CHAPTER 2

Finite Automata and the Languages They Accept

In this chapter, we introduce the first of the models of computation we will study. A *finite automaton* is a model of a particularly simple computing device, which acts as a language acceptor. We will describe how one works and look at examples of languages that can be accepted this way. Although the examples are simple, they illustrate how finite automata can be useful, both in computer science and more generally. We will also see how their limitations prevent them from being general models of computation, and exactly what might keep a language from being accepted by a finite automaton. The simplicity of the finite automaton model makes it possible, not only to characterize in an elegant way the languages that can be accepted, but also to formulate an algorithm for simplifying one of these devices as much as possible.

2.1 FINITE AUTOMATA: EXAMPLES AND DEFINITIONS

In this section we introduce a type of computer that is simple, partly because the output it produces in response to an input string is limited to "yes" or "no", but mostly because of its primitive memory capabilities during a computation.

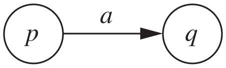
Any computer whose outputs are either "yes" or "no" acts as a *language acceptor*; the language the computer accepts is the set of input strings that cause it to produce the answer yes. In this chapter, instead of thinking of the computer as receiving an input string and then producing an answer, it's a little easier to think of it as receiving individual input symbols, one at a time, and producing after every one the answer for the *current string* of symbols that have been read so far. Before the computer has received any input symbols, the current string is Λ, and there is an answer for that string too. If the current string is *abbab*, for example, the computer might have produced the answers "yes, no, yes, yes, no, no" so far, one for each of the six prefixes of *abbab*.

The *very* simplest device for accepting a language is one whose response doesn't even depend on the input symbols it receives. There are only two possibilities: to announce at each step that the current string is accepted, and to announce at each step that it is not accepted. These two language acceptors are easy to construct, because they don't have to remember anything about the input symbols they have received, but of course they are not very useful. The only languages they can accept are the entire set Σ^* and the empty language $\emptyset$.

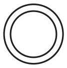
Slightly more complicated is the case in which the answer depends on the last input symbol received and not on any symbols before that. For example, if $\Sigma = \{a, b\}$, a device might announce every time it receives the symbol a, and only in that case, that the current string is accepted. In this case, the language it accepts is the set of all strings that end with a.

These are examples of a type of language acceptor called a *finite automaton* (FA), or *finite-state machine*. At each step, a finite automaton is in one of a finite number of *states* (it is a *finite* automaton because its set of states is finite). Its response depends only on the current state and the current symbol. A "response" to being in a certain state and receiving a certain input symbol is simply to enter a certain state, possibly the same one it was already in. The FA "announces" acceptance or rejection in the sense that its current state is either an *accepting* state or a *nonaccepting* state. In the two trivial examples where the response is always the same, only one state is needed. For the language of strings ending with a, two states are sufficient, an accepting state for the strings ending with a and a nonaccepting state for all the others.

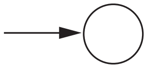
Before a finite automaton has received any input, it is in its initial state, which is an accepting state precisely if the null string is accepted. Once we know how many states there are, which one is the initial state, and which ones are the accepting states, the only other information we need in order to describe the operation of the machine is the *transition function*, which specifies for each combination of state and input symbol the state the FA enters. The transition function can be described by either a table of values or (the way we will use most often) a transition diagram. In the diagram, states are represented by circles, transitions are represented by arrows with input symbols as labels,



and accepting states are designated by double instead of single circles.



The initial state will have an arrow pointing to it that doesn't come from another state.



An FA can proceed through the input, automaton-like, by just remembering at each step what state it's in and changing states in response to input symbols in accordance with the transition function. With the diagram, we can trace the computation for a particular input string by simply following the arrows that correspond to the symbols of the string.

A Finite Automaton Accepting the Language of Strings Ending in *aa* EXAMPLE 2.1

In order to accept the language

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

an FA can operate with three states, corresponding to the number of consecutive a's that must still be received next in order to produce a string in L_1: two, because the current string does not end with a; one, because the current string ends in a but not in aa; or none, because the current string is already in L_1. It is easy to see that a transition diagram can be drawn as in Figure 2.2.

In state q_0, the input symbol b doesn't represent any progress toward obtaining a string in L_1, and it causes the finite automaton to stay in q_0; input a allows it to go to q_1. In q_1, the input b undoes whatever progress we had made and takes us back to q_0, while an a gives us a string in L_1. In q_2, the accepting state, input a allows us to stay in q_2, because the last two symbols of the current string are still both a, and b sends us back to the initial state q_0.

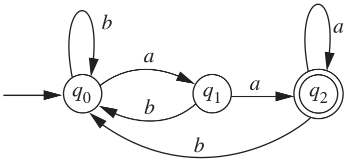


Figure 2.2 | An FA accepting the strings ending with aa.

An FA Accepting the Language of Strings Ending in *b* and Not Containing the Substring *aa* EXAMPLE 2.3

Let L_2 be the language

$$L_2 = \{x \in \{a, b\}^* \mid x \text{ ends with } b \text{ and does not contain the substring } aa\}$$

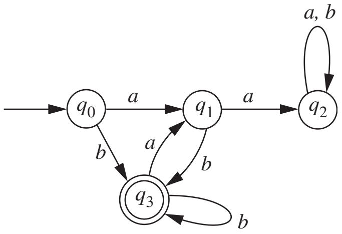


Figure 2.4 |
An FA accepting the strings ending with b and not containing aa.

In Example 2.1, no matter what the current string is, if the next two input symbols are both a, the FA ends up in an accepting state. In accepting L_2, if the next two input symbols are a's, not only do we want to end up in a nonaccepting state, but we want to make sure that from that nonaccepting state we can never reach an accepting state. We can copy the previous example by having three states q_0, q_1, and q_2, in which the current strings don't end in a, end in exactly one a, and end in two a's, respectively. This time all three are nonaccepting states, and from q_2 both transitions return to q_2, so that once our FA reaches this state it will never return to an accepting state.

The only other state we need is an accepting state q_3. Once the FA reaches this state, by receiving the symbol b in either q_0 or q_1, it stays there as long as it continues to receive b's and moves to q_1 on input a. The transition diagram for this FA is shown in Figure 2.4.

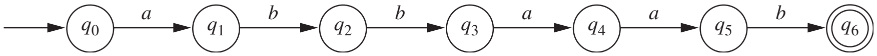
EXAMPLE 2.5 An FA Illustrating a String Search Algorithm

Suppose we have a set of strings over $\{a, b\}$ and we want to identify all the ones containing a particular substring, say $abbaab$. A reasonable way to go about it is to build an FA accepting the language

$$L_3 = \{x \in \{a, b\}^* \mid x \text{ contains the substring } abbaab\}$$

and use it to test each string in the set. Once we have the FA, the number of steps required to test each string is no more than the number of symbols in the string, so that we can be confident this is an efficient approach.

The idea behind the FA is the same as in Example 2.1, but using a string with both a's and b's will make it easier to identify the underlying principle. We can start by drawing this diagram:



For each i, the FA will be in state q_i whenever the current string ends with the prefix of $abbaab$ having length i and not with any longer prefix. Now we just have to try to add transitions to complete the diagram. The transitions leaving q_6 simply return to q_6, because this FA should accept the strings containing, not ending with, $abbaab$. For each $i < 6$, we already have one transition from q_i, and we have to decide where to send the other one.

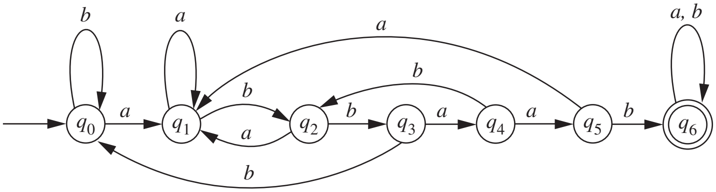


Figure 2.6 |
An FA accepting the strings containing the substring *abbaab*.

Consider $i = 4$. One string that causes the FA to be in state q_4 is *abba*, and we must decide what state corresponds to the string *abbab*. Because *abbab* ends with *ab*, and not with any longer prefix of *abbaab*, the transition should go to q_2. The other cases are similar, and the resulting FA is shown in Figure 2.6.

If we want an FA accepting all the strings ending in *abbaab*, instead of all the strings containing this substring, we can use the transition diagram in Figure 2.6 with different transitions from the accepting state. The transition on input a should go from state q_6 to some earlier state corresponding to a prefix of *abbaab*. Which prefix? The answer is a, the longest one that is a suffix of *abbaaba*. In other words, we can proceed as if we were drawing the FA accepting the set of strings containing *abbaabb* and after six symbols we received input a instead of b. Similarly, the transition from q_6 on input b should go to state q_3.

An FA Accepting Binary Representations of Integers Divisible by 3

EXAMPLE 2.7

We consider the language L of strings over the alphabet {0, 1} that are the binary representations of natural numbers divisible by 3. Another way of saying that n is divisible by 3 is to say that n mod 3, the remainder when n is divided by 3, is 0. This seems to suggest that the only information concerning the current string x that we really need to remember is the remainder when the number represented by x is divided by 3.

The question is, if we know the remainder when the number represented by x is divided by 3, is that enough to find the remainders when the numbers represented by $x0$ and $x1$ are divided by 3? And that raises the question: What are the numbers represented by $x0$ and $x1$?

Just as adding 0 to the end of a decimal representation corresponds to multiplying by ten, adding 0 to the end of a binary representation corresponds to multiplying by 2. Just as adding 1 to the end of a decimal representation corresponds to multiplying by ten and then adding 1 (example: $39011 = 10 * 3901 + 1$), adding 1 to the end of a binary representation corresponds to multiplying by 2 and then adding 1 (example: 1110 represents 14, and 11101 represents 29).

Now we are ready to answer the first question: If x represents n, and n mod 3 is r, then what are $2n$ mod 3 and $(2n + 1)$ mod 3? It is almost correct that the answers are $2r$ and $2r + 1$; the only problem is that these numbers may be 3 or bigger, and in that case we must do another mod 3 operation.

These facts are enough for us to construct our FA. We begin with states corresponding to remainders 0, 1, and 2. The only one of these that is an accepting state is 0, because remainder 0 means that the integer is divisible by 3, and the transitions from these states

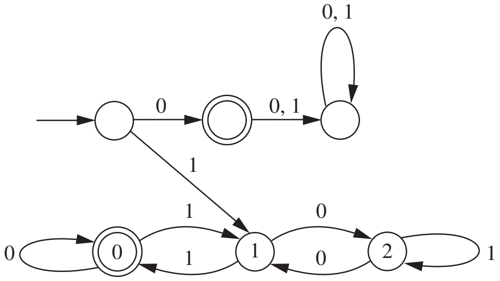


Figure 2.8 |
An FA accepting binary representations of integers divisible by 3.

follow the rules outlined above. These states do not include the initial state, because the null string doesn't qualify as a binary representation of a natural number, or the accepting state corresponding to the string 0. We will disallow leading 0's in binary representations, except for the number 0 itself, and so we need one more state for the strings that start with 0 and have more than one digit. The resulting transition diagram is shown in Figure 2.8.

EXAMPLE 2.9 Lexical Analysis

Another real-world problem for which finite automata are ideally suited is lexical analysis, the first step in compiling a program written in a high-level language.

Before a C compiler can begin to determine whether a string such as

```
main(){ double b=41.3; b *= 4; ...
```

satisfies the many rules for the syntax of C, it must be able to break up the string into *tokens*, which are the indecomposable units of the program. Tokens include reserved words (in this example, "main" and "double"), punctuation symbols, identifiers, operators, various types of parentheses and brackets, numeric literals such as "41.3" and "4", and a few others.

Programming languages differ in their sets of reserved words, as well as in their rules for other kinds of tokens. For example, "41." is a legal token in C but not in Pascal, which requires a numeric literal to have at least one digit on both sides of a decimal point.

In any particular language, the rules for constructing tokens are reasonably simple. Testing a substring to see whether it represents a valid token can be done by a finite automaton in software form; once this is done, the string of alphabet symbols can be replaced by a sequence of tokens, each one represented in a form that is easier for the compiler to use in its later processing.

We will illustrate a lexical-analysis FA for a C-like language in a greatly simplified case, in which the only tokens are identifiers, semicolons, the assignment operator =, the reserved word aa, and numeric literals consisting of one or more digits and possibly a decimal point. An identifier will start with a lowercase letter and will contain only lowercase letters and/or digits. Saying that aa is reserved means that it cannot be an identifier, although longer identifiers might begin with aa or contain it as a substring. The job of the FA will be to accept strings that consist of one or more consecutive tokens. (The FA will not be required to determine whether a particular sequence of tokens makes sense; that job will have to be performed at a later stage in the compilation.) The FA will be in an accepting state each time it finishes reading another legal token, and the state will be one that is reserved for a particular type of token; in this sense, the lexical analyzer will be able to classify the tokens.

Another way to simplify the transition diagram considerably is to make another assumption: that two consecutive tokens are always separated by a blank space.

The crucial transitions of the FA are shown in Figure 2.10. The input alphabet is the set containing the 26 lowercase letters, the 10 digits, a semicolon, an equals sign, a decimal point, and the blank space Δ. We have used a few abbreviations: D for any numerical digit, L for any lowercase letter other than a, M for any numerical digit or letter other than a, and N for any letter or digit. You can check that all possible transitions from the initial state are shown. From every other state, transitions not shown go to a "reject" state, from which all transitions return to that state; no attempt is made to continue the lexical analysis once an error is detected.

The two portions of the diagram that require a little care are the ones involving tokens with more than one symbol. State 3 corresponds to the identifier a, state 4 to the reserved word aa, and state 5 to any other identifier. Transitions to state 5 are possible from state 3 with any letter or digit except a, from states 4 or 5 with any letter or digit, and from the initial state with any letter other than a. State 6 corresponds to numeric literals without decimal points and state 7 to those with decimal points. State 8 is not an accepting state, because a numeric literal must have at least one digit.

This FA could be incorporated into lexical-analysis software as follows. Each time a symbol causes the FA to make a transition out of the initial state, we mark that symbol in the string; each time we are in one of the accepting states and receive a blank space, we mark

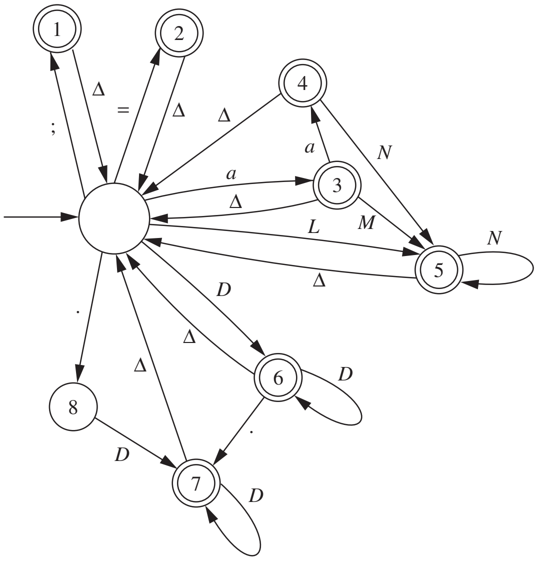


Figure 2.10 |
An FA illustrating a simplified version of lexical analysis.

the symbol just before the blank; and the token, whose type is identified by the accepting state, is represented by the substring that starts with the first of these two symbols and ends with the second.

The restriction that tokens be separated by blanks makes the job of recognizing the beginnings and ends of tokens very simple, but in practice there is no such rule, and we could construct an FA without it. The transition diagram would be considerably more cluttered; the FA would not be in the initial state at the beginning of each token, and many more transitions between the other states would be needed.

Without a blank space to tell us where a token ends, we would normally adopt the convention that each token extends as far as possible. A substring like "`=2b3aa1`" would then be interpreted as containing two tokens, "`=`" and "`2`", and at least the first five symbols of a third. There are substrings, such as "`3...2`", that cannot be part of any legal string. There are others, like "`1.2..3`", that can but only if the extending-as-far-as-possible policy is abandoned. Rejecting this particular string is probably acceptable anyway, because no way of breaking it into tokens is compatible with the syntax of C.

See Example 3.5 for more discussion of tokens and lexical analysis.

Giving the following official definition of a finite automaton and developing some related notation will make it easier to talk about these devices precisely.

Definition 2.11 A Finite Automaton

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

Q is a finite set of *states*;
Σ is a finite *input alphabet*;
$q_0 \in Q$ is the *initial* state;
$A \subseteq Q$ is the set of *accepting* states;
$\delta : Q \times \Sigma \to Q$ is the *transition* function.

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

The first line of the definition deserves a comment. What does it mean to say that a simple computer is a 5-tuple? This is simply a formalism that allows us to define an FA in a concise way. Describing a finite automaton precisely requires us to specify five things, and it is easier to say

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

than it is to say

Let M be an FA with state set Q, input alphabet Σ, initial state q_0, set of accepting states A, and transition function δ.

We write $\delta(q, \sigma)$ to mean the state an FA M goes to from q after receiving the input symbol σ. The next step is to extend the notation to allow a corresponding expression $\delta^*(q, x)$ that will represent the state the FA ends up in if it starts out in state q and receives the string x of input symbols. In other words, we want to define an "extended transition function" δ^* from $Q \times \Sigma^*$ to Q. The easiest way to define it is recursively, using the recursive definition of Σ^* in Example 1.17. We begin by defining $\delta^*(q, \Lambda)$, and since we don't expect the state of M to change as a result of getting the input string Λ, we give the expression the value q.

Definition 2.12 The Extended Transition Function δ^*

Let $M = (Q, \Sigma, q_0, A, \delta)$ be a finite automaton. We define the extended transition function

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

as follows:

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

$$\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma)$$

The recursive part of the definition says that we can evaluate $\delta^*(q, x)$ if we know that $x = y\sigma$, for some string y and some symbol σ, and if we know what state the FA is in after starting in q and processing the symbols of y. We do it by just starting in that state and applying one last transition, the one corresponding to the symbol σ. For example, if M contains the transitions

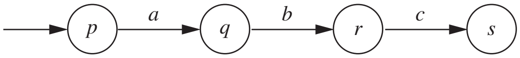


Figure 2.13 |

then

$$\begin{aligned}
\delta^*(p, abc) &= \delta(\delta^*(p, ab), c) \\
&= \delta(\delta(\delta^*(p, a), b), c) \\
&= \delta(\delta(\delta^*(p, \Lambda a), b), c) \\
&= \delta(\delta(\delta(\delta^*(p, \Lambda), a), b), c) \\
&= \delta(\delta(\delta(p, a), b), c) \\
&= \delta(\delta(q, b), c) \\
&= \delta(r, c) \\
&= s
\end{aligned}$$

Looking at the diagram, of course, we can get the same answer by just following the arrows. The point of this derivation is not that it's always the simplest way to find the answer by hand, but that the recursive definition is a reasonable way of defining the extended transition function, and that the definition provides a systematic algorithm.

