other strings. How many states are required?

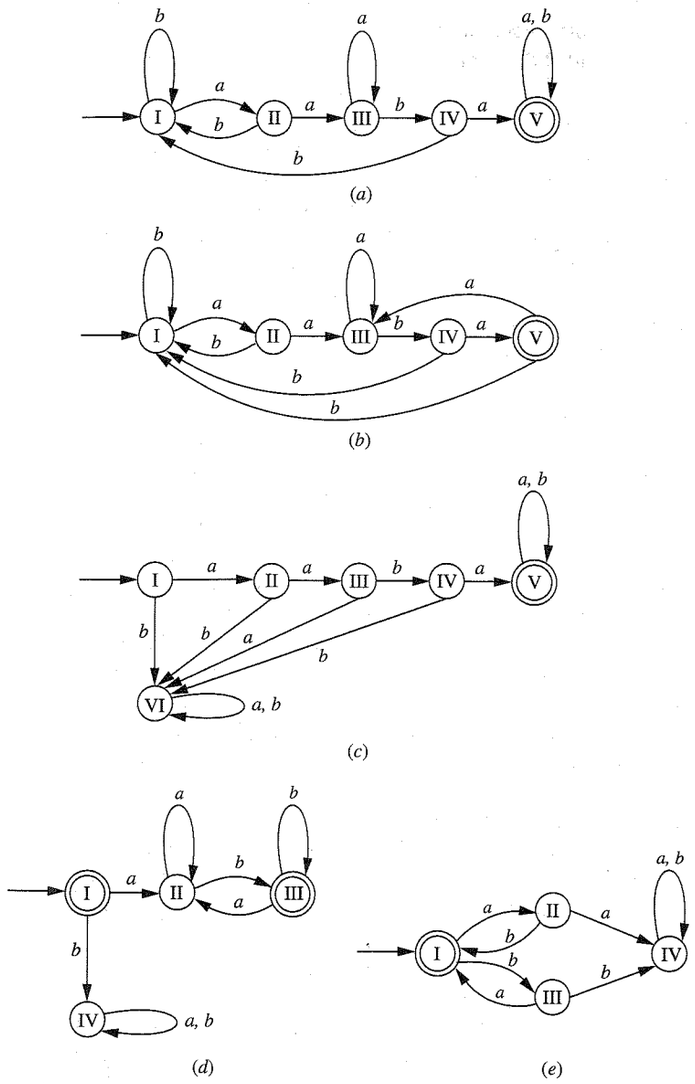



**Figure 3.12 |**

**3.19.** For each of the languages in Exercise 3.9, draw an FA recognizing the language.

**3.20.** For each of the following regular expressions, draw an FA recognizing the corresponding language.

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

b. $(11 + 10)^*$

c. $(0 + 1)^*(1 + 00)(0 + 1)^*$

d. $(111 + 100)^*0$

e. $0 + 10^* + 01^*0$

f. $(0 + 1)^*(01 + 110)$

**3.21.** Draw an FA that recognizes the language of all strings of 0's and 1's of length at least 1 that, if they were interpreted as binary representations of integers, would represent integers evenly divisible by 3. Your FA should accept the string 0 but no other strings with leading 0's.

**3.22.** Suppose $M$ is the finite automaton $(Q, \Sigma, q_0, A, \delta)$.

a. Using mathematical induction or structural induction, show that for any $x$ and $y$ in $\Sigma^*$, and any $q \in Q$,

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

Two reasonable approaches are to base the induction on $x$ and to base it on $y$. One, however, works better than the other.

b. Show that if for some state $q$, $\delta(q, a) = q$ for every $a \in \Sigma$, then $\delta^*(q, x) = q$ for every $x \in \Sigma^*$.

c. Show that if for some state $q$ and some string $x$, $\delta^*(q, x) = q$, then for every $n \geq 0$, $\delta^*(q, x^n) = q$.

**3.23.** If $L$ is a language accepted by an FA $M$, then there is an FA accepting $L$ with more states than $M$ (and therefore there is no limit to the number of states an FA accepting $L$ can have). Explain briefly why this is true.

**3.24.** Show by an example that for some regular language $L$, any FA recognizing $L$ must have more than one accepting state. Characterize those regular languages for which this is true.

**3.25.** For the FA pictured in Figure 3.11*d*, show that there cannot be any other FA with fewer states accepting the same language.