Other properties you would expect δ^* to satisfy can be derived from our definition. For example, a natural generalization of the recursive statement in the definition is the formula

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

which is true for every state q and every two strings x and y in Σ^*. The proof is by structural induction on y and is similar to the proof of the formula $r(xy) = r(y)r(x)$ in Example 1.27.

The extended transition function makes it possible to say concisely what it means for an FA to accept a string or a language.

Definition 2.14 Acceptance by a Finite Automaton

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA, and let $x \in \Sigma^*$. The string x is *accepted* by M if

$$\delta^*(q_0, x) \in A$$

and is *rejected* by M otherwise. The *language* accepted by M is the set

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

If L is a language over Σ, L is accepted by M if and only if $L = L(M)$.

Notice what the last statement in Definition 2.14 does not say. It doesn't say that L is accepted by M if every string in L is accepted by M. To take this as the definition would not be useful (it's easy to describe a one-state FA that accepts every string in Σ^*). A finite automaton accepting a language L does its job by distinguishing between strings in L and strings not in L: accepting the strings in L and rejecting all the others.

2.2 ACCEPTING THE UNION, INTERSECTION, OR DIFFERENCE OF TWO LANGUAGES

Suppose L_1 and L_2 are both languages over Σ. If $x \in \Sigma^*$, then knowing whether $x \in L_1$ and whether $x \in L_2$ is enough to determine whether $x \in L_1 \cup L_2$. This means that if we have one algorithm to accept L_1 and another to accept L_2, we can easily formulate an algorithm to accept $L_1 \cup L_2$. In this section we will show that if we actually have finite automata accepting L_1 and L_2, then there is a finite automaton accepting $L_1 \cup L_2$, and that the same approach also gives us FAs accepting $L_1 \cap L_2$ and $L_1 - L_2$.

The idea is to construct an FA that effectively executes the two original ones simultaneously, one whose current state records the current states of both. This isn't difficult; we can simply use "states" that are ordered pairs (p, q), where p and q are states in the two original FAs.

Theorem 2.15

Suppose $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are finite automata accepting L_1 and L_2, respectively. Let M be the FA $(Q, \Sigma, q_0, A, \delta)$, where

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

and the transition function δ is defined by the formula

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

for every $p \in Q_1$, every $q \in Q_2$, and every $\sigma \in \Sigma$. 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 consider statement 1, and the other two are similar. The way the transition function δ is defined allows us to say that at any point during the operation of M, if (p, q) is the current state, then p and q are the current states of M_1 and M_2, respectively. This will follow immediately from the formula

$$\delta^*(q_0, x) = (\delta_1^*(q_1, x), \delta_2^*(q_2, x))$$

which is true for every $x \in \Sigma^*$. The proof is by structural induction on x and is left to Exercise 2.12. For every string x, x is accepted by M precisely if $\delta^*(q_0, x) \in A$, and according to the definition of A in statement 1 and the formula for δ^*, this is true precisely if $\delta_1^*(q_1, x) \in A_1$ or $\delta_2^*(q_2, x) \in A_2$—i.e., precisely if $x \in L_1 \cup L_2$.

As we will see in Example 2.16, we don't always need to include every ordered pair in the state set of the composite FA.

Constructing an FA Accepting $L_1 \cap L_2$ **EXAMPLE 2.16**

Let L_1 and L_2 be the languages

$$L_1 = \{x \in \{a, b\}^* \mid aa \text{ is not a substring of } x\}$$
$$L_2 = \{x \in \{a, b\}^* \mid x \text{ ends with } ab\}$$

Finite automata M_1 and M_2 accepting these languages are easy to obtain and are shown in Figure 2.17a. The construction in Theorem 2.15 produces an FA with the nine states shown in Figure 2.17b. Rather than drawing all eighteen transitions, we start at the initial state (A, P), draw the two transitions to (B, Q) and (A, P) using the definition of δ in the theorem, and continue in this way, at each step drawing transitions from a state that has already been reached by some other transition. At some point, we have six states such that every transition from one of these six goes to one of these six. Since none of the remaining three states is reachable from any of the first six, we can leave them out.

If we want our finite automaton to accept $L_1 \cup L_2$, then we designate as accepting states the ordered pairs among the remaining six that involve at least one of the accepting states A, B, and R. The result is shown in Figure 2.17c.

If instead we want to accept $L_1 \cap L_2$, then the only accepting state is (A, R), since (B, R) was one of the three omitted. This allows us to simplify the FA even further. None of the three states (C, P), (C, Q), and (C, R) is accepting, and every transition from one of these three goes to one of these three; therefore, we can combine them all into a single nonaccepting state. An FA accepting $L_1 \cap L_2$ is shown in Figure 2.17d. The FA we would get for $L_1 - L_2$ is similar and also has just four states, but in that case two are accepting states.

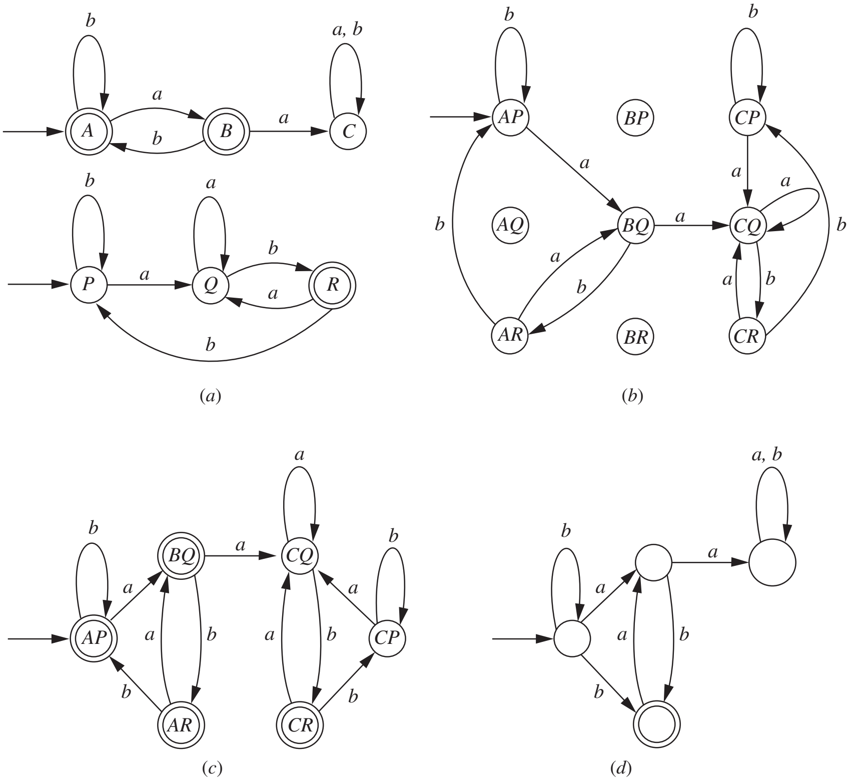


Figure 2.17 |
Constructing an FA to accept the intersection of two languages.

An FA Accepting Strings That Contain Either *ab* or *bba* EXAMPLE 2.18

Figure 2.19a shows FAs M_1 and M_2 accepting $L_1 = \{x \in \{a, b\}^* \mid x \text{ contains the substring } ab\}$ and $L_2 = \{x \in \{a, b\}^* \mid x \text{ contains the substring } bba\}$, respectively. They are both obtained by the technique described in Example 2.5. Using Theorem 2.15 to construct an FA accepting $L_1 \cup L_2$ could result in one with twelve states, but Figure 2.19b illustrates an approach that seems likely to require considerably fewer. If it works, the FA will need only the states we've drawn, and the two paths to the accepting state will correspond to strings in L_1 and strings in L_2, respectively.

This approach does work; the five states shown are sufficient, and it is not difficult to complete the transitions from the three intermediate states. Instead, let's see whether the construction in the theorem gives us the same answer or one more complicated. Figure 2.19c shows a partially completed diagram; to complete it, we must draw the transitions from $(3, q)$ and $(2, s)$ and any additional states that may be required. So far we have six states, and you can check that $(3, p)$, $(3, r)$, and $(3, s)$ will also appear if we continue this way. Notice, however, that because states 3 and s are accepting, and transitions from either state return to that state, every state we add will be an ordered pair involving 3 or s or both, and every transition from one of these accepting states will return to one. The conclusion is that we can combine $(3, q)$ and $(2, s)$ and we don't need any more states; the final diagram is in Figure 2.19d.

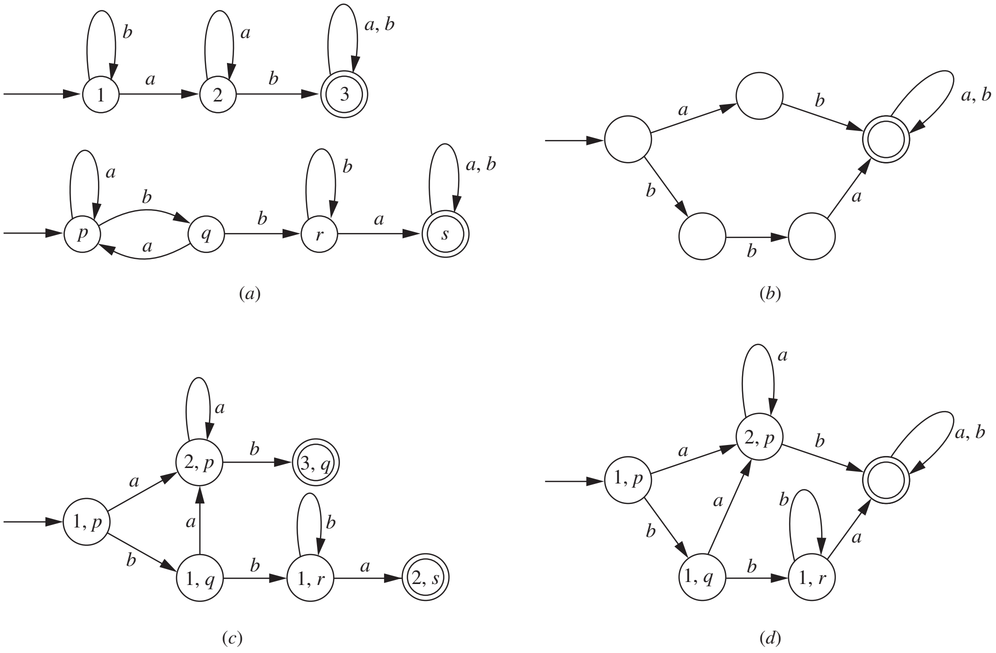


Figure 2.19 |
Constructing an FA to accept strings containing either *ab* or *bba*.

The construction in Theorem 2.15 will always work, but the FA it produces may not be the simplest possible. Fortunately, if we need the simplest possible one, we don't need to rely on the somewhat unsystematic methods in these two examples; we will see in Section 2.6 how to start with an arbitary FA and find one with the fewest possible states accepting the same language.

2.3 DISTINGUISHING ONE STRING FROM ANOTHER

The finite automaton M in Example 2.1, accepting the language L of strings in $\{a, b\}^*$ ending with aa, had three states, corresponding to the three possible numbers of a's still needed to have a string in L. As simple as this sounds, it's worth taking a closer look. Could the FA be constructed with fewer than three states? And can we be sure that three are enough? These are different questions; we will answer the first in this section and return to the second in Section 2.5.

Any FA with three states ignores, or forgets, almost all the information pertaining to the current string. In the case of M, it makes no difference whether the current string is aba or $aabbabbabaaaba$; the only relevant feature of these two strings is that both end with a and neither ends with aa. However, it does make a difference whether the current string is aba or ab, even though neither string is in L. It makes a difference because of what input symbols might come next. If the next input symbol is a, the current string at that point would be $abaa$ in the first case and aba in the second; one string is in L and the other isn't. The FA has to be able to distinguish aba and ab now, so that in case the next input symbol is a it will be able to distinguish the two corresponding longer strings. We will describe the difference between aba and ab by saying that they are *distinguishable* with respect to L: there is at least one string z such that of the two strings $abaz$ and abz, one is in L and the other isn't.

Definition 2.20 Strings Distinguishable with Respect to *L*

If L is a language over the alphabet Σ, and x and y are strings in Σ^*, then x and y are *distinguishable with respect to* L, or L*-distinguishable*, if there is a string $z \in \Sigma^*$ such that either $xz \in L$ and $yz \notin L$, or $xz \notin L$ and $yz \in L$. A string z having this property is said to distinguish x and y with respect to L. An equivalent formulation is to say that x and y are L-distinguishable if $L/x \neq L/y$, where

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

The two strings x and y are L-indistinguishable if $L/x = L/y$, which means that for every $z \in \Sigma^*$, $xz \in L$ if and only if $yz \in L$.

The strings in a set $S \subseteq \Sigma^*$ are *pairwise* L*-distinguishable* if for every pair x, y of distinct strings in S, x and y are L-distinguishable.

The crucial fact about two L-distinguishable strings, or more generally about a set of pairwise L-distinguishable strings, is given in Theorem 2.21, and it will provide the answer to the first question we asked in the first paragraph.

Theorem 2.21

Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting the language $L \subseteq \Sigma^*$. If x and y are two strings in Σ^* that are L-distinguishable, then $\delta^*(q_0, x) \neq \delta^*(q_0, y)$. For every $n \geq 2$, if there is a set of n pairwise L-distinguishable strings in Σ^*, then Q must contain at least n states.

Proof

If x and y are L-distinguishable, then for some string z, one of the strings xz, yz is in L and the other isn't. Because M accepts L, this means that one of the states $\delta^*(q_0, xz)$, $\delta^*(q_0, yz)$ is an accepting state and the other isn't. In particular,

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

According to Exercise 2.5, however,

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

Because the left sides are different, the right sides must be also, and so $\delta^*(q_0, x) \neq \delta^*(q_0, y)$.

The second statement in the theorem follows from the first: If M had fewer than n states, then at least two of the n strings would cause M to end up in the same state, but this is impossible if the two strings are L-distinguishable.

Returning to our example, we can now say why there must be three states in an FA accepting L, the language of strings ending with aa. We already have an FA with three states accepting L. Three states are actually necessary if there are three pairwise L-distinguishable strings, and we can find three such strings by choosing one corresponding to each state. We choose Λ, a, and aa. The string a distinguishes Λ and a, because $\Lambda a \notin L$ and $aa \in L$; the string Λ distinguishes Λ and aa; and it also distinguishes a and aa.

As the next example illustrates, the first statement in Theorem 2.21 can be useful in constructing a finite automaton to accept a language, because it can help us decide at each step whether a transition should go to a state we already have or whether we need to add another state.

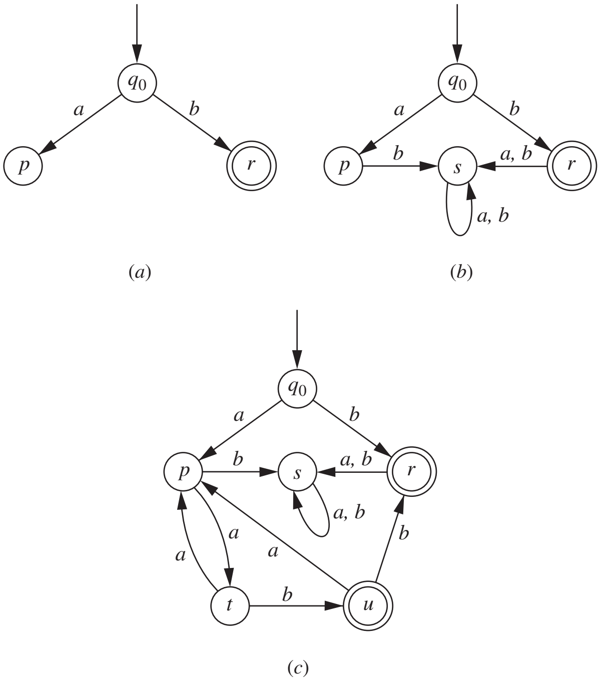
Constructing an FA to Accept $\{aa, aab\}^*\{b\}$ EXAMPLE 2.22

Let L be the language $\{aa, aab\}^*\{b\}$. In Chapter 3 we will study a systematic way to construct finite automata for languages like this one. It may not be obvious at this stage that it will even be possible, but we will proceed by adding states as needed and hope that we will eventually have enough.

The null string is not in L, and so the initial state should not be an accepting state. The string b is in L, the string a is not, and the two strings Λ and a are L-distinguishable because $\Lambda ab \notin L$ and $aab \in L$. We have therefore determined that we need at least the states in Figure 2.23a.

The language L contains b but no other strings beginning with b. It also contains no strings beginning with ab. These two observations suggest that we introduce a state s to take care of all strings that fail for either reason to be a prefix of an element of L (Fig.2.23b). Notice that if two strings are L-distinguishable, at least one of them must be a prefix of an element of L; therefore, two strings ending up in state s cannot be L-distinguishable. All transitions from s will return to s.

Suppose the FA is in state p and receives the input a. It can't stay in p, because the strings a and aa are distinguished relative to L by the string ab. It can't return to the initial state, because Λ and aa are L-distinguishable. Therefore, we need a new state t. From t, the input b must lead to an accepting state, because $aab \in L$; this accepting state cannot be r, because aab and a are L-distinguishable; call the new accepting state u. One of the strings that gets the FA to state u is aab. If we receive another b in state u, the situation is the same as for an initial b; $aabb$ and b are both in L, but neither is a prefix of any longer string in L. We can therefore let $\delta(u, b) = r$.



(a)

(b)

(c)

Figure 2.23 |
Constructing an FA to accept $\{aa, aab\}^*\{b\}$.

We have yet to define $\delta(t, a)$ and $\delta(u, a)$. States t and u can be thought of as representing the end of one of the strings aa and aab. (The reason u is an accepting state is that one of these two strings, aab, also happens to be the other one followed by b.) In either case, if the next symbol is a, we should view it as the first symbol in *another* occurrence of one of these two strings. For this reason, we can define $\delta(t, a) = \delta(u, a) = p$, and we arrive at the FA shown in Figure 2.23c.

It may be clear already that because we added each state only if necessary, the FA we have constructed is the one with the fewest possible states. If we had not constructed it ourselves, we could use Theorem 2.21 to show this. The FA has six states, and applying the second statement in the theorem seems to require that we produce six pairwise L-distinguishable strings. Coming up with six strings is easy—we can choose one corresponding to each state—but verifying directly that they are pairwise L-distinguishable requires looking at all 21 choices of two of them. A slightly easier approach, since there are four nonaccepting states and two accepting states, is to show that the four strings that are not in L are pairwise L-distinguishable and the two strings in L are L-distinguishable. The argument in the proof of the theorem can easily be adapted to show that every FA accepting L must then have at least four nonaccepting states and two accepting states. This way we have to consider only seven sets of two strings and for each set find a string that distinguishes the two relative to L.

An FA Accepting Strings with *a* in the *n*th Symbol from the End EXAMPLE 2.24

Suppose n is a positive integer, and L_n is the language of strings in $\{a, b\}^*$ with at least n symbols and an a in the nth position from the end.

The first observation about accepting this language is that if a finite automaton "remembers" the most recent n input symbols it has received, or remembers the entire current string as long as its length is less than n, then it has enough information to continue making correct decisions. Another way to say this is that no symbol that was received more than n symbols ago should play any part in deciding whether the current string is accepted.

We can also see that if $i < n$ and x is any string of length i, then the string $b^{n-i}x$ can be treated the same as x by an FA accepting L_n. For example, suppose $n = 5$ and $x = baa$. Neither of the strings $bbbaa$ and baa is in L_5, three more symbols are required in both cases before an element of L_5 will be obtained, and from that point on the two current strings will always agree in the last five symbols. As a result, an FA accepting L_n will require no more than 2^n states, the number of strings of length n.

Finally, if we can show that the strings of length n are pairwise L_n-distinguishable, Theorem 2.21 will tell us that we need this many states. Let x and y be distinct strings of length n. They must differ in the ith symbol (from the left), for some i with $1 \le i \le n$. Every string z of length $i - 1$ distinguishes these two relative to L_n, because xz and yz differ in the ith symbol, which is the nth symbol from the right.

Figure 2.25 shows an FA with four states accepting L_2. The labeling technique we have used here also works for $n > 2$; if each state is identified with a string x of length n, and $x = \sigma_1 y$ where $|y| = n - 1$, the transition function can be described by the formula

$$\delta(\sigma_1 y, \sigma) = y\sigma$$

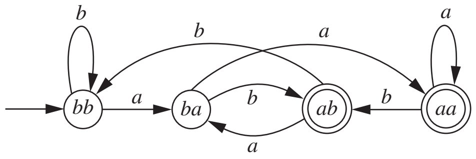


Figure 2.25 |
An FA accepting L_n in the case $n = 2$.

We can carry the second statement of Theorem 2.21 one step further, by considering a language L for which there are infinitely many pairwise L-distinguishable strings.

Theorem 2.26
For every language $L \subseteq \Sigma^*$, if there is an infinite set S of pairwise L-distinguishable strings, then L cannot be accepted by a finite automaton.

Proof
If S is infinite, then for every n, S has a subset with n elements. If M were a finite automaton accepting L, then Theorem 2.21 would say that for every n, M would have at least n states. No *finite* automaton can have this property!

It is not hard to find languages with the property in Theorem 2.26. In Example 2.27 we take L to be the language *Pal* from Example 1.18, the set of palindromes over $\{a,b\}$. Not only is there an infinite set of pairwise L-distinguishable strings, but *all* the strings in $\{a,b\}^*$ are pairwise L-distinguishable.

EXAMPLE 2.27 For Every Pair x, y of Distinct Strings in $\{a,b\}^*$, x and y Are Distinguishable with Respect to *Pal*

First suppose that $x \neq y$ and $|x| = |y|$. Then x^r, the reverse of x, distinguishes the two with respect to *Pal*, because $xx^r \in Pal$ and $yx^r \notin Pal$. If $|x| \neq |y|$, we assume x is shorter. If x is not a prefix of y, then $xx^r \in Pal$ and $yx^r \notin Pal$. If x is a prefix of y, then $y = xz$ for some nonnull string z. If we choose the symbol σ (either a or b) so that $z\sigma$ is not a palindrome, then $x\sigma x^r \in Pal$ and $y\sigma x^r = xz\sigma x^r \notin Pal$.

An explanation for this property of *Pal* is easy to find. If a computer is trying to accept *Pal*, has read the string x, and starts to receive the symbols of another string z, it can't be expected to decide whether z is the reverse of x unless it can actually remember every symbol of x. The only thing a finite automaton M can remember is what state it's in, and there are only a finite number of states. If x is a sufficiently long string, remembering every symbol of x is too much to expect of M.

2.4 | THE PUMPING LEMMA

A finite automaton accepting a language operates in a very simple way. Not surprisingly, the languages that can be accepted in this way are "simple" languages, but it is not yet clear exactly what this means. In this section, we will see one property that every language accepted by an FA must have.

Suppose that $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting $L \subseteq \Sigma^*$ and that Q has n elements. If x is a string in L with $|x| = n - 1$, so that x has n distinct prefixes, it is still conceivable that M is in a different state after processing every one. If $|x| \geq n$, however, then by the time M has read the symbols of x, it must have entered some state twice; there must be two different prefixes u and uv (saying they are different is the same as saying that $v \neq \Lambda$) such that

$$\delta^*(q_0, u) = \delta^*(q_0, uv)$$

This means that if $x \in L$ and w is the string satisfying $x = uvw$, then we have the situation illustrated by Figure 2.28. In the course of reading the symbols of $x = uvw$, M moves from the initial state to an accepting state by following a path that contains a loop, corresponding to the symbols of v. There may be more than one such loop, and more than one such way of breaking x into three pieces u, v, and w; but at least one of the loops must have been completed by the time M has read the first n symbols of x. In other words, for at least one of the choices of u, v, and w such that $x = uvw$ and v corresponds to a loop, $|uv| \leq n$.

The reason this is worth noticing is that it tells us there must be many more strings besides x that are also accepted by M and are therefore in L: strings that cause M to follow the same path but traverse the loop a different number of times. The string obtained from x by omitting the substring v is in L, because M doesn't have to traverse the loop at all. For each $i \geq 2$, the string $uv^i w$ is in L, because M can take the loop i times before proceeding to the accepting state.

The statement we have now proved is known as the Pumping Lemma for Regular Languages. "Pumping" refers to the idea of pumping up the string x by inserting additional copies of the string v (but remember that we also get one of the new strings by leaving out v). "Regular" won't be defined until Chapter 3, but we will see after we define regular languages that they turn out to be precisely the ones that can be accepted by finite automata.

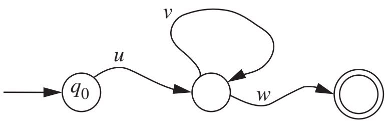


Figure 2.28 |
What the three strings u, v, and w in the pumping lemma might look like.

Theorem 2.29 The Pumping Lemma for Regular Languages
Suppose L is a language over the alphabet Σ. If L is accepted by a finite automaton $M = (Q, \Sigma, q_0, A, \delta)$, and if n is the number of states of M, then for every $x \in L$ satisfying $|x| \geq n$, there are three strings u, v, and w such that $x = uvw$ and the following three conditions are true:

1. $|uv| \leq n$.
2. $|v| > 0$ (i.e., $v \neq \Lambda$).
3. For every $i \geq 0$, the string uv^iw also belongs to L.

Later in this section we will find ways of applying this result for a language L that is accepted by an FA. But the most common application is to show that a language *cannot* be accepted by an FA, by showing that it doesn't have the property described in the pumping lemma.

A proof using the pumping lemma that L cannot be accepted by a finite automaton is a proof by contradiction. We assume, for the sake of contradiction, that L can be accepted by M, an FA with n states, and we try to select a string in L with length at least n so that statements 1–3 lead to a contradiction. There are a few places in the proof where it's easy to go wrong, so before looking at an example, we consider points at which we have to be particularly careful.

Before we can think about applying statements 1–3, we must have a string $x \in L$ with $|x| \geq n$. What is n? It's the number of states in M, but we don't know what M is—the whole point of the proof is to show that it can't exist! In other words, our choice of x must involve n. We can't say "let $x = aababaabbab$", because there's no reason to expect that $11 \geq n$. Instead, we might say "let $x = a^nba^{2n}$", or "let $x = b^{n+1}a^nb$", or something comparable, depending on L.

The pumping lemma tells us some properties that *every* string in L satisfies, as long as its length is at least n. It is very possible that for some choices of x, the fact that x has these properties does not produce any contradiction. If we don't get a contradiction, we haven't proved anything, and so we look for a string x that *will* produce one. For example, if we are trying to show that the language of palindromes over $\{a, b\}$ cannot be accepted by an FA, there is no point in considering a string x containing only a's, because all the new strings that we will get by using the pumping lemma will also contain only a's, and they're all palindromes too. No contradiction!

Once we find a string x that looks promising, the pumping lemma says that there is *some* way of breaking x into shorter strings u, v, and w satisfying the three conditions. It doesn't tell us what these shorter strings are, only that they satisfy conditions 1–3. If $x = a^nb^na^n$, we can't say "let $u = a^{10}$, $v = a^{n-10}$, and $w = b^na^n$". It's not enough to show that *some* choices for u, v, and w produce a contradiction—we have to show that we *must* get a contradiction, no matter what u, v, and w are, as long as they satisfy conditions 1–3.

Let's try an example.

The Language *AnBn* **EXAMPLE 2.30**

Let L be the language *AnBn* introduced in Example 1.18:

$$L = \{a^i b^i \mid i \geq 0\}$$

It would be surprising if *AnBn* could be accepted by an FA; if the beginning input symbols are a's, a computer accepting L surely needs to remember how many of them there are, because otherwise, once the input switches to b's, it won't be able to compare the two numbers.

Suppose for the sake of contradiction that there is an FA M having n states and accepting L. We choose $x = a^n b^n$. Then $x \in L$ and $|x| \geq n$. Therefore, by the pumping lemma, there are strings u, v, and w such that $x = uvw$ and the conditions 1–3 in the theorem are true.

Because $|uv| \leq n$ (by condition 1) and the first n symbols of x are a's (because of the way we chose x), all the symbols in u and v must be a's. Therefore, $v = a^k$ for some $k > 0$ (by condition 2). We can get a contradiction from statement 3 by using any number i other than 1, because $uv^i w$ will still have exactly n b's but will no longer have exactly n a's. The string uv^2w, for example, is $a^{n+k}b^n$, obtained by inserting k additional a's into the first part of x. This is a contradiction, because the pumping lemma says $uv^2w \in L$, but $n + k \neq n$.

Not only does the string uv^2w fail to be in L, but it also fails to be in the bigger language *AEqB* containing all strings in $\{a, b\}^*$ with the same number of a's as b's. Our proof, therefore, is also a proof that *AEqB* cannot be accepted by an FA.

The Language $\{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$ **EXAMPLE 2.31**

Let L be the language

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

The first sentence of a proof using the pumping lemma is always the same: Suppose for the sake of contradiction that there is an FA M that accepts L and has n states. There are more possibilities for x than in the previous example; we will suggest several choices, all of which satisfy $|x| \geq n$ but some of which work better than others in the proof.

First we try $x = b^n a^{2n}$. Then certainly $x \in L$ and $|x| \geq n$. By the pumping lemma, $x = uvw$ for some strings u, v, and w satisfying conditions 1–3. Just as in Example 2.30, it follows from conditions 1 and 2 that $v = b^k$ for some $k > 0$. We can get a contradiction from condition 3 by considering $uv^i w$, where i is large enough that $n_b(uv^i w) \geq n_a(uv^i w)$. Since $|v| \geq 1$, $i = n + 1$ is guaranteed to be large enough. The string $uv^{n+1}w$ has at least n more b's than x does, and therefore at least $2n$ b's, but it still has exactly $2n$ a's.

Suppose that instead of $b^n a^{2n}$ we choose $x = a^{2n} b^n$. This time $x = uvw$, where v is a string of one or more a's and $uv^i w \in L$ for every $i \geq 0$. The way to get a contradiction now is to consider $uv^0 w$, which has fewer a's than x does. Unfortunately, this produces a contradiction only if $|v| = n$. Since we don't know what $|v|$ is, the proof will not work for this choice of x.

The problem is not that x contains a's before b's; rather, it is that the original numbers of a's and b's are too far apart to guarantee a contradiction. Getting a contradiction in this case means making an inequality fail; if we start with a string in which the inequality is just barely satisfied, then ideally any change in the right direction will cause it to fail. A better choice, for example, is $x = a^{n+1}b^n$. (If we had used $x = b^n a^{n+1}$ instead of $b^n a^{2n}$ for our first choice, we could have used $i = 2$ instead of $i = n$ to get a contradiction.)

Letting $x = (ab)^n a$ is also a bad choice, but for a different reason. We know that $x = uvw$ for some strings u, v, and w satisfying conditions 1–3, but now we don't have enough information about the string v. It might be $(ab)^k a$ for some k, so that uv^0w produces a contradiction; it might be $(ba)^k b$, so that uv^2w produces a contradiction; or it might be either $(ab)^k$ or $(ba)^k$, so that changing the number of copies of v doesn't change the relationship between n_a and n_b and doesn't give us a contradiction.

EXAMPLE 2.32 The Language $L = \{a^{i^2} \mid i \geq 0\}$

Whether a string of a's is an element of L depends only on its length; in this sense, our proof will be more about numbers than about strings.

Suppose L can be accepted by an FA M with n states. Let us choose x to be the string a^{n^2}. Then according to the pumping lemma, $x = uvw$ for some strings u, v, and w satisfying conditions 1–3. Conditions 1 and 2 tell us that $0 < |v| \leq n$. Therefore,

$$n^2 = |uvw| < |uv^2w| = n^2 + |v| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$$

This is a contradiction, because condition 3 says that $|uv^2w|$ must be i^2 for some integer i, but there is no integer i whose square is strictly between n^2 and $(n+1)^2$.

EXAMPLE 2.33 Languages Related to Programming Languages

Almost exactly the same pumping-lemma proof that we used in Example 2.30 to show *AnBn* cannot be accepted by a finite automaton also works for several other languages, including several that a compiler for a high-level programming language must be able to accept. These include both the languages *Balanced* and *Expr* introduced in Example 1.19, because $(^m)^n$ is a balanced string, and $(^m a)^n$ is a legal expression, if and only if $m = n$.

Another example is the set L of legal C programs. We don't need to know much about the syntax of C to show that this language can't be accepted by an FA—only that the string

```
main(){{{ ... }}}
```

with m occurrences of "{" and n occurrences of "}", is a legal C program precisely if $m = n$. As usual, we start our proof by assuming that L is accepted by some FA with n states and letting x be the string `main()` $\{^n\}^n$. If $x = uvw$, and these three strings satisfy conditions 1–3, then the easiest way to obtain a contradiction is to use $i = 0$ in condition 3. The string v cannot contain any right brackets because of condition 1; if the shorter string uw is missing any of the symbols in "`main()`", then it doesn't have the legal header necessary for a C program, and if it is missing any of the left brackets, then the two numbers don't match.

As the pumping lemma demonstrates, one way to answer questions about a language is to examine a finite automaton that accepts it. In particular, for languages that can be accepted by FAs, there are several *decision problems* (questions with yes-or-no answers) we can answer this way, and some some of them have decision algorithms that take advantage of the pumping lemma.

Decision Problems Involving Languages Accepted by Finite Automata — EXAMPLE 2.34

The most fundamental question about a language L is which strings belong to it. The *membership problem* for a language L accepted by an FA M asks, for an arbitrary string x over the input alphabet of M, whether $x \in L(M)$. We can think of the problem as specific to M, so that an *instance* of the problem is a particular string x; we might also ask the question for an arbitrary M and an arbitrary x, and consider an instance to be an ordered pair (M, x). In either case, the way a finite automaton works makes it easy to find a decision algorithm for the problem. Knowing the string x and the specifications for $M = (Q, \Sigma, q_0, A, \delta)$ allows us to compute the state $\delta^*(q_0, x)$ and check to see whether it is an element of A.

The two problems below are examples of other questions we might ask about $L(M)$:

1. Given an FA $M = (Q, \Sigma, q_0, A, \delta)$, is $L(M)$ nonempty?
2. Given an FA $M = (Q, \Sigma, q_0, A, \delta)$, is $L(M)$ infinite?

One way for the language $L(M)$ to be empty is for A to be empty, but this is not the only way. The real question is not whether M has any accepting states, but whether it has any that are *reachable* from q_0. The same algorithm that we used in Example 1.21 to find the set of cities reachable from city S can be used here; another algorithm that is less efficient but easier to describe is the following.

■ Decision Algorithm for Problem 1

Let n be the number of states of M. Try strings in order of length, to see whether any are accepted by M; $L(M) \neq \emptyset$ if and only if there is a string of length less than n that is accepted, where n is the number of states of M.

The reason this algorithm works is that according to the pumping lemma, for every string $x \in L(M)$ with $|x| \geq n$, there is a shorter string in $L(M)$, the one obtained by omitting the middle portion v. Therefore, it is impossible for the shortest string accepted by M to have length n or more.

It may be less obvious that an approach like this will work for problem 2, but again we can use the pumping lemma. If n is the number of states of M, and x is a string in $L(M)$ with $|x| \geq n$, then there are infinitely many longer strings in $L(M)$. On the other hand, if $x \in L(M)$ and $|x| \geq 2n$, then $x = uvw$ for some strings u, v, and w satisfying the conditions in the pumping lemma, and $uv^0w = uw$ is a shorter string in $L(M)$ whose length is still n or more, because $|v| \leq n$. In other words, if there is a string in L with length at least n, the shortest such string must have length less than $2n$. It follows that the algorithm below, which is undoubtedly inefficient, is a decision algorithm for problem 2.