**3.26.** Let $z$ be a fixed string of length $n$ over the alphabet $\{0, 1\}$. What is the smallest number of states an FA can have if it accepts the language $\{0, 1\}^*\{z\}$? Prove your answer.

**3.27.** Suppose $L$ is a subset of $\{0, 1\}^*$. Does an infinite set of distinguishable pairs with respect to $L$ imply an infinite set that is pairwise distinguishable with respect to $L$? In particular, if $x_0, x_1, \ldots$ is a sequence of distinct strings in $\{0, 1\}^*$ so that for any $n \geq 0$, $x_n$ and $x_{n+1}$ are distinguishable with respect to $L$, does it follow that $L$ is not regular? Either give a proof that it does follow, or provide an example of a regular language $L$ and a sequence of strings $x_0$, $x_1, \ldots$ with this property.



**3.28.** Let $L \subseteq \{0, 1\}^*$ be an infinite language, and for each $n \geq 0$, let $L_n = \{x \in L \mid |x| = n\}$. Denote by $s(n)$ the number of states an FA must have in order to accept $L_n$. What is the smallest that $s(n)$ can be if $L_n \neq \emptyset$? Give an example of an infinite language $L$ so that for every $n$ satisfying $L_n \neq \emptyset$, $s(n)$ is this minimum number.

**3.29.** Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA. If $p$ and $q$ are elements of $Q$, we say $q$ is *reachable* from $p$ if there is a string $x \in \Sigma^*$ so that $\delta^*(p, x) = q$. Let $M_1$ be the FA obtained from $M$ by deleting any states that are not reachable from $q_0$ (and any transitions to or from such states). Show that $M_1$ and $M$ recognize the same language. (Note: we might also try to simplify $M$ by eliminating the states from which no accepting state is reachable. The result might be a pair $(q, a) \in Q \times \Sigma$ for which no transition is defined, so that we do not have an FA. In Chapter 4 we will see that this simplification can still be useful.)

**3.30.** Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. Let $M_1 = (Q, \Sigma, q_0, R, \delta)$, where $R$ is the set of states in $Q$ from which some element of $A$ is reachable (see Exercise 3.29). What is the relationship between the language recognized by $M_1$ and the language recognized by $M$? Prove your answer.

**3.31.** Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA, and suppose that there is some string $z$ so that for every $q \in Q$, $\delta^*(q, z) \in A$. What conclusion can you draw, and why?

**3.32.** (For this problem, refer to the proof of Theorem 3.4.) Show that for any $x \in \Sigma^*$ and any $(p, q) \in Q$, $\delta^*((p, q), x) = (\delta_1^*(p, x), \delta_2^*(q, x))$.

**3.33.** Let $M_1$, $M_2$, and $M_3$ be the FAs pictured in Figure 3.13, recognizing languages $L_1$, $L_2$, and $L_3$, respectively.

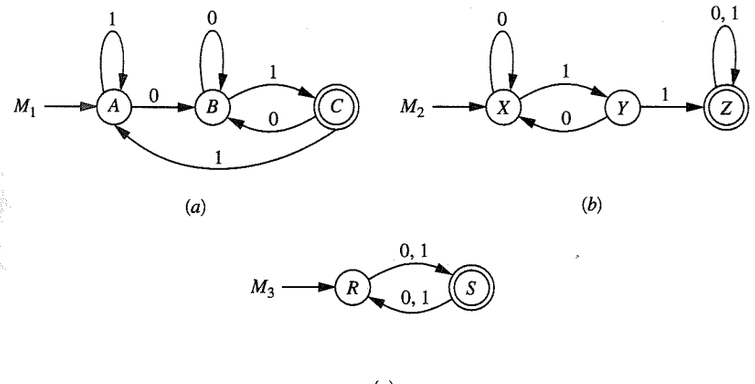



**Figure 3.13 |**

Draw FAs recognizing the following languages.

a. $L_1 \cup L_2$
b. $L_1 \cap L_2$
c. $L_1 - L_2$
d. $L_1 \cap L_3$
e. $L_3 - L_2$

## MORE CHALLENGING PROBLEMS

**3.34.** Prove that for any two regular expressions $r$ and $s$ over $\Sigma$, $(r^*s^*)^* = (r + s)^*$.

**3.35.** Prove the formula

$$(00^*1)^*1 = 1 + 0(0 + 10)^*11$$

**3.36.** Find a regular expression corresponding to each of the following subsets of $\{0, 1\}^*$.

a. The language of all strings not containing the substring 000.
b. The language of all strings that do not contain the substring 110.
c. The language of all strings containing both 101 and 010 as substrings.
d. The language of all strings in which both the number of 0's and the number of 1's are even.
e. The language of all strings in which both the number of 0's and the number of 1's are odd.

**3.37.** Suppose $r$ is a regular expression over $\Sigma$. Show that if for each symbol in $r$ that is an element of $\Sigma$ another regular expression over $\Sigma$ is substituted, the result is a regular expression over $\Sigma$.

**3.38.** Let $c$ and $d$ be regular expressions over $\Sigma$.

a. Show that the formula $r = c + rd$, involving the variable $r$, is true if the regular expression $cd^*$ is substituted for $r$.
b. Show that if $\Lambda$ is not in the language corresponding to $d$, then any regular expression $r$ satisfying $r = c + rd$ corresponds to the same language as $cd^*$.

**3.39.** Describe precisely an algorithm that could be used to eliminate the symbol $\emptyset$ from any regular expression that does not correspond to the empty language.

**3.40.** Describe an algorithm that could be used to eliminate the symbol $\Lambda$ from any regular expression whose corresponding language does not contain the null string.

**3.41.** The *order* of a regular language $L$ is the smallest integer $k$ for which $L^k = L^{k+1}$, if there is one, and $\infty$ otherwise.

a. Show that the order of $L$ is finite if and only if there is an integer $k$ so that $L^k = L^*$, and that in this case the order of $L$ is the smallest $k$ so that $L^k = L^*$.



b. What is the order of the regular language $\{\Lambda\} \cup \{aa\}\{aaa\}^*$?
c. What is the order of the regular language $\{a\} \cup \{aa\}\{aaa\}^*$?
d. What is the order of the language corresponding to the regular expression $(\Lambda + b^*a)(b + ab^*ab^*a)^*$?

**3.42.** A *generalized regular expression* is defined the same way as an ordinary regular expression, except that two additional operations, intersection and complement, are allowed. So, for example, the generalized regular expression $abb\emptyset' \cap (\emptyset'aaa\emptyset')'$ represents the set of all strings in $\{a, b\}^*$ that start with $abb$ and don't contain the substring $aaa$.

a. Show that the subset $\{aba\}^*$ of $\{a, b\}^*$ can be described by a generalized regular expression with no occurrences of $^*$.
b. Can the subset $\{aa\}^*$ be described this way? Give reasons for your answer.

**3.43.** For each of the following regular expressions, draw an FA recognizing the corresponding language.

a. $(1 + 110)^*0$
b. $(1 + 10 + 110)^*0$
c. $1(01 + 10)^* + 0(11 + 10)^*$
d. $1(1 + 10)^* + 10(0 + 01)^*$
e. $(010 + 00)^*(10)^*$

**3.44.** Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. Below are other conceivable methods of defining the extended transition function $\delta^*$ (see Definition 3.3). In each case, determine whether it is in fact a valid definition of a function on the set $Q \times \Sigma^*$, and why. *If* it is, show using mathematical induction that it defines the same function that Definition 3.3 does.

a. For any $q \in Q$, $\delta^*(q, \Lambda) = q$; for any $y \in \Sigma^*$, $a \in \Sigma$, and $q \in Q$, $\delta^*(q, ya) = \delta^*(\delta^*(q, y), a)$.
b. For any $q \in Q$, $\delta^*(q, \Lambda) = q$; for any $y \in \Sigma^*$, $a \in \Sigma$, and $q \in Q$, $\delta^*(q, ay) = \delta^*(\delta(q, a), y)$.
c. For any $q \in Q$, $\delta^*(q, \Lambda) = q$; for any $q \in Q$ and any $a \in \Sigma$, $\delta^*(q, a) = \delta(q, a)$; for any $q \in Q$, and any $x$ and $y$ in $\Sigma^*$, $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$.

**3.45.** Let $L$ be the set of even-length strings over $\{0, 1\}$ whose first and second halves are identical; in other words, $L = \{ww \mid w \in \{0, 1\}^*\}$. Use the technique of Theorem 3.3 to show that $L$ is not regular: Show that for any two strings $x$ and $y$ in $\{0, 1\}^*$, $x$ and $y$ are distinguishable with respect to $L$.