■ Decision Algorithm for Problem 2

Try strings in order of length, starting with strings of length n, to see whether any are accepted by M. $L(M)$ is infinite if and only if a string x is found with $n \leq |x| < 2n$.

2.5 HOW TO BUILD A SIMPLE COMPUTER USING EQUIVALENCE CLASSES

In Section 2.3 we considered a three-state finite automaton M accepting L, the set of strings over $\{a, b\}$ that end with aa. We showed that three states were necessary by finding three pairwise L-distinguishable strings, one for each state of M. Now we are interested in why three states are enough. Of course, if M really does accept L, then three are enough; we can check that this is the case by showing that if x and y are two strings that cause M to end up in the same state, then M doesn't need to distinguish them, because they are not L-distinguishable.

Each state of M corresponds to a set of strings, and we described the three sets in Example 2.1: the strings that do not end with a, the strings that end with a but not aa, and the strings in L. If x is a string in the second set, for example, then for every string z, $xz \in L$ precisely if $z = a$ or z itself ends in aa. We don't need to know what x is in order to say this, only that x ends with a but not aa. Therefore, no two strings in this set are L-distinguishable. A similar argument works for each of the other two sets.

We will use "L-indistinguishable" to mean not L-distinguishable. We can now say that if S is any one of these three sets, then

1. Any two strings in S are L-indistinguishable.
2. No string of S is L-indistinguishable from a string not in S.

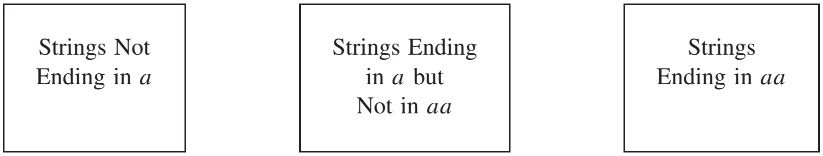
If the "L-indistinguishability" relation is an equivalence relation, then as we pointed out at the end of Section 1.3, these two statements about a set S say precisely that S is one of the equivalence classes. The relation is indeed an equivalence relation. Remember that x and y are L-indistinguishable if and only if $L/x = L/y$ (see Definition 2.20); this characterization makes it easy to see that the relation is reflexive, symmetric, and transitive, because the equality relation has these properties.

Definition 2.35 The *L*-Indistinguishability Relation

For a language $L \subseteq \{a, b\}^*$, we define the relation I_L (an equivalence relation) on Σ^* as follows: For $x, y \in \Sigma^*$,

$$x I_L y \text{ if and only if } x \text{ and } y \text{ are } L\text{-indistinguishable}$$

In the case of the language L of strings ending with aa, we started with the three-state FA and concluded that for each state, the corresponding set was one of the equivalence classes of I_L. What if we didn't have an FA but had figured out what the equivalence classes were, and that there were only three?


Strings Not Ending in *a*
Strings Ending in *a* but Not in *aa*
Strings Ending in *aa*

Then we would have at least the first important ingredient of a finite automaton accepting L: a finite set, whose elements we could call *states*. Calling a set of strings a state is reasonable, because we already had in mind an association between a state and the set of strings that led the FA to that state. But in case it seems questionable, states don't have to *be* anything special—they only have to be representable by circles (or, as above, rectangles), with some coherent way of defining an initial state, a set of accepting states, and a transition function.

Once we start to describe this FA, the details fall into place. The initial state should be the equivalence class containing the string Λ, because Λ is one of the strings corresponding to the initial state of any FA. Because we want the FA to accept L, the accepting states (in this case, only one) should be the equivalence classes containing elements of L.

Let us take one case to illustrate how the transitions can be defined. The third equivalence class is the set containing strings ending with aa. If we choose an arbitrary element, say $abaa$, we have one string that we want to correspond to that state. If the next input symbol is b, then the current string that results is $abaab$. Now we simply have to determine which of the three sets this string belongs to, and the answer is the first. You can see in the other cases as well that what we end up with is the diagram shown in Figure 2.2.

That's almost all there is to the construction of the FA, although there are one or two subtleties in the proof of Theorem 2.36. The other half of the theorem says that there is an FA accepting L *only* if the set of equivalence classes of I_L is finite. The two parts therefore give us an if-and-only-if characterization of the languages that can be accepted by finite automata. The pumping lemma in Section 2.4 does not; we will see in Example 2.39 that there are languages L for which, although L is not accepted by any FA, the pumping lemma does not allow us to prove this and Theorem 2.36 does.

Theorem 2.36

If $L \subseteq \Sigma^*$ can be accepted by a finite automaton, then the set Q_L of equivalence classes of the relation I_L on Σ^* is finite. Conversely, if the set Q_L is finite, then the finite automaton $M_L = (Q_L, \Sigma, q_0, A, \delta)$ accepts L, where $q_0 = [\Lambda]$, $A = \{q \in Q_L \mid q \subseteq L\}$, and for every $x \in \Sigma^*$ and every $a \in \Sigma$,

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

Finally, M_L has the fewest states of any FA accepting L.

Proof

If Q_L is infinite, then a set S containing exactly one string from each equivalence class is an infinite set of pairwise L-distinguishable strings. If there were an FA accepting L, it would have k states, for some k; but S has $k+1$ pairwise L-distinguishable strings, and it follows from Theorem 2.21 that every FA accepting L must have at least $k+1$ states. Therefore, there is no FA accepting L.

If Q_L is finite, on the other hand, we want to show that the FA M_L accepts L. First, however, we must consider the question of whether the definition of M_L even makes sense. The formula $\delta([x], a) = [xa]$ is supposed to assign an equivalence class (exactly one) to the ordered pair $([x], a)$. The equivalence class $[x]$ containing the string x might also contain another string y, so that $[x] = [y]$. In order for the formula to be a sensible definition of δ, it should be true that in this case

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

The question, then, is whether the statement $[x] = [y]$ implies the statement $[xa] = [ya]$. Fortunately, it does. If $[x] = [y]$, then $x I_L y$, which means that for every string z', xz' and yz' are either both in L or both not in L; therefore, for every z, xaz and yaz are either both in L or both not in L (because of the previous statement with $z' = az$), and so xaI_Lya.

The next step in the proof is to verify that with this definition of δ, the formula

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

holds for every two strings $x, y \in \Sigma^*$. This is a straightforward proof by structural induction on y, which uses the definition of δ for this FA and the definition of δ^* for any FA.

From this formula, it follows that

$$\delta^*(q_0, x) = \delta^*([\Lambda], x) = [x]$$

It follows from our definition of A that x is accepted by M_L if and only if $[x] \subseteq L$. What we want is that x is accepted if and only if $x \in L$, and so we must show that $[x] \subseteq L$ if and only if $x \in L$. If the first statement is true, then the second is, because $x \in [x]$. On the other hand, if $x \in L$, and y is any element of $[x]$, then yI_Lx. Since $x\Lambda \in L$, and x and y are L-indistinguishable, $y\Lambda = y \in L$. Therefore, $[x] \subseteq L$.

A set containing one element from each equivalence class of I_L is a set of pairwise L-distinguishable strings. Therefore, by the second statement of Theorem 2.21, every FA accepting L must have at least as many states as there are equivalence classes. M_L, with exactly this number of states, has the fewest possible.

The statement that L can be accepted by an FA if and only if the set of equivalence classes of I_L is finite is known as the Myhill-Nerode Theorem.

As a practical matter, if we are trying to construct a finite automaton to accept a language L, it is often easier to attack the problem directly, as in Example 2.22, than to determine the equivalence classes of I_L. Theorem 2.36 is interesting because it can be interpreted as an answer to the question of how much a computer accepting a language L needs to remember about the current string x. It can forget everything about x except the equivalence class it belongs to. The theorem provides an elegant description of an "abstract" finite automaton accepting a language, and we will see in the next section that it will help us if we already have an FA and are trying to simplify it by eliminating all the unnecessary states.

If identifying equivalence classes is not always the easiest way to construct an FA accepting a language L, identifying the equivalence classes from an existing FA $M = (Q, \Sigma, q_0, A, \delta)$ is relatively straightforward. For each state q, we define

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

Every one of the sets L_q is a subset of some equivalence class of I_L; otherwise (if for some q, L_q contained strings in two different equivalence classes), there would be two L-distinguishable strings that caused M to end up in the same state, which would contradict Theorem 2.21. It follows that the number of equivalence classes is no larger than the number of states. If M has the property that strings corresponding to different states are L-distinguishable, then the two numbers are the same, and the equivalence classes are precisely the sets L_q, just as for the language of strings ending in aa discussed at the beginning of this section.

It is possible that an FA M accepting L has more states than I_L has equivalence classes. This means that for at least one equivalence class S, there are two different states q_1 and q_2 such that L_{q_1} and L_{q_2} are both subsets of S. We will see in the next section that in this case M has more states than it needs, and we will obtain an algorithm to simplify M by combining states wherever possible. The sets L_q for the simplified FA are the equivalence classes of I_L, as in the preceding paragraph.

In the next example in this section, we return to the language *AnBn* and calculate the equivalence classes. We have already used the pumping lemma to show there is no FA accepting this language (Example 2.30), so we will not be surprised to find that there are infinitely many equivalence classes.

The Equivalence Classes of I_L, Where $L = AnBn$ **EXAMPLE 2.37**

As we observed in Example 2.30, accepting $AnBn = \{a^n b^n \mid n \geq 0\}$ requires that we remember how many a's we have read, so that we can compare that number to the number of b's. A precise way to say this is that two different strings of a's are L-distinguishable: if $i \neq j$, then $a^i b^i \in L$ and $a^j b^i \notin L$. Therefore, the equivalence classes $[a^j]$ are all distinct. If we were interested only in showing that the set of equivalence classes is infinite, we could stop here.

Exactly what are the elements of $[a^j]$? Not only is the string a^j L-distinguishable from a^i, but it is L-distinguishable from *every* other string x: A string that distinguishes the two is ab^{j+1}, because $a^j ab^{j+1} \in L$ and $xab^{j+1} \notin L$. Therefore, there are no other strings in the set $[a^j]$, and

$$[a^j] = \{a^j\}$$

Each of the strings a^i is a prefix of an element of L. Other prefixes of elements of L include elements of L themselves and strings of the form $a^i b^j$ where $i > j$. All other strings in $\{a, b\}^*$ are nonprefixes of elements of L.

Two nonnull elements of L are L-indistinguishable, because if a string other than Λ is appended to the end of either one, the result is not in L; and every nonnull string in L can be distinguished from every string not in L by the string Λ. Therefore, the set $L - \{\Lambda\}$ is an equivalence class of I_L.

The set of nonprefixes of elements of L is another equivalence class: No two nonprefixes can be distinguished relative to L, and if $xy \in L$, then the string y distinguishes x from every nonprefix.

We are left with the strings $a^i b^j$ with $i > j > 0$. Let's consider an example, say $x = a^7 b^3$. The only string z with $xz \in L$ is b^4. However, there are many other strings y that share this property with x; every string $a^{i+4} b^i$ with $i > 0$ does. The equivalence class $[a^7 b^3]$ is the set $\{a^{i+4} b^i \mid i > 0\} = \{a^5 b, a^6 b^2, a^7 b^3, \ldots\}$. Similarly, for every $k > 0$, the set $\{a^{i+k} b^i \mid i > 0\}$ is an equivalence class.

To summarize, L and the set of nonprefixes of elements of L are two equivalence classes that are infinite sets. For each $j \geq 0$, the set with the single element a^j is an equivalence class; and for every $k > 0$, the infinite set $\{a^{k+i} b^i \mid i > 0\}$ is an equivalence class. We have now accounted for all the strings in $\{a, b\}^*$.

EXAMPLE 2.38 The Equivalence Classes of I_L, Where $L = \{a^{n^2} | n \in \mathcal{N}\}$

We show that if L is the language in Example 2.36 of strings in $\{a\}^*$ with length a perfect square, then the elements of $\{a\}^*$ are pairwise L-distinguishable. Suppose i and j are two integers with $0 \leq i < j$; we look for a positive integer k so that $j + k$ is a perfect square and $i + k$ is not, so that $a^j a^k \in L$ and $a^i a^k \notin L$. Let $k = (j + 1)^2 - j = j^2 + j + 1$. Then $j + k = (j + 1)^2$ but $i + k = j^2 + i + j + 1 > j^2$, so that $i + k$ falls between j^2 and $(j + 1)^2$.

The language $Pal \subseteq \{a, b\}^*$ (Examples 1.18 and 2.31) was the first one we found for which no equivalence class of I_L has more than one element. That example was perhaps a little more dramatic than this one, in which the argument depends only on the length of the string. Of course, palindromes over a one-symbol alphabet are not very interesting. If we took L to be the set of all strings in $\{a, b\}^*$ with length a perfect square, each equivalence class would correspond to a natural number n and would contain all strings of that length.

According to the pumping lemma, if L is accepted by an FA M, then there is a number n, depending on M, such that we can draw some conclusions about every string in L with length at least n. In the applications in Section 2.4, we didn't need to know where the number n came from, only that it existed. If there is no such number (that is, if the assumption that there is leads to a contradiction), then L cannot be accepted by an FA. If there is such a number—never mind where it comes from—does it follow that there is an FA accepting L? The answer is no, as the following slightly complicated example illustrates.

A Language Can Satisfy the Conclusions of the Pumping Lemma and Still Not Be Accepted by a Finite Automaton

EXAMPLE 2.39

Consider the language

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

Strings in L contain a's, then b's, then c's; if there is at least one a in the string, then the number of b's and the number of c's have to be the same, and otherwise they don't.

We will show that for the number $n = 1$, the statement in the pumping lemma is true for L. Suppose $x \in L$ and $|x| \geq 1$. If there is at least one a in the string x, and $x = a^i b^j c^j$, we can define

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

Every string of the form $uv^i w$ is still of the form $a^k b^j c^j$ and is therefore still an element of L, whether k is 0 or not. If x is $b^i c^j$, on the other hand, then again we define u to be Λ and v to be the first symbol in x, which is either b or c. It is also true in this case that $uv^i w \in L$ for every $i \geq 0$.

We can use Theorem 2.26 to show that there is no finite automaton accepting our language L, because there is an infinite set of pairwise L-distinguishable strings. If S is the set $\{ab^n \mid n \geq 0\}$, any two distinct elements ab^m and ab^n are distinguished by the string c^m.

2.6 MINIMIZING THE NUMBER OF STATES IN A FINITE AUTOMATON

Suppose we have a finite automaton $M = (Q, \Sigma, q_0, A, \delta)$ accepting $L \subseteq \Sigma^*$. For a state q of M, we have introduced the notation L_q to denote the set of strings that cause M to be in state q:

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

The first step in reducing the number of states of M as much as possible is to eliminate every state q for which $L_q = \emptyset$, along with transitions from these states. None of these states is reachable from the initial state, and eliminating them does not change the language accepted by M. For the remainder of this section, we assume that all the states of M are reachable from q_0.

We have defined an equivalence relation on Σ^*, the L-indistinguishability relation I_L, and we have seen that for each state q in M, all the strings in L_q are L-indistinguishable. In other words, the set L_q is a subset of one of the equivalence classes of L_q.

The finite automaton we described in Theorem 2.36, with the fewest states of any FA accepting L, is the one in which each state corresponds precisely to (according to our definition, *is*) one of the equivalence classes of I_L. For each state q in this FA, L_q is as large as possible—it contains every string in some equivalence class of I_L. Every FA in which this statement doesn't hold has more states than it needs. There are states p and q such that the strings in L_p are L-indistinguishable from those in L_q; if M doesn't need to distinguish between these two types of strings, then q can be eliminated, and the set L_p can be enlarged by adding the strings in L_q.

The equivalence relation on Σ^* gives us an equivalence relation $\equiv$ on the set Q of states of M. For $p, q \in Q$,

$$p \equiv q \text{ if and only if strings in } L_p \text{ are } L\text{-indistinguishable from strings in } L_q$$

This is the same as saying

$$p \equiv q \text{ if and only if } L_p \text{ and } L_q \text{ are subsets of the same equivalence class of } I_L$$

Two strings x and y are L-distinguishable if for some string z, exactly one of the two strings xz, yz is in L. Two states p and q fail to be equivalent if strings x and y corresponding to p and q, respectively, are L-distinguishable, and this means:

$$p \not\equiv q \text{ if and only if, for some string } z \text{, exactly one of the states } \delta^*(p, z), \delta^*(q, z) \text{ is in } A$$

In order to simplify M by eliminating unnecessary states, we just need to identify the unordered pairs (p, q) for which the two states can be combined into one. The definition of $p \not\equiv q$ makes it easier to identify the pairs (p, q) for which p and q cannot be combined, the ones for which $p \not\equiv q$. We look systematically for a string z that might distinguish the states p and q (or distinguish a string in L_p from one in L_q). With this in mind, we let S_M be the set of such unordered pairs.

$$S_M \text{ is the set of unordered pairs } (p, q) \text{ of distinct states satisfying } p \not\equiv q$$

A Recursive Definition of S_M

The set S_M can be defined as follows:

1. For every pair (p, q) with $p \neq q$, if exactly one of the two states is in A, $(p, q) \in S_M$.
2. For every pair (r, s) of distinct states, if there is a symbol $\sigma \in \Sigma$ such that the pair $(\delta(r, \sigma), \delta(s, \sigma))$ is in S_M, then $(r, s) \in S_M$.

In the basis statement we get the pairs of states that can be distinguished by Λ. If the states $\delta(r, \sigma)$ and $\delta(s, \sigma)$ are distinguished by the string z, then the states r and s are distinguished by the string σz; as a result, if the states r and s can be distinguished by a string of length n, then the pair (r, s) can be added to the set by using the recursive part of the definition n or fewer times.

Because the set S_M is finite, this recursive definition provides an algorithm for identifying the elements of S_M.

Algorithm 2.40 Identifying the Pairs *(p, q)* with $p \not\equiv q$ List all unordered pairs of distinct states (p, q). Make a sequence of passes through these pairs as follows. On the first pass, mark each pair (p, q) for which exactly one of the two states p and q is in A. On each subsequent pass, examine each unmarked pair (r, s); if there is a symbol $\sigma \in \Sigma$ such that $\delta(r, \sigma) = p$, $\delta(s, \sigma) = q$, and the pair (p, q) has already been marked, then mark (r, s).

After a pass in which no new pairs are marked, stop. At that point, the marked pairs (p, q) are precisely those for which $p \not\equiv q$. ■

Algorithm 2.40 is the crucial ingredient in the algorithm to simplify the FA by minimizing the number of states. When Algorithm 2.40 terminates, every pair (p, q) that remains unmarked represents two states that can be combined into one, because the corresponding sets L_p and L_q are subsets of the same equivalence class. It may happen that every pair ends up marked; this means that for distinct states p and q, strings in p are already L-distinguishable from strings in q, and M already has the fewest states possible.

We finish up by making one final pass through the states. The first state to be considered represents an equivalence class, or a state in our new minimal FA. After that, a state q represents a new equivalence class, or a new state, only if the pair (p, q) is marked for every state p considered previously. Each time we consider a state q that does not produce a new state in the minimal FA, because it is equivalent to a previous state p, we will add it to the set of original states that are being combined with p.

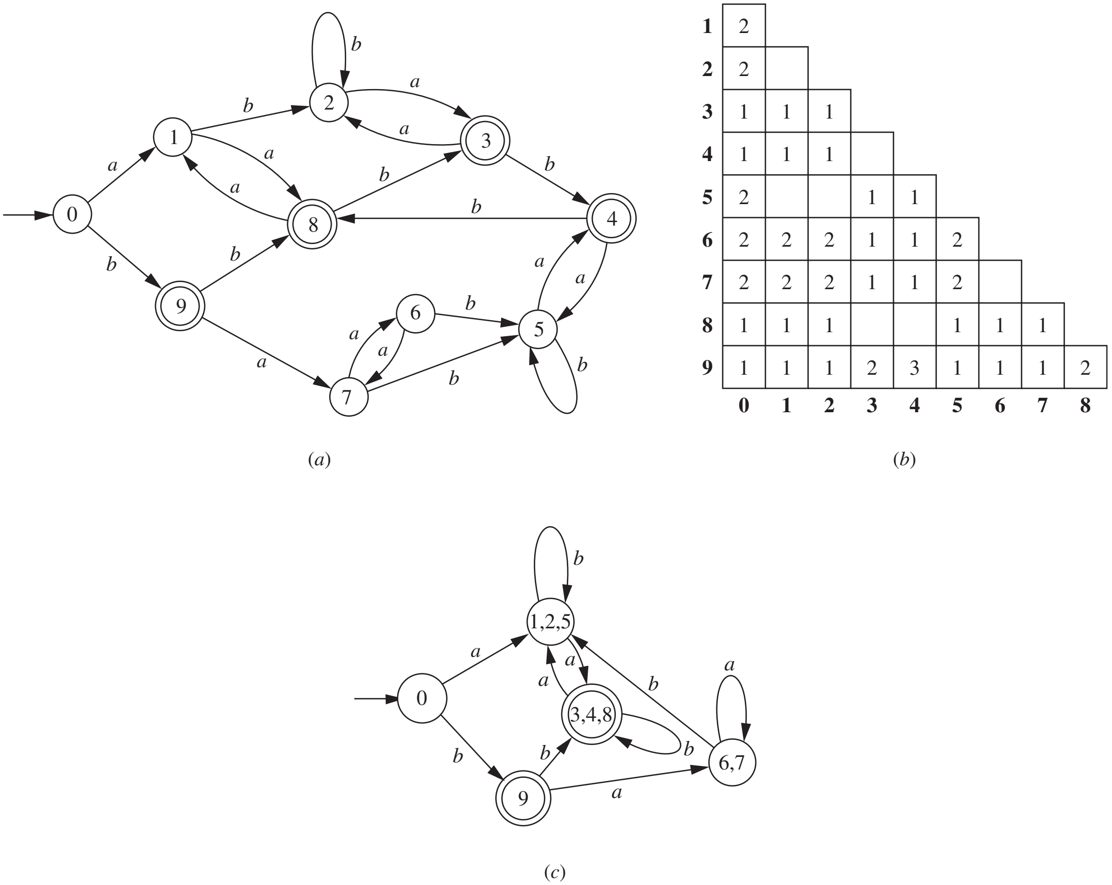
Once we have the states in the resulting minimum-state FA, determining the transitions is straightforward. Example 2.41 illustrates the algorithm.

Applying the Minimization Algorithm EXAMPLE 2.41

Figure 2.42a shows a finite automaton with ten states, numbered 0–9, and Figure 2.42b shows the unordered pairs (p, q) with $p \neq q$. The pairs marked 1 are the ones marked on pass 1, in which exactly one state is an accepting state, and those marked 2 or 3 are the ones marked on the second or third pass. In this example, the third pass is the last one on which new pairs were marked.

How many passes are required and which pairs are marked on each one may depend on the order in which the pairs are considered during each pass. The results in Figure 2.42b are obtained by proceeding one vertical column at a time, and considering the pairs in each column from top to bottom.

The pair (6, 3) is one of the pairs marked on the first pass, since 3 is an accepting state and 6 is not. When the pair (7, 2) is considered on the second pass, it is marked because $\delta(7, a) = 6$ and $\delta(2, a) = 3$. When the pair (9, 3) is considered later on the second pass, it is



1	2								
2	2								
3	1	1	1						
4	1	1	1						
5	2			1	1				
6	2	2	2	1	1	2			
7	2	2	2	1	1	2			
8	1	1	1			1	1	1	
9	1	1	1	2	3	1	1	1	2
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Figure 2.42 |
Minimizing the number of states in an FA.

also marked, because $\delta(9, a) = 7$ and $\delta(3, a) = 2$. The pair $(7, 5)$ is marked on the second pass. We have $\delta(9, a) = 7$ and $\delta(4, a) = 5$, but $(9, 4)$ was considered earlier on the second pass, and so it is not marked until the third pass.

With the information from Figure 2.42b, we can determine the states in the minimal FA as follows. State 0 will be a new state. State 1 will be a new state, because the pair $(1, 0)$ is marked. State 2 will not, because $(2, 1)$ is unmarked, which means we combine states 2 and 1. State 3 will be a new state. State 4 will be combined with 3. State 5 will be combined with states 1 and 2, because both $(5, 1)$ and $(5, 2)$ are unmarked. State 6 will be a new state; state 7 is combined with state 6; state 8 is combined with 3 and 4; and state 9 is a new state. At this point, we have the five states shown in Figure 2.42c.

If we designate each state in the FA by the set of states in the original FA that were combined to produce it, we can compute the transitions from the new state by considering any of the elements of that set. For example, one of the new states is $\{1, 2, 5\}$; in the original FA, $\delta(1, a) = 8$, which tells us that the a-transition from $\{1, 2, 5\}$ goes to $\{3, 4, 8\}$. (If there are any inconsistencies, such as $\delta(5, a)$ not being an element of $\{3, 4, 8\}$, then we've made a mistake somewhere!)

EXERCISES

2.1. In each part below, draw an FA accepting the indicated language over $\{a, b\}$.

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

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

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

d. The language of all strings that begin or end with aa or bb.

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

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

g. The language of all strings in which both the number of a's and the number of b's are even.

h. The language of all strings containing no more than one occurrence of the string aa. (The string aaa contains two occurrences of aa.)

i. The language of all strings in which every a (if there are any) is followed immediately by bb.

j. The language of all strings containing both bb and aba as substrings.

k. The language of all strings containing both aba and bab as substrings.

2.2. For each of the FAs pictured in Fig. 2.43, give a simple verbal description of the language it accepts.

2.3. a. Draw a transition diagram for an FA that accepts the string $abaa$ and no other strings.

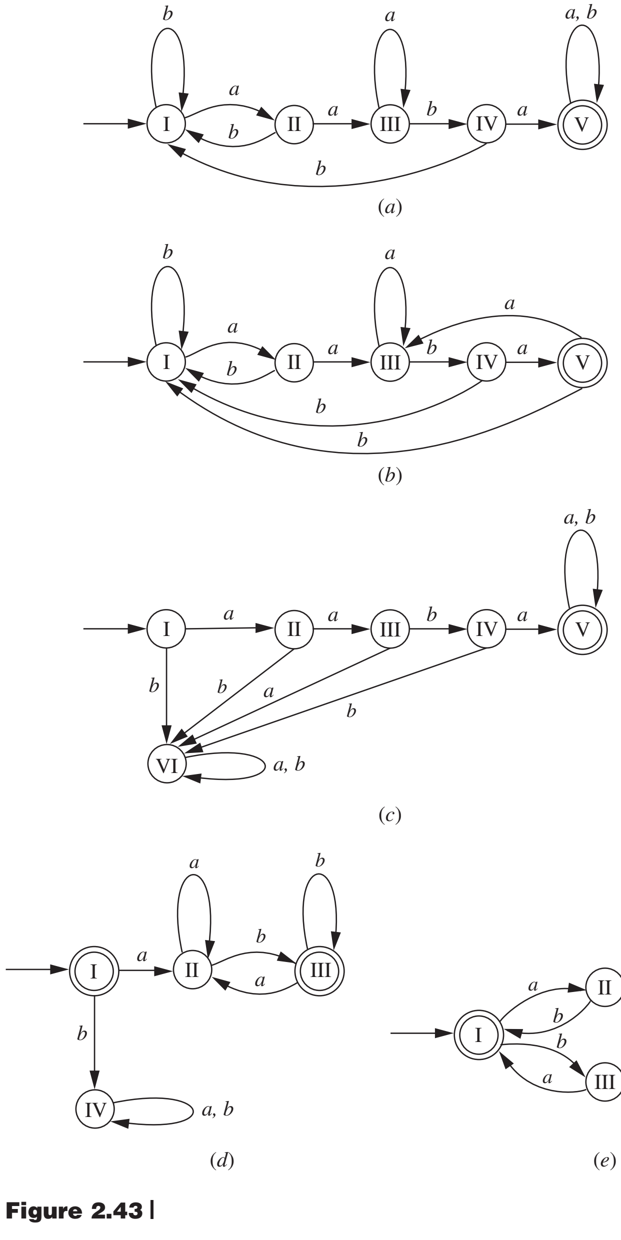
b. For a string $x \in \{a, b\}^*$ with $|x| = n$, how many states are required for an FA accepting x and no other strings? For each of these states, describe the strings that cause the FA to be in that state.

c. For a string $x \in \{a, b\}^*$ with $|x| = n$, how many states are required for an FA accepting the language of all strings in $\{a, b\}^*$ that begin with x? For each of these states, describe the strings that cause the FA to be in that state.

2.4. Example 2.7 describes an FA accepting L_3, the set of strings in $\{0, 1\}^*$ that are binary representations of integers divisible by 3. Draw a transition diagram for an FA accepting L_5.

2.5. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA, q is an element of Q, and x and y are strings in Σ^*. Using structural induction on y, prove the formula

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



(a)

(b)

(c)

(d)

(e)

Figure 2.43 |

2.6. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA, q is an element of Q, and $\delta(q, \sigma) = q$ for every $\sigma \in \Sigma$. Show using structural induction that for every $x \in \Sigma^*$, $\delta^*(q, x) = q$.

2.7. 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 p in Q for which $\delta^*(p, z) \in A$ for some string z. What is the relationship between the language accepted by M_1 and the language accepted by M? Prove your answer.

2.8. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. Below are other conceivable methods of defining the extended transition function δ^* (see Definition 2.12). 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 2.12 does.

a. For every $q \in Q$, $\delta^*(q, \Lambda) = q$; for every $y \in \Sigma^*$, $\sigma \in \Sigma$, and $q \in Q$, $\delta^*(q, y\sigma) = \delta^*(\delta^*(q, y), \sigma)$.

b. For every $q \in Q$, $\delta^*(q, \Lambda) = q$; for every $y \in \Sigma^*$, $\sigma \in \Sigma$, and $q \in Q$, $\delta^*(q, \sigma y) = \delta^*(\delta(q, \sigma), y)$.

c. For every $q \in Q$, $\delta^*(q, \Lambda) = q$; for every $q \in Q$ and every $\sigma \in \Sigma$, $\delta^*(q, \sigma) = \delta(q, \sigma)$; for every $q \in Q$, and every x and y in Σ^*, $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$.

2.9. In order to test a string for membership in a language like the one in Example 2.1, 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 such that for every string x of length n or greater, x is in the language if and only if $x = yz$ for some $z \in S$.

a. Show that every language L having this property can be accepted by an FA.

b. Show that every finite language has this property.

c. Give an example of an infinite language that can be accepted by an FA but does not have this property.

2.10. Let M_1 and M_2 be the FAs pictured in Figure 2.44, accepting languages L_1 and L_2, respectively.
Draw FAs accepting the following languages.

a. $L_1 \cup L_2$

b. $L_1 \cap L_2$

c. $L_1 - L_2$

2.11. (For this problem, refer to the proof of Theorem 2.15.) Show that for every $x \in \Sigma^*$ and every $(p, q) \in Q$, $\delta^*((p, q), x) = (\delta_1^*(p, x), \delta_2^*(q, x))$.

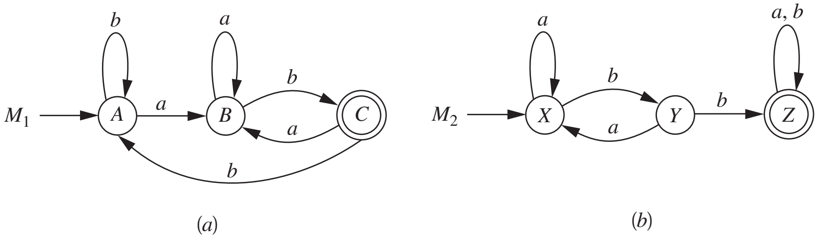


Figure 2.44 |

2.12. For each of the following languages, draw an FA accepting it.

a. $\{a, b\}^*\{a\}$

b. $\{bb, ba\}^*$

c. $\{a, b\}^*\{b, aa\}\{a, b\}^*$

d. $\{bbb, baa\}^*\{a\}$

e. $\{a\} \cup \{b\}\{a\}^* \cup \{a\}\{b\}^*\{a\}$

f. $\{a, b\}^*\{ab, bba\}$

g. $\{b, bba\}^*\{a\}$

h. $\{aba, aa\}^*\{ba\}^*$

2.13. For the FA pictured in Fig. 2.17d, show that there cannot be any other FA with fewer states accepting the same language. (See Example 2.24, in which the same result is established for the FA accepting the language L_n.)

2.14. Let z be a fixed string of length n over the alphabet $\{a, b\}$. Using the argument in Example 2.5, we can find an FA with $n + 1$ states accepting the language of all strings in $\{a, b\}^*$ that end in z. The states correspond to the $n + 1$ distinct prefixes of z. Show that there can be no FA with fewer than $n + 1$ states accepting this language.

2.15. Suppose L is a subset of $\{a, b\}^*$. If $x_0, x_1, \ldots$ is a sequence of distinct strings in $\{a, b\}^*$ such that for every $n \geq 0$, x_n and x_{n+1} are L-distinguishable, does it follow that the strings $x_0, x_1, \ldots$ are pairwise L-distinguishable? Either give a proof that it does follow, or find an example of a language L and strings $x_0, x_1, \ldots$ that represent a counterexample.

2.16. Let $L \subseteq \{a, b\}^*$ 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 $L \subseteq \{a, b\}^*$ such that for every n satisfying $L_n \neq \emptyset$, $s(n)$ is this minimum number.

2.17. Let L be the language $AnBn = \{a^n b^n \mid n \geq 0\}$.

a. Find two distinct strings x and y in $\{a, b\}^*$ that are not L-distinguishable.

b. Find an infinite set of pairwise L-distinguishable strings.

2.18. Let n be a positive integer and $L = \{x \in \{a, b\}^* \mid |x| = n$ and $n_a(x) = n_b(x)\}$. What is the minimum number of states in any FA that accepts L? Give reasons for your answer.

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

$$L = \{xx^r \mid x \in \{a, b\}^n\}$$

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

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

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

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

b. Show that if the sequence n_i is bounded (i.e., there is a constant C such that $n_i \leq C$ for every i), then L can be accepted by an FA. Show in particular that if there is some fixed FA M that accepts L relative to L_i for every i, then M accepts L.

2.21. For each of the following languages $L \subseteq \{a, b\}^*$, show that the elements of the infinite set $\{a^n \mid n \geq 0\}$ are pairwise L-distinguishable.

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

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

c. $L = \{a^i b^j \mid j = i \text{ or } j = 2i\}$

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

e. $L = \{x \in \{a, b\}^* \mid n_a(x) < 2n_b(x)\}$

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

g. $L = \{a^{n^3} \mid n \geq 1\}$

h. $L = \{ww \mid w \in \{a, b\}^*\}$

2.22. For each of the languages in Exercise 2.21, use the pumping lemma to show that it cannot be accepted by an FA.

2.23. By ignoring some of the details in the statement of the pumping lemma, we can easily get these two weaker statements.

I. If $L \subseteq \Sigma^*$ is an infinite language that can be accepted by an FA, then there are strings u, v, and w such that $|v| > 0$ and $uv^i w \in L$ for every $i \geq 0$.

II. If $L \subseteq \Sigma^*$ is an infinite language that can be accepted by an FA, then there are integers p and q such that $q > 0$ and for every $i \geq 0$, L contains a string of length $p + iq$.

For each language L in Exercise 2.21, decide whether statement II is enough to show that L cannot be accepted by an FA, and explain your answer. If statement II is not sufficient, decide whether statement I is, and explain your answer.

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

a. $x_2 = uvw$

b. $|v| > 0$

c. For every $m \geq 0$, $x_1uv^mwx_3 \in L$

2.25. Find a language $L \subseteq \{a, b\}^*$ such that, in order to prove that L cannot be accepted by an FA, the pumping lemma is not sufficient but the statement in Exercise 2.24 is.

2.26. The pumping lemma says that if M accepts a language L, and if n is the number of states of M, then for every $x \in L$ satisfying $|x| \geq n$, ... Show that the statement provides no information if L is finite: If M accepts a finite language L, and n is the number of states of M, then L can contain no strings of length n or greater.

2.27. Describe decision algorithms to answer each of the following questions.

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

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

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

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

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

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

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

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

2.28. Suppose L is a language over $\{a, b\}$, and there is a fixed integer k such that for every $x \in \Sigma^*$, $xz \in L$ for some string z with $|z| \leq k$. Does it follow that there is an FA accepting L? Why or why not?