**3.46.** Let $L$ be the language $\{0^n1^n \mid n \geq 0\}$.

a. Find two distinct strings $x$ and $y$ that are indistinguishable with respect to $L$.
b. Show that $L$ is not regular, by showing that there is an infinite set of strings, any two of which are distinguishable with respect to $L$.

**3.47.** As in Example 3.17, let

$$L_n = \{x \in \{0, 1\}^* \mid |x| \geq n \text{ and the } n\text{th symbol from the right in } x \text{ is } 1\}$$

It is shown in that example that any FA recognizing $L_n$ must have at least $2^n$ states. Draw an FA with four states that recognizes $L_2$. For any $n \geq 1$, describe how to construct an FA with $2^n$ states that recognizes $L_n$.

**3.48.** Let $n$ be a positive integer and $L = \{x \in \{0, 1\}^* \mid |x| = n \text{ and } n_0(x) = n_1(x)\}$. What is the minimum number of states in any FA that recognizes $L$? Give reasons for your answer.

**3.49.** Let $n$ be a positive integer and $L = \{x \in \{0, 1\}^* \mid |x| = n \text{ and } n_0(x) < n_1(x)\}$. What is the minimum number of states in any FA that recognizes $L$? Give reasons for your answer.

**3.50.** Let $n$ be a positive integer, and let $L$ be the set of all strings in *pal* of length $2n$. In other words,

$$L = \{xx^r \mid x \in \{0, 1\}^n\}$$

What is the minimum number of states in any FA that recognizes $L$? Give reasons for your answer.

**3.51.** Languages such as that in Example 3.9 are regular for what seems like a particularly simple reason: In order to test a string for membership, we need to examine only the last few symbols. More precisely, there is an integer $n$ and a set $S$ of strings of length $n$ so that for any string $x$ of length $n$ or greater, $x$ is in the language if and only if $x = yz$ for some $z \in S$. (In Example 3.9, we may take $n$ to be 2 and $S$ to be the set $\{11\}$.) Show that any language $L$ having this property is regular.

**3.52.** Show that every finite language has the property in the previous problem.

**3.53.** Give an example of an infinite regular language (a subset of $\{0, 1\}^*$) that does not have the property in Exercise 3.51, and prove that it does not.

**3.54.** (This exercise is due to Hermann Stamm-Wilbrandt.) Consider the language

$$L = \{x \in \{0, 1, 2\}^* \mid x \text{ does not contain the substring } 01\}$$

Show that for every $i \geq 0$, the number of strings of length $i$ in $L$ is $f(2i + 2)$, where $f$ is the Fibonacci function (see Example 2.13). Hint: draw an FA accepting $L$, with initial state $q_0$, and consider for each state $q$ in your FA and each integer $i$, the set $S(q, i)$ of all strings $x$ of length $i$ satisfying $\delta^*(q_0, x) = q$.

**3.55.** Consider the two FAs in Figure 3.14. If you examine them closely you can see that they are really identical, except that the states have different names: State $p$ corresponds to state $A$, $q$ corresponds to $B$, and $r$ corresponds to $C$. Let us describe this correspondence by the *relabeling function* $i$; that is, $i(p) = A, i(q) = B, i(r) = C$. What does it mean to say that under this correspondence, the two FAs are "really identical?" It means several things: First, the initial states correspond to each other; second, a state is an accepting state if and only if the corresponding state is; and finally, the



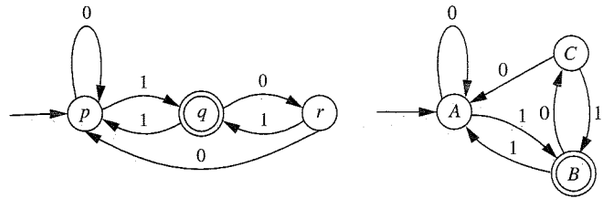



**Figure 3.14 |**

transitions among the states of the first FA are the same as those among the corresponding states of the other. For example, if $\delta_1$ and $\delta_2$ are the transition functions,

$$\delta_1(p, 0) = p \quad \text{and} \quad \delta_2(i(p), 0) = i(p)$$
$$\delta_1(p, 1) = q \quad \text{and} \quad \delta_2(i(p), 1) = i(q)$$

These formulas can be rewritten

$$\delta_2(i(p), 0) = i(\delta_1(p, 0)) \text{ and } \delta_2(i(p), 0) = i(\delta_1(p, 0))$$

and these and all the other relevant formulas can be summarized by the general formula

$$\delta_2(i(s), a) = i(\delta_1(s, a)) \text{ for every state } s \text{ and alphabet symbol } a$$

In general, if $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are FAs and $i : Q_1 \to Q_2$ is a bijection (i.e., one-to-one and onto), we say that $i$ is an *isomorphism from* $M_1$ *to* $M_2$ if these conditions are satisfied:

(i) $i(q_1) = q_2$

(ii) for any $q \in Q_1, i(q) \in A_2$ if and only if $q \in A_1$

(iii) for every $q \in Q_1$ and every $a \in \Sigma, i(\delta_1(q, a)) = \delta_2(i(q), a)$

and we say $M_1$ is *isomorphic to* $M_2$ if there is an isomorphism from $M_1$ to $M_2$. This is simply a precise way of saying that $M_1$ and $M_2$ are "essentially the same."

a. Show that the relation $\sim$ on the set of FAs over $\Sigma$, defined by $M_1 \sim M_2$ if $M_1$ is isomorphic to $M_2$, is an equivalence relation.

b. Show that if $i$ is an isomorphism from $M_1$ to $M_2$ (notation as above), then for every $q \in Q_1$ and $x \in \Sigma^*$,

$$i(\delta_1^*(q, x)) = \delta_2^*(i(q), x)$$

c. Show that two isomorphic FAs accept the same language.

d. How many one-state FAs over the alphabet $\{0, 1\}$ are there, no two of which are isomorphic?

e. How many pairwise nonisomorphic two-state FAs over {0, 1} are there, in which both states are reachable from the initial state and at least one state is accepting?

f. How many distinct languages are accepted by the FAs in the previous part?

g. Show that the FAs described by these two transition tables are isomorphic. The states are 1–6 in the first, A–F in the second; the initial states are 1 and A, respectively; the accepting states in the first FA are 5 and 6, and D and E in the second.

| $q$ | $\delta_1(q,0)$ | $\delta_1(q,1)$ |
|---|---|---|
| 1 | 3 | 5 |
| 2 | 4 | 2 |
| 3 | 1 | 6 |
| 4 | 4 | 3 |
| 5 | 2 | 4 |
| 6 | 3 | 4 |

| $q$ | $\delta_2(q,0)$ | $\delta_2(q,1)$ |
|---|---|---|
| A | B | E |
| B | A | D |
| C | C | B |
| D | B | C |
| E | F | C |
| F | C | F |

h. Specify a reasonable algorithm for determining whether or not two given FAs are isomorphic.

CHAPTER

# 4

# Nondeterminism and Kleene's Theorem

## 4.1 | NONDETERMINISTIC FINITE AUTOMATA

One of our goals in this chapter is to prove that a language is regular if and only if it can be accepted by a finite automaton (Theorem 3.1). However, examples like Example 3.16 suggest that finding a finite automaton (FA) corresponding to a given regular expression can be tedious and unintuitive if we rely only on the techniques we have developed so far, which involve deciding at each step how much information about the input string it is necessary to remember. An alternative approach is to consider a formal device called a nondeterministic finite automaton (NFA), similar to an FA but with the rules relaxed somewhat. Constructing one of these devices to correspond to a given regular expression is often much simpler. Furthermore, it will turn out that NFAs accept exactly the same languages as FAs, and that there is a straightforward procedure for converting an NFA to an equivalent FA. As a result, not only will it be easier in many examples to construct an FA by starting with an NFA, but introducing these more general devices will also help when we get to our proof.

### A Simpler Approach to Accepting {11, 110}*{0} — EXAMPLE 4.1

Figure 4.1*a* shows the FA constructed in Example 3.16, which recognizes the language $L$ corresponding to the regular expression $(11 + 110)^*0$. Now look at the diagram in Figure 4.1*b*. If we concentrate on the resemblance between this device and an FA, and ignore for the moment the ways in which it fails to satisfy the normal FA rules, we can see that it reflects much more clearly the structure of the regular expression.

For any string $x$ in $L$, a path through the diagram corresponding to $x$ can be described as follows. It starts at $q_0$, and for each occurrence of one of the strings 11 or 110 in the portion of $x$ corresponding to $(11 + 110)^*$, it takes the appropriate loop that returns to $q_0$; when the final 0 is encountered, the path moves to the accepting state $q_4$. In the other direction, we can also