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

a. If $L_1 \subseteq L_2$, and L_1 cannot be accepted by an FA, then L_2 cannot.

b. If $L_1 \subseteq L_2$, and L_2 cannot be accepted by an FA, then L_1 cannot.

c. If neither L_1 nor L_2 can be accepted by an FA, then $L_1 \cup L_2$ cannot.

d. If neither L_1 nor L_2 can be accepted by an FA, then $L_1 \cap L_2$ cannot.

e. If L cannot be accepted by an FA, then L' cannot.

f. If L_1 can be accepted by an FA and L_2 cannot, then $L_1 \cup L_2$ cannot.

g. If L_1 can be accepted by an FA, L_2 cannot, and $L_1 \cap L_2$ can, then $L_1 \cup L_2$ cannot.

h. If L_1 can be accepted by an FA and neither L_2 nor $L_1 \cap L_2$ can, then $L_1 \cup L_2$ cannot.

i. If each of the languages $L_1, L_2, \dots$ can be accepted by an FA, then $\cup_{n=1}^{\infty} L_n$ can.

j. If none of the languages $L_1, L_2, \dots$ can be accepted by an FA, and $L_i \subseteq L_{i+1}$ for each i, then $\cup_{n=1}^{\infty} L_n$ cannot be accepted by an FA.

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

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

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

a. Show that if S is an arithmetic progression, then A can be accepted by an FA.

b. Show that if A can be accepted by an FA, then S is the union of a finite number of arithmetic progressions.

2.31. †This exercise involves languages of the form

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

for some function f from the set of natural numbers to itself. Example 2.30 shows that if f is the function defined by $f(n) = n$, then L cannot be accepted by an FA. If f is any constant function (e.g., $f(n) = 4$), there is an FA accepting L. One might ask whether this is still true when f is not restricted quite so severely.

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

b. Show that if $f(n) = n \bmod 2$, then L can be accepted by an FA.

c. The function f in part (b) is an *eventually periodic* function; that is, there are integers n_0 and p, with $p > 0$, such that for every $n \geq n_0$, $f(n) = f(n + p)$. Show that if f is any eventually periodic function, L can be accepted by an FA.

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

2.32. For which languages $L \subseteq \{a, b\}^*$ does the equivalence relation I_L have exactly one equivalence class?

2.33. Let x be a string of length n in $\{a, b\}^*$, and let $L = \{x\}$. How many equivalence classes does I_L have? Describe them.

2.34. Show that if $L \subseteq \Sigma^*$, and there is a string $x \in \Sigma^*$ that is not a prefix of an element of L, then the set of all strings that are not prefixes of elements of L is an infinite set that is one of the equivalence classes of I_L.

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

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

2.37. Suppose $L \subseteq \{a, b\}^*$ and I_L has three equivalence classes. Suppose they can be described as the three sets $[a]$, $[aa]$, and $[aaa]$, and also as the three sets $[b]$, $[bb]$, and $[bbb]$. How many possibilities are there for the language L? For each one, draw a transition diagram for an FA accepting it.

2.38. In each part, find every possible language $L \subseteq \{a, b\}^*$ for which the equivalence classes of I_L are the three given sets.

a. $\{a, b\}^*\{b\}$, $\{a, b\}^*\{ba\}$, $\{\Lambda, a\} \cup \{a, b\}^*\{aa\}$

b. $(\{a, b\}\{a\}^*\{b\})^*$, $(\{a, b\}\{a\}^*\{b\})^*\{a\}\{a\}^*$, $(\{a, b\}^*\{a\}^*\{b\})^*\{b\}\{a\}^*$

c. $\{\Lambda\}$, $\{a\}(\{b\} \cup \{a\}\{a\}^*\{b\})^*$, $\{b\}(\{a\} \cup \{b\}\{b\}^*\{a\})^*$

2.39. In Example 2.37, if the language is changed to $\{a^n b^n \mid n > 0\}$, so that it does not contain Λ, are there any changes in the partition of $\{a, b\}^*$ corresponding to I_L? Explain.

2.40. Consider the language $L = AEqB = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$.

a. Show that if $n_a(x) - n_b(x) = n_a(y) - n_b(y)$, then $x \ I_L \ y$.

b. Show that if $n_a(x) - n_b(x) \neq n_a(y) - n_b(y)$, then x and y are L-distinguishable.

c. Describe all the equivalence classes of I_L.

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

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

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

2.43. Let $L = \{ww \mid w \in \{a, b\}^*\}$. Describe all the equivalence classes of I_L.

2.44. Let L be the language *Balanced* of balanced strings of parentheses. Describe all the equivalence classes of I_L.

2.45. †Let L be the language of all fully parenthesized algebraic expressions involving the operator $+$ and the identifier a. (L can be defined recursively by saying that $a \in L$ and $(x + y) \in L$ for every x and y in L.) Describe all the equivalence classes of I_L.

2.46. †For a language L over Σ, and two strings x and y in Σ^* that are L-distinguishable, let

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

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

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

2.47. For an arbitrary string $x \in \{a, b\}^*$, denote by $x^\sim$ the string obtained by replacing all a's by b's and vice versa. For example, $\Lambda^\sim = \Lambda$ and $(abb)^\sim = baa$.

a. Define

$$L = \{xx^\sim \mid x \in \{a, b\}^*\}$$

Determine the equivalence classes of I_L.

b. Define

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

Determine the equivalence classes of I_{L_1}.

2.48. †Let $L = \{x \in \{a, b\}^* \mid n_b(x) \text{ is an integer multiple of } n_a(x)\}$. Determine the equivalence classes of I_L.

2.49. Let L be a language over Σ. We know that I_L is a *right-invariant* equivalence relation; i.e., for any x and y in Σ^* and any $a \in \Sigma$, if $x\ I_L\ y$, then $xa\ I_L\ ya$. It follows from Theorem 2.36 that if the set of equivalence classes of I_L is finite, L can be accepted by an FA, and in this case L is the union of some (zero or more) of these equivalence classes. Show that if R is *any* right-invariant equivalence relation such that the set of equivalence classes of R is finite and L is the union of some of the equivalence classes of R, then L can be accepted by an FA.

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

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

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

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

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

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

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

Show that if L_1 can be accepted by an FA and L_2 is any language, then L_1/L_2 can be accepted by an FA.

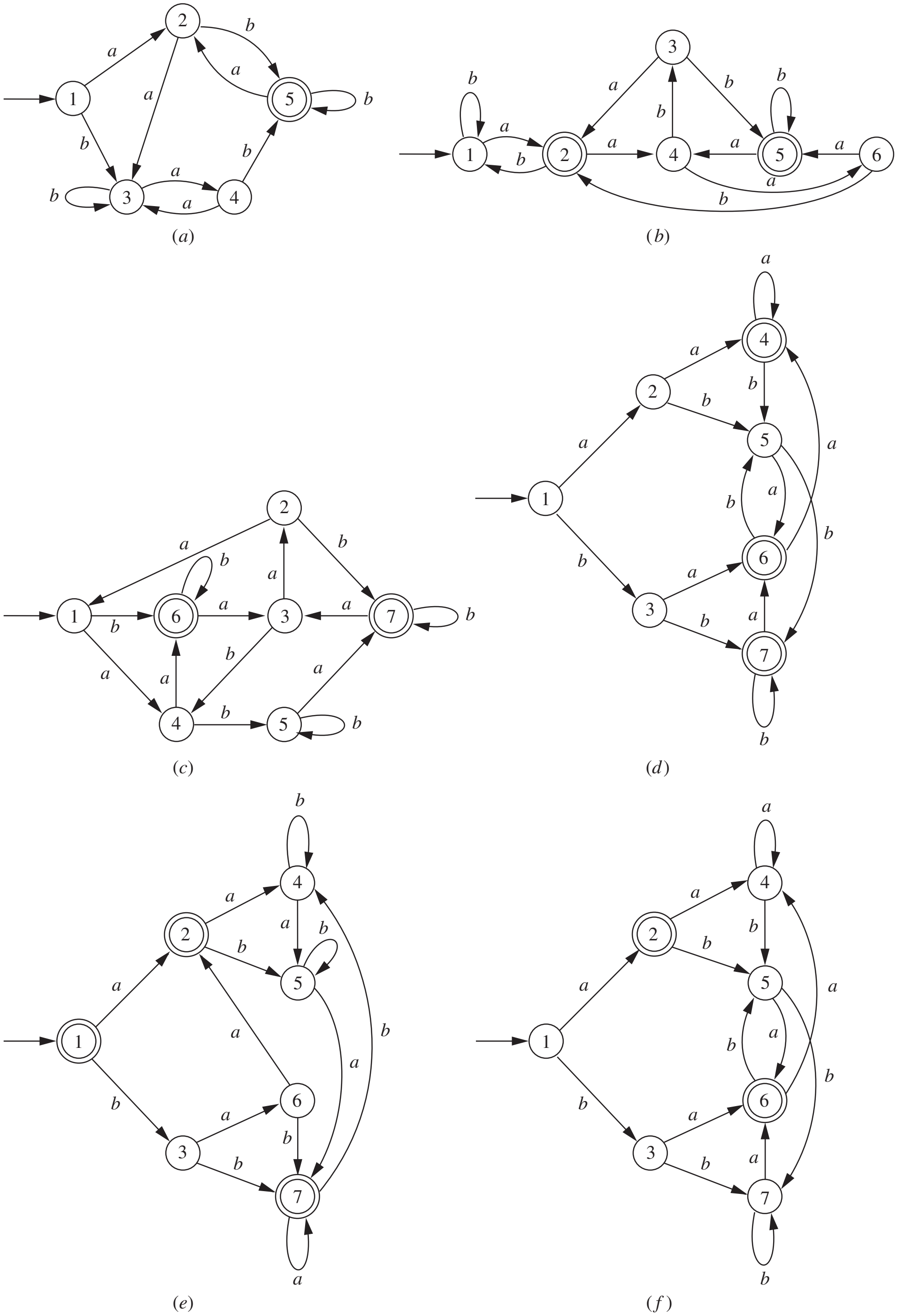
2.52. Suppose L is a language over Σ, and $x_1, x_2, \ldots, x_n$ are strings that are pairwise L-distinguishable. How many distinct strings are necessary in order to distinguish between the x_i's? In other words, what is the smallest number k such that for some set $\{z_1, z_2, \ldots, z_k\}$, any two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

2.53. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting L. We know that if $p, q \in Q$ and $p \not\equiv q$, then there is a string z such that exactly one of the two states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A. Show that there is an integer n such that for every p and q with $p \not\equiv q$, such a z can be found whose length is no greater than n, and say what n is.

2.54. Show that L can be accepted by an FA if and only if there is an integer n such that, for every pair of L-distinguishable strings, the two strings can be distinguished by a string of length $\leq n$. (Use the two previous exercises.)

2.55. For each of the FAs pictured in Fig. 2.45, use the minimization algorithm described in Section 2.6 to find a minimum-state FA recognizing the same language. (It's possible that the given FA may already be minimal.)

2.56. Suppose that in applying the minimization algorithm in Section 2.6, we establish some fixed order in which to process the pairs, and we follow the same order on each pass.



(a) (b) (c) (d) (e) (f)

Figure 2.45 |

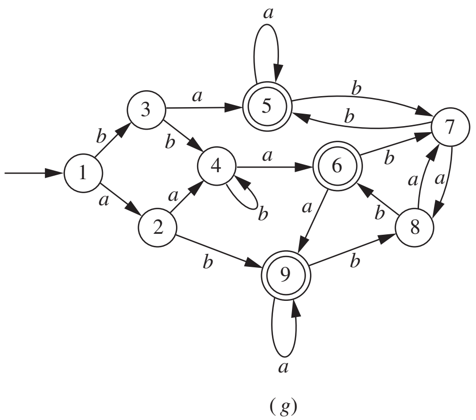


Figure 2.45 | Continued

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

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

2.57. Each case below defines a language over $\{a, b\}$. In each case, decide whether the language can be accepted by an FA, and prove that your answer is correct.

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

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

c. The set of all strings x having some nonnull substring of the form www. (You may assume the following fact: there are arbitrarily long strings in $\{a, b\}^*$ that do not contain any nonnull substring of the form www.)

d. The set of odd-length strings with middle symbol a.

e. The set of even-length strings of length at least 2 with the two middle symbols equal.

f. The set of strings of the form xyx for some x with $|x| \geq 1$.

g. The set of non-palindromes.

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

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

j. The set of strings having the property that in some prefix, the number of a's is 3 more than the number of b's.

k. The set of strings in which the number of a's and the number of b's are both divisible by 5.

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

m. (Assuming that L can be accepted by an FA), $Max(L) = \{x \in L \mid$ there is no nonnull string y so that $xy \in L\}$.

n. (Assuming that L can be accepted by an FA), $Min(L) = \{x \in L \mid$ no prefix of x other than x itself is in $L\}$.

2.58. Find an example of a language $L \subseteq \{a, b\}^*$ such that L^* cannot be accepted by an FA.

2.59. Find an example of a language L over $\{a, b\}$ such that L cannot be accepted by an FA but L^* can.

2.60. Find an example of a language L over $\{a, b\}$ such that L cannot be accepted by an FA but LL can.

2.61. †Show that if L is any language over a one-symbol alphabet, then L^* can be accepted by an FA.

2.62. †Consider the two FAs in Fig. 2.46.
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 transitions among the states of the first FA are the same as those among the corresponding states of the other. For example, if δ_1 and δ_2 are the transition functions, then

$$\delta_1(p, a) = p \text{ and } \delta_2(i(p), a) = i(p)$$
$$\delta_1(p, b) = q \text{ and } \delta_2(i(p), b) = i(q)$$

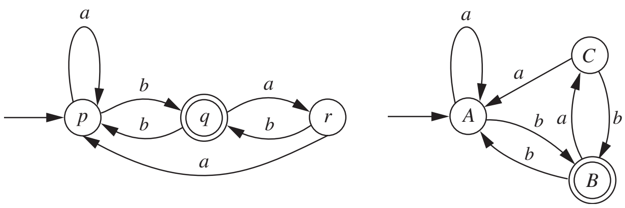


Figure 2.46 |

These formulas can be rewritten

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

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

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

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 every $q \in Q_1$, $i(q) \in A_2$ if and only if $q \in A_1$

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

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 Σ, 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 $\{a, b\}$ are there, no two of which are isomorphic?

e. How many pairwise nonisomorphic two-state FAs over $\{a, b\}$ 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	$\delta_1(q, a)$	$\delta_1(q, b)$
1	3	5
2	4	2
3	1	6
4	4	3
5	2	4
6	3	4

q	$\delta_2(q, a)$	$\delta_2(q, b)$
A	B	E
B	A	D
C	C	B
D	B	C
E	F	C
F	C	F

2.63. Suppose that $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are both FAs accepting the language L, and that both have as few states as possible. Show that M_1 and M_2 are isomorphic (see Exercise 2.62). Note that in both cases, the sets L_q forming the partition of Σ^* are precisely the equivalence classes of I_L. This tells you how to come up with a bijection from Q_1 to Q_2. What you must do next is to show that the other conditions of an isomorphism are satisfied.

2.64. Use Exercise 2.63 to describe another decision algorithm to answer the question "Given two FAs, do they accept the same language?"

CHAPTER

3

Regular Expressions, Nondeterminism, and Kleene's Theorem

A simple way of describing a language is to describe a finite automaton that accepts it. As with the models of computation we will study later, an alternative approach is to use some appropriate notation to describe how the strings of the language can be generated. Languages that can be accepted by finite automata are the same as *regular* languages, which can be represented by formulas called regular expressions involving the operations of union, concatenation, and Kleene star. In the case of finite automata, demonstrating this equivalence (by proving the two parts of Kleene's theorem) is simplified considerably by introducing *nondeterminism*, which will also play a part in the computational models we will study later. Here, although allowing nondeterminism seems at first to enhance the accepting power of these devices, we will see that it can be eliminated.

3.1 REGULAR LANGUAGES AND REGULAR EXPRESSIONS

Three of the languages over $\{a, b\}$ that we considered in Chapter 2 are L_1, the language of strings ending in aa; L_2, the language of strings containing either the substring ab or the substring bba; and L_3, the language $\{aa, aab\}^*\{b\}$. Like L_3, both L_1 and L_2 can be expressed by a formula involving the operations of union, concatenation, and Kleene *: L_1 is $\{a, b\}^*\{aa\}$ and L_2 is $\{a, b\}^*(\{ab\} \cup \{bba\})\{a, b\}^*$. Languages that have formulas like these are called *regular* languages. In this section we give a recursive definition of the set of regular languages over an alphabet Σ, and later in this chapter we show that these are precisely the languages that can be accepted by a finite automaton.

Definition 3.1 Regular Languages over an Alphabet Σ

If Σ is an alphabet, the set $\mathcal{R}$ of regular languages over Σ is defined as follows.

1. The language $\emptyset$ is an element of $\mathcal{R}$, and for every $a \in \Sigma$, the language $\{a\}$ is in $\mathcal{R}$.
2. For any two languages L_1 and L_2 in $\mathcal{R}$, the three languages

$$L_1 \cup L_2, \ L_1 L_2, \ \text{ and } L_1^*$$

are elements of $\mathcal{R}$.

The language $\{\Lambda\}$ is a regular language over Σ, because $\emptyset^* = \{\Lambda\}$. If $\Sigma = \{a, b\}$, then $L_1 = \{a, b\}^*\{aa\}$ can be obtained from the definition by starting with the two languages $\{a\}$ and $\{b\}$ and then using the recursive statement in the definition four times: The language $\{a, b\}$ is the union $\{a\} \cup \{b\}$; $\{aa\}$ is the concatenation $\{a\}\{a\}$; $\{a, b\}^*$ is obtained by applying the Kleene star operation to $\{a, b\}$; and the final language is the concatenation of $\{a, b\}^*$ and $\{aa\}$.

A regular language over Σ has an explicit formula. A *regular expression* for the language is a slightly more user-friendly formula. The only differences are that in a regular expression, parentheses replace {} and are omitted whenever the rules of precedence allow it, and the union symbol $\cup$ is replaced by $+$. Here are a few examples (see Example 3.5 for a discussion of the last one):

Regular Language	***Corresponding Regular Expression***
$\emptyset$	$\emptyset$
$\{\Lambda\}$	Λ
$\{a, b\}^*$	$(a + b)^*$
$\{aab\}^*\{a, ab\}$	$(aab)^*(a + ab)$
$(\{aa, bb\} \cup \{ab, ba\}\{aa, bb\}^*\{ab, ba\})^*$	$(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$

When we write a regular expression like Λ or aab, which contains neither $+$ nor * and corresponds to a one-element language, the regular expression looks just like the string it represents. A more general regular expression involving one or both of these operations can't be mistaken for a string; we can think of it as representing the general form of strings in the language. A regular expression describes a regular language, and a regular language can be described by a regular expression.

We say that two regular expressions are equal if the languages they describe are equal. Some regular-expression identities are more obvious than others. The formula

$$(a^*b^*)^* = (a + b)^*$$

is true because the language corresponding to a^*b^* contains both a and b. The formula

$$(a+b)^*ab(a+b)^* + b^*a^* = (a+b)^*$$

is true because the first term on the left side corresponds to the strings in $\{a, b\}^*$ that contain the substring ab and the second term, b^*a^*, corresponds to the strings that don't.

EXAMPLE 3.2 The Language of Strings in $\{a, b\}^*$ with an Odd Number of a's

A string with an odd number of a's has at least one a, and the additional a's can be grouped into pairs. There can be arbitrarily many b's before the first a, between any two consecutive a's, and after the last a. The expression

$$b^*ab^*(ab^*a)^*b^*$$

is not correct, because it doesn't allow b's between the second a in one of the repeating pairs ab^*a and the first a in the next pair. One correct regular expression describing the language is

$$b^*ab^*(ab^*ab^*)^*$$

The expression

$$b^*a(b^*ab^*ab^*)^*$$

is also not correct, because it doesn't allow strings with just one a to end with b, and the expression

$$b^*a(b^*ab^*a)^*b^*$$

corrects the mistake. Another correct expression is

$$b^*a(b+ab^*a)^*$$

All of these could also be written with the single a on the right, as in

$$(b+ab^*a)^*ab^*$$

EXAMPLE 3.3 The Language of Strings in $\{a, b\}^*$ Ending with b and Not Containing aa

If a string does not contain the substring aa, then every a in the string either is followed immediately by b or is the last symbol in the string. If the string ends with b, then every a is followed immediately by b. Therefore, every string in the language L of strings that end with b and do not contain aa matches the regular expression $(b+ab)^*$. This regular expression does not describe L, however, because it allows the null string, which does not end with b. At least one of the two strings b and ab must occur, and so a regular expression for L is

$$(b+ab)^*(b+ab)$$

Strings in $\{a,b\}^*$ in Which Both the Number of a's and the Number of b's Are Even

EXAMPLE 3.4

One of the regular expressions given in Example 3.2, $b^*a(b + ab^*a)^*$, describes the language of strings with an odd number of a's, and the final portion of it, $(b + ab^*a)^*$, describes the language of strings with an even number of a's. We can interpret the two terms inside the parentheses as representing the two possible ways of adding to the string without changing the parity (the evenness or oddness) of the number of a's: adding a string that has no a's, and adding a string that has two a's. Every string x with an even number of a's has a prefix matching one of these two terms, and x can be decomposed into nonoverlapping substrings that match one of these terms.

Let L be the subset of $\{a, b\}^*$ containing the strings x for which both $n_a(x)$ and $n_b(x)$ are even. Every element of L has even length. We can use the same approach to find a regular expression for L, but this time it's sufficient to consider substrings of even length. The easiest way to add a string of even length without changing the parity of the number of a's or the number of b's is to add aa or bb. If a nonnull string $x \in L$ does not begin with one of these, then it starts with either ab or ba, and the shortest substring following this that restores the evenness of n_a and n_b must also end with ab or ba, because its length is even and strings of the form aa and bb don't change the parity of n_a or n_b.

The conclusion is that every nonnull string in L has a prefix that matches the regular expression

$$aa + bb + (ab + ba)(aa + bb)^*(ab + ba)$$

and that a regular expression for L is

$$(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

Regular Expressions and Programming Languages

EXAMPLE 3.5

In Example 2.9 we built a finite automaton to carry out a very simple version of lexical analysis: breaking up a part of a computer program into tokens, which are the basic building blocks from which the expressions or statements are constructed. The last two sections of this chapter are devoted to proving that finite automata can accept exactly the same languages that regular expressions can describe, and in this example we construct regular expressions for two classes of tokens.

An identifier in the C programming language is a string of length 1 or more that contains only letters, digits, and underscores ("_") and does not begin with a digit. If we use the abbreviations l for "letter," either uppercase or lowercase, and d for "digit," then l stands for the regular expression

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

and d for the regular expression

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

(which has nothing to do with the integer 45), and a regular expression for the language of C identifiers is

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

Next we look for a regular expression to describe the language of numeric "literals," which typically includes strings such as 14, +1, −12, 14.3, −.99, 16., 3E14, −1.00E2, 4.1E−1, and .3E+2. Let us assume that such an expression may or may not begin with a plus sign or a minus sign; it will contain one or more decimal digits, and possibly a decimal point, and it may or may not end with a subexpression starting with E. If there is such a subexpression, the portion after E may or may not begin with a sign and will contain one or more decimal digits.

Our regular expression will involve the abbreviations d and l introduced above, and we will use s to stand for "sign" (either Λ or a plus sign or a minus sign) and p for a decimal point. It is not hard to convince yourself that a regular expression covering all the possibilities is

$$s(dd^*(\Lambda + pd^*) + pdd^*)(\Lambda + \mathrm{E}sdd^*)$$

In some programming languages, numeric literals are not allowed to contain a decimal point unless there is at least one digit on both sides. A regular expression incorporating this requirement is

$$sdd^*(\Lambda + pdd^*)(\Lambda + \mathrm{E}sdd^*)$$

Other tokens in a high-level language can also be described by regular expressions, in most cases even simpler than the ones in this example. Lexical analysis is the first phase in compiling a high-level-language program. There are programs called lexical-analyzer generators; the input provided to such a program is a set of regular expressions describing the structure of tokens, and the output produced by the program is a software version of an FA that can be incorporated as a token-recognizing module in a compiler. One of the most widely used lexical-analyzer generators is `lex`, a tool provided in the Unix operating system. It can be used in many situations that require the processing of structured input, but it is often used together with `yacc`, another Unix tool. The lexical analyzer produced by `lex` creates a string of tokens; and the *parser* produced by `yacc`, on the basis of grammar rules provided as input, is able to determine the syntactic structure of the token string. (`yacc` stands for *yet another compiler compiler*.)

Regular expressions come up in Unix in other ways as well. The Unix text editor allows the user to specify a regular expression and searches for patterns in the text that match it. Other commands such as `grep` (global regular expression print) and `egrep` (extended global regular expression print) allow a user to search a file for strings that match a specified regular expression.

3.2 NONDETERMINISTIC FINITE AUTOMATA

The goal in the rest this chapter is to prove that regular languages, defined in Section 3.1, are precisely the languages accepted by finite automata. In order to do this, we will introduce a more general "device," a *nondeterministic* finite automaton. The advantage of this approach is that it's much easier to start with an arbitrary regular expression and draw a transition diagram for something that accepts the corresponding language and has an obvious connection to the regular expression. The only problem is that the *something* might not be a finite automaton, although it has a superficial resemblance to one, and we have to figure out how to interpret it in order to think of it as a physical device at all.

Accepting the Language $\{aa, aab\}^*\{b\}$) EXAMPLE 3.6

There is a close resemblance between the diagram in Figure 3.7 and the regular expression $(aa + aab)^*b$. The top loop corresponds to aa, the bottom one corresponds to aab, and the remaining b-transition corresponds to the last b in the regular expression. To the extent that we think of it as a transition diagram like that of an FA, its resemblance to the regular expression suggests that the string *aaaabaab*, for example, should be accepted, because it allows us to start at q_0, take the top loop once, the bottom loop once, the top loop again, and finish up with the transition to the accepting state.

This diagram, however, is not the transition diagram for an FA, because there are three transitions from q_0 and fewer than two from several other states. The input string *aaaabaab allows* us to reach the accepting state, but it also allows us to follow, or at least start to follow, other paths that don't result in acceptance. We can imagine an idealized "device" that would work by using the input symbols to follow paths shown in the diagram, making arbitrary choices at certain points. It has to be *nondeterministic*, in the sense that the path it follows is not determined by the input string. (If the first input symbol is a, it chooses whether to start up the top path or down the bottom one.) Even if there were a way to build a physical device that acted like this, it wouldn't accept this language in the same way that an FA could. Suppose we watched it process a string x. If it ended up in the accepting state at the end, we could say that x was in the language; if it didn't, all we could say for sure is that the moves it chose to make did not lead to the accepting state.

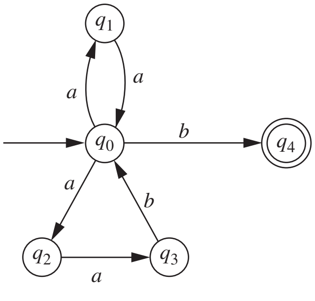


Figure 3.7 |
Using nondeterminism to accept $\{aa,aab\}^*\{b\}$.

Allowing nondeterminism, by relaxing the rules for an FA, makes it easy to draw diagrams corresponding to regular expressions. However, we should no longer think of the diagram as representing an explicit algorithm for accepting the language, because an algorithm refers to a sequence of steps that are determined by the input and would be the same no matter how many times the algorithm was executed on the same input.

If the diagram doesn't represent an explicit accepting algorithm, what good is it? One way to answer this is to think of the diagram as describing a number of different sequences of steps that might be followed. We can visualize these sequences for the input string *aaaabaab* by drawing a *computation tree*, pictured in Figure 3.8.

A level of the tree corresponds to the input (the prefix of the entire input string) read so far, and the states appearing on this level are those in which the device *could* be, depending on the choices it has made so far. Two paths in the tree, such as the one that

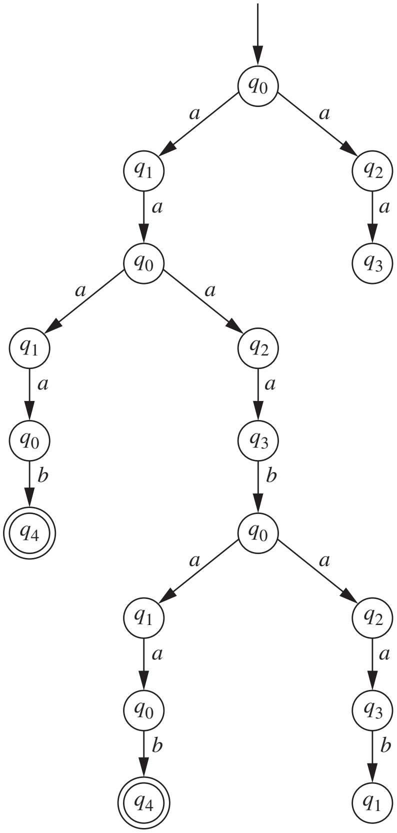


Figure 3.8 |
The computation tree for Figure 3.7 and the input string *aaaabaab*.

starts by treating the initial a as the first symbol of aab, terminate prematurely, because the next input symbol does not allow a move from the current state. One path, which corresponds to interpreting the input string as $(aa)(aab)(aab)$, allows all the input to be read and ends up in a nonaccepting state. The path in which the device makes the "correct" choice at each step ends up at the accepting state when all the input symbols have been read.

If we had the transition diagram in Figure 3.7 and were trying to use it to accept the language, we could systematically keep track of the current sequence of steps, and use a backtracking strategy whenever we couldn't proceed any further or finished in a nonaccepting state. The result would, in effect, be to search the computation tree using a depth-first search. In the next section we will see how to develop an ordinary finite automaton that effectively executes a breadth-first search of the tree, by keeping track after each input symbol of all the possible states the various sequences of steps could have led us to.

Accepting the Language $\{aab\}^*\{a,aba\}^*$ **EXAMPLE 3.9**

In this example we consider the regular expression $(aab)^*(a + aba)^*$. The techniques of Example 3.6 don't provide a simple way to draw a transition diagram related to the regular expression, but Figure 3.10 illustrates another type of nondeterminism that does.

The new feature is a "Λ-transition," which allows the device to change state with no input. If the input a is received in state 0, there are three options: take the transition from state 0 corresponding to the a in aab; move to state 3 and take the transition corresponding to a; and move to state 3 and take the transition corresponding to the a in aba. The diagram shows two a-transitions from state 3, but because of the Λ-transition, we would have a choice of moves even if there were only one.

Figure 3.11 shows a computation tree illustrating the possible sequences of moves for the input string $aababa$. The Λ-transition is drawn as a horizontal arrow, so that as in the previous example, a new level of the tree corresponds to a new input symbol.

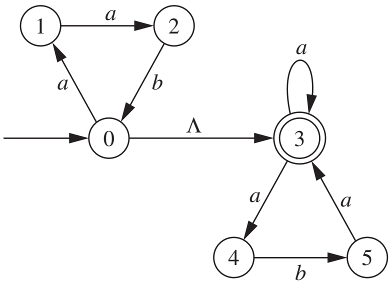


Figure 3.10 | Using nondeterminism to accept $\{aab\}^*\{a, aba\}^*$.

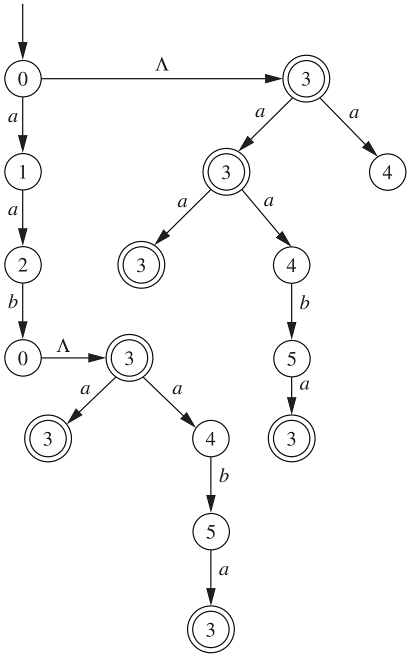


Figure 3.11 |
The computation tree for Figure 3.10 and the input string *aababa*.

The string is accepted, because the device can choose to take the first loop, execute the Λ-transition, and take the longer loop from state 3.

The transition diagrams in our first two examples show four of the five ingredients of an ordinary finite automaton. The one that must be handled differently is the transition function δ. For a state q and an alphabet symbol σ, it is no longer correct to say that $\delta(q, \sigma)$ is a state: There may be no transitions from state q on input σ, or one, or more than one. There may also be Λ-transitions. We can incorporate both of these features by making two changes: first, enlarging the domain of δ to include ordered pairs (q, Λ) as well as the pairs in $Q \times \Sigma$; and second, making the values of δ *sets* of states instead of individual states.

Definition 3.12 A Nondeterministic Finite Automaton

A *nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where

Q is a finite set of states;

Σ is a finite input alphabet;

$q_0 \in Q$ is the initial state;
$A \subseteq Q$ is the set of accepting states;
$\delta : Q \times (\Sigma \cup \{\Lambda\}) \to 2^Q$ is the transition function.

For every element q of Q and every element σ of $\Sigma \cup \{\Lambda\}$, we interpret $\delta(q, \sigma)$ as the set of states to which the FA can move, if it is in state q and receives the input σ, or, if $\sigma = \Lambda$, the set of states other than q to which the NFA can move from state q without receiving any input symbol.

In Example 3.9, for example, $\delta(0, a) = \{1\}$, $\delta(0, \Lambda) = \{3\}$, $\delta(0, b) = \emptyset$, and $\delta(0, a) = \{3, 4\}$.

In the case of an NFA $M = (Q, \Sigma, q_0, A, \delta)$, we want $\delta^*(q, x)$ to tell us all the states M can get to by starting at q and using the symbols in the string x. We can still define the function δ^* recursively, but the mathematical notation required to express this precisely is a little more involved, particularly if M has Λ-transitions.

In order to define $\delta^*(q, x\sigma)$, where $x \in \Sigma^*$ and $\sigma \in \Sigma$, we start by considering $\delta^*(q, x)$, just as in the simple case of an ordinary FA. This is now a set of states, and for each state p in this set, $\delta(p, \sigma)$ is itself a set. In order to include all the possibilities, we need to consider

$$\bigcup\{\delta(p, a) \mid p \in \delta^*(q, x)\}$$

Finally, we must keep in mind that in the case of Λ-transitions, "using all the symbols in the string x" really means using all the symbols in x and perhaps Λ-transitions where they are possible. In the recursive step of the definition of δ^*, once we have the union we have just described, we must consider all the additional states we might be able to reach from elements of this union, using nothing but Λ-transitions.

You can probably see at this point how the following definition will be helpful in our discussion.

Definition 3.13 The Λ-Closure of a Set of States

Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA, and $S \subseteq Q$ is a set of states. The Λ-closure of S is the set $\Lambda(S)$ that can be defined recursively as follows.

1. $S \subseteq \Lambda(S)$.
2. For every $q \in \Lambda(S)$, $\delta(q, \Lambda) \subseteq \Lambda(S)$.

In exactly the same way as in Example 1.21, we can convert the recursive definition of $\Lambda(S)$ into an algorithm for evaluating it, as follows.

Algorithm to Calculate $\Lambda(S)$ Initialize T to be S. Make a sequence of passes, in each pass considering every $q \in T$ and adding to T every state in $\delta(q, \Lambda)$ that is not already an element. Stop after the first pass in which T is not changed. The final value of T is $\Lambda(S)$. ■

A state is in $\Lambda(S)$ if it is an element of S or can be reached from an element of S using one or more Λ-transitions.

With the help of Definition 3.13 we can now define the extended transition function δ^* for a nondeterministic finite automaton.

Definition 3.14 The Extended Transition Function δ^* for an NFA, and the Definition of Acceptance

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

$$\delta^* : Q \times \Sigma^* \to 2^Q$$

as follows:

1. For every $q \in Q$, $\delta^*(q, \Lambda) = \Lambda(\{q\})$.
2. For every $q \in Q$, every $y \in \Sigma^*$, and every $\sigma \in \Sigma$,

$$\delta^*(q, y\sigma) = \Lambda\left(\bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, y)\}\right)$$

A string $x \in \Sigma^*$ is accepted by M if $\delta^*(q_0, x) \cap A \neq \emptyset$. The language $L(M)$ accepted by M is the set of all strings accepted by M.

For the NFA in Example 3.9, which has only one Λ-transition, it is easy to evaluate $\delta^*(aababa)$ by looking at the computation tree in Figure 3.11. The two states on the first level of the diagram are 0 and 3, the elements of the Λ-closure of $\{0\}$. The states on the third level, for example, are 2, 3, and 4, because $\delta^*(0, aa) = \{2, 3, 4\}$. When we apply the recursive part of the definition to evaluate $\delta^*(0, aab)$, we first evaluate

$$\begin{aligned}\bigcup\{\delta(p, b) \mid p \in \{2, 3, 4\}\} &= \delta(2, b) \cup \delta(3, b) \cup \delta(4, b) = \{0\} \cup \emptyset \cup \{5\} \\ &= \{0, 5\}\end{aligned}$$

and then we compute the Λ-closure of this set, which contains the additional element 3.

For an NFA M with no Λ-transitions, both statements in the definition can be simplified, because for every subset S of Q, $\Lambda(S) = S$.

We illustrate the definition once more in a slightly more extended example.

EXAMPLE 3.15 Applying the Definitions of $\Lambda(S)$ and δ^*

We start by evaluating the Λ-closure of the set $\{v\}$ in the NFA whose transition diagram is shown in Figure 3.16. When we apply the algorithm derived from Definition 3.13, after one

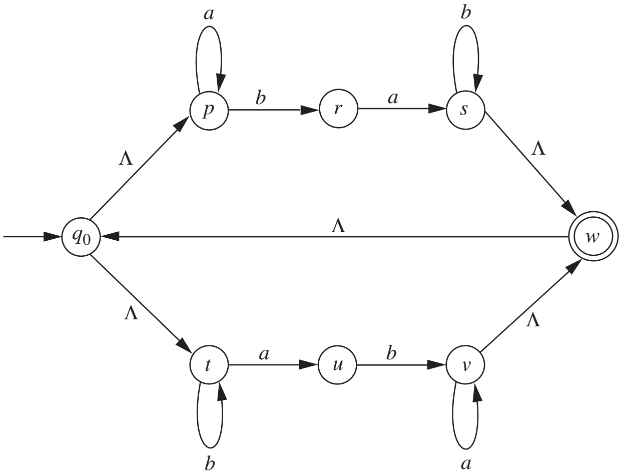


Figure 3.16 |
Evaluating the extended transition function when there are Λ-transitions.

pass T is $\{v, w\}$, after two passes it is $\{v, w, q_0\}$, after three passes it is $\{v, w, q_0, p, t\}$, and during the next pass it remains unchanged. The set $\Lambda(\{s\})$ is therefore $\{v, w, q_0, p, t\}$.

If we want to apply the definition of δ^* to evaluate $\delta^*(q_0, aba)$, the easiest way is to begin with Λ, the shortest prefix of aba, and work our way up one symbol at a time.

$$\begin{aligned}
\delta^*(q_0, \Lambda) &= \Lambda(\{q_0\}) \\
&= \{q_0, p, t\} \\
\delta^*(q_0, a) &= \Lambda\left(\bigcup\{\delta(k, a) \mid k \in \delta^*(q_0, \Lambda)\}\right) \\
&= \Lambda\,(\delta(q_0, a) \cup \delta(p, a) \cup \delta(t, a)) \\
&= \Lambda\,(\emptyset \cup \{p\} \cup \{u\}) \\
&= \Lambda(\{p, u\}) \\
&= \{p, u\} \\
\delta^*(q_0, ab) &= \Lambda\left(\bigcup\{\delta(k, b) \mid k \in \{p, u\}\}\right) \\
&= \Lambda(\delta(p, b) \cup \delta(u, b)) \\
&= \Lambda(\{r, v\}) \\
&= \{r, v, w, q_0, p, t\} \\
\delta^*(q_0, aba) &= \Lambda\left(\bigcup\{\delta(k, a) \mid k \in \{r, v, w, q_0, p, t\}\}\right) \\
&= \Lambda(\delta(r, a) \cup \delta(v, a) \cup \delta(w, a) \cup \delta(q_0, a) \cup \delta(p, a) \cup \delta(t, a)) \\
&= \Lambda(\{s\} \cup \{v\} \cup \emptyset \cup \emptyset \cup \{p\} \cup \{u\}) \\
&= \Lambda(\{s, v, p, u\}) \\
&= \{s, v, p, u, w, q_0, t\}
\end{aligned}$$

The evaluation of $\Lambda(\{r, v\})$ is very similar to that of $\Lambda(\{v\})$, since there are no Λ-transitions from r, and the evaluation of $\Lambda(\{s, v, p, u\})$ is also similar. Because $\delta^*(q_0, aba)$ contains the accepting state w, the string aba is accepted.

A state r is an element of $\delta^*(q, x)$ if in the transition diagram there is a path from q to r, in which there are transitions for every symbol in x and the next transition at each step corresponds either to the next symbol in x or to Λ. In simple examples, including this one, you may feel that it's easier to evaluate δ^* by looking at the diagram and determining by inspection what states you can get to. One reason for having a precise recursive definition of δ^* and a systematic algorithm for evaluating it is that otherwise it's easy to overlook things.

3.3 THE NONDETERMINISM IN AN NFA CAN BE ELIMINATED

We have observed nondeterminism in two slightly different forms in our discussion of NFAs. It is most apparent if there is a state q and an alphabet symbol σ such that several different transitions are possible in state q on input σ. A choice of moves can also occur as a result of Λ-transitions, because there may be states from which the NFA can make either a transition on an input symbol or one on no input.

We will see in this section that both types of nondeterminism can be eliminated. The idea in the second case is to introduce new transitions so that we no longer need Λ-transitions: In every case where there is no σ-transition from p to q but the NFA can go from p to q by using one or more Λ's as well as σ, we will introduce the σ-transition. The resulting NFA may have even more nondeterminism of the first type than before, but it will be able to accept the same strings without using Λ-transitions.

The way we eliminate nondeterminism from an NFA having no Λ-transitions is simply to define it away, by finding an appropriate definition of *state*. We have used this technique twice before, in Section 2.2 when we considered states that were ordered pairs, and in Section 2.5 when we defined a state to be a set of strings. Here a similar approach is already suggested by the way we define the transition function of an NFA, whose value is a set of states. If we say that for an element p of a set $S \subseteq Q$, the transition on input σ can possibly go to several states, it sounds like nondeterminism; if we say that starting with an element of the set S, the set of states to which we can go on input σ is

$$\bigcup\{\delta(p, \sigma) \mid p \in S\}$$

and if both S and this set qualify as states in our new definition, then it sounds as though we have eliminated the nondeterminism. The only question then is whether the FA we obtain accepts the same strings as the NFA we started with.

Theorem 3.17

For every language $L \subseteq \Sigma^*$ accepted by an NFA $M = (Q, \Sigma, q_0, A, \delta)$, there is an NFA M_1 with no Λ-transitions that also accepts L.

Proof

As we have already mentioned, we may need to add transitions in order to guarantee that the same strings will be accepted even when the Λ-transitions are eliminated. In addition, if $q_0 \notin A$ but $\Lambda \in L$, we will also make q_0 an accepting state of M_1 in order to guarantee that M_1 accepts Λ.

We define

$$M_1 = (Q, \Sigma, q_0, A_1, \delta_1)$$

where for every $q \in Q$, $\delta_1(q, \Lambda) = \emptyset$, and for every $q \in Q$ and every $\sigma \in \Sigma$,

$$\delta_1(q, \sigma) = \delta^*(q, \sigma)$$

Finally, we define

$$A_1 = \begin{cases} A \cup \{q_0\} & \text{if } \Lambda \in L \\ A & \text{if not} \end{cases}$$

For every state q and every $x \in \Sigma^*$, the way we have defined the extended transition function δ^* for the NFA M tells us that $\delta^*(q, x)$ is the set of states M can reach by using the symbols of x together with Λ-transitions. The point of our definition of δ_1 is that we want $\delta_1^*(q, x)$ to be the same set, even though M_1 has no Λ-transitions. This may not be true for $x = \Lambda$, because $\delta^*(q, \Lambda) = \Lambda(\{q\})$ and $\delta_1(q, \Lambda) = \{q\}$; this is the reason for the definition of A_1 above. We sketch the proof that for every q and every x with $|x| \geq 1$,

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

The proof is by structural induction on x. If $x = a \in \Sigma$, then by definition of δ_1, $\delta_1(q, x) = \delta^*(q, x)$, and because M_1 has no Λ-transitions, $\delta_1(q, x) = \delta_1^*(q, x)$ (see Exercise 3.24).

Suppose that for some y with $|y| \geq 1$, $\delta_1^*(q, y) = \delta^*(q, y)$ for every state q, and let σ be an arbitrary element of Σ.

$$\begin{aligned} \delta_1^*(q, y\sigma) &= \bigcup\{\delta_1(p, \sigma) \mid p \in \delta_1^*(q, y)\} \\ &= \bigcup\{\delta_1(p, \sigma) \mid p \in \delta^*(q, y)\} \text{ (by the induction hypothesis)} \\ &= \bigcup\{\delta^*(p, \sigma) \mid p \in \delta^*(q, y)\} \text{ (by definition of } \delta_1) \end{aligned}$$

The last step in the induction proof is to check that this last expression is indeed $\delta^*(q, y\sigma)$. This is a special case of the general formula

$$\delta^*(q, yz) = \bigcup\{\delta^*(p, z) \mid p \in \delta^*(q, y)\}$$

See Exercise 3.30 for the details.

Now we can verify that $L(M_1) = L(M) = L$. If the string Λ is accepted by M, then it is accepted by M_1, because in this case $q_0 \in A_1$ by definition. If $\Lambda \notin L(M)$, then $A = A_1$; therefore, $q_0 \notin A_1$, and $\Lambda \notin L(M_1)$.

Suppose that $|x| \geq 1$. If $x \in L(M)$, then $\delta^*(q_0, x)$ contains an element of A; therefore, since $\delta^*(q_0, x) = \delta_1^*(q_0, x)$ and $A \subseteq A_1$, $x \in L(M_1)$.

Now suppose $|x| \geq 1$ and $x \in L(M_1)$. Then $\delta_1^*(q_0, x)$ contains an element of A_1. The state q_0 is in A_1 only if $\Lambda \in L$; therefore, if $\delta_1^*(q_0, x)$ (which is the same as $\delta^*(q_0, x)$) contains q_0, it also contains every element of A in $\Lambda(\{q_0\})$. In any case, if $x \in L(M_1)$, then $\delta_1^*(q_0, x)$ must contain an element of A, which implies that $x \in L(M)$.

Theorem 3.18

For every language $L \subseteq \Sigma^*$ accepted by an NFA $M = (Q, \Sigma, q_0, A, \delta)$, there is an FA $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ that also accepts L.

Proof

Because of Theorem 3.17, it is sufficient to prove the theorem in the case when M has no Λ-transitions. The formulas defining δ^* are simplified accordingly: $\delta^*(q, \Lambda) = \{q\}$ and $\delta^*(q, x\sigma) = \cup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\}$.

The finite automaton M_1 can be defined as follows, using the *subset construction*: The states of M_1 are sets of states of M, or

$$Q_1 = 2^Q$$

The initial state q_1 of Q_1 is $\{q_0\}$. For every $q \in Q_1$ and every $\sigma \in \Sigma$,

$$\delta_1(q, \sigma) = \bigcup\{\delta(p, \sigma) \mid p \in q\}$$

and the accepting states of M_1 are defined by the formula

$$A_1 = \{q \in Q_1 \mid q \cap A \neq \emptyset\}$$

The last definition is the correct one, because a string x should be accepted by M_1 if, when the NFA M processes x, there is at least one state it might end up in that is an element of A.

There is no doubt that M_1 is an ordinary finite automaton. The expression $\delta_1^*(q_1, x)$, however, is a set of states of M—not because M_1 is nondeterministic, but because we have defined states of M_1 to be sets of states of M. The fact that the two devices accept the same language follows from the fact that for every $x \in \Sigma^*$,

$$\delta_1^*(q_1, x) = \delta^*(q_0, x)$$

and we now prove this formula using structural induction on x. We must keep in mind during the proof that δ_1^* and δ^* are defined in different ways, because M_1 is an FA and M is an NFA.

If $x = \Lambda$, then

$$\begin{aligned} \delta_1^*(q_1, x) &= \delta_1^*(q_1, \Lambda) \\ &= q_1 \text{ (by the definition of } \delta_1^*) \end{aligned}$$

$$\begin{aligned} &= \{q_0\} \text{ (by the definition of } q_1) \\ &= \delta^*(q_0, \Lambda) \text{ (by the definition of } \delta^*) \\ &= \delta^*(q_0, x) \end{aligned}$$

The induction hypothesis is that x is a string for which $\delta_1^*(q_1, x) = \delta^*(q_0, x)$, and we must show that for every $\sigma \in \Sigma$, $\delta_1^*(q_1, x\sigma) = \delta^*(q_0, x\sigma)$.

$$\begin{aligned} \delta_1^*(q_1, x\sigma) &= \delta_1(\delta_1^*(q_1, x), \sigma) \text{ (by the definition of } \delta_1^*) \\ &= \delta_1(\delta^*(q_0, x), \sigma) \text{ (by the induction hypothesis)} \\ &= \bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q_0, x)\} \text{ (by the definition of } \delta_1) \\ &= \delta^*(q_0, x\sigma) \text{ (by the definition of } \delta^*) \end{aligned}$$

A string x is accepted by M_1 precisely if $\delta_1^*(q_1, x) \in A_1$. We know now that this is true if and only if $\delta^*(q_0, x) \in A_1$; and according to the definition of A_1, this is true if and only if $\delta^*(q_0, x) \cap A \neq \emptyset$. Therefore, x is accepted by M_1 if and only if x is accepted by M.

We present three examples: one that illustrates the construction in Theorem 3.17, one that illustrates the subset construction in Theorem 3.18, and one in which we use both to convert an NFA with Λ-transitions to an ordinary FA.

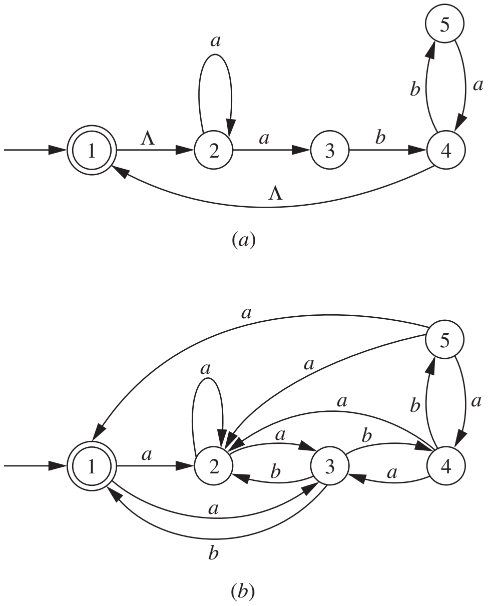
Eliminating Λ-Transitions from an NFA **EXAMPLE 3.19**

Figure 3.20a shows the transition diagram for an NFA M with Λ-transitions; it is not hard to see that it accepts the language corresponding to the regular expression $(a^*ab(ba)^*)^*$. We show in tabular form the values of the transition function δ, as well as the values $\delta^*(q, a)$ and $\delta^*(q, b)$ that will give us the transition function δ_1 in the resulting NFA M_1.

q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, \Lambda)$	$\delta^*(q, a)$	$\delta^*(q, b)$
1	∅	∅	{2}	{2, 3}	∅
2	{2, 3}	∅	∅	{2, 3}	∅
3	∅	{4}	∅	∅	{1, 2, 4}
4	∅	{5}	{1}	{2, 3}	{5}
5	{4}	∅	∅	{1, 2, 4}	∅

For example, the value $\delta^*(5, a)$ is the set {1, 2, 4}, because $\delta(5, a) = \{4\}$ and there are Λ-transitions from 4 to 1 and from 1 to 2.

Figure 3.20b shows the NFA M_1, whose transition function has the values in the last two columns of the table. In this example, the initial state of M is already an accepting state, and so drawing the new transitions and eliminating the Λ-transitions are the only steps required to obtain M_1.



(*a*)

(*b*)

Figure 3.20 |
Eliminating Λ-transitions from an NFA.

EXAMPLE 3.21 Using the Subset Construction to Eliminate Nondeterminism

We consider the NFA $M = (Q, \{a, b\}, 0, A, \delta)$ in Example 3.6, shown in Figure 3.7. Instead of labeling states as q_i, here we will use only the subscript i. We will describe the FA $M_1 = (2^Q, \{a, b\}, \{0\}, A_1, \delta_1)$ obtained from the construction in the proof of Theorem 3.18. Because a set with n elements has 2^n subsets, using this construction might require an exponential increase in the number of states. As this example will illustrate, we can often get by with fewer by considering only the states of M_1 (subsets of Q) that are reachable from $\{0\}$, the initial state of M_1.

It is helpful, and in fact recommended, to use a transition table for δ in order to obtain the values of δ_1. The table is shown below.

q	$\delta(q, a)$	$\delta(q, b)$
0	$\{1, 2\}$	$\{4\}$
1	$\{0\}$	$\emptyset$
2	$\{3\}$	$\emptyset$
3	$\emptyset$	$\{0\}$
4	$\emptyset$	$\emptyset$

The transition diagram for M_1 is shown in Figure 3.22. For example, $\delta_1(\{1, 2\}, a) = \delta(1, a) \cup \delta(2, a) = \{0, 3\}$. If you compare Figure 3.22 to Figure 2.23c, you will see that they are the same except for the way the states are labeled. The subset construction doesn't always produce the FA with the fewest possible states, but in this example it does.

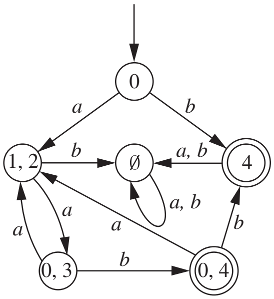


Figure 3.22 | Applying the subset construction to the NFA in Example 3.21.

Converting an NFA with Λ-Transitions to an FA **EXAMPLE 3.23**

For the NFA pictured in Figure 3.24a, we show the transition function in tabular form below, as well as the transition function for the resulting NFA without Λ-transitions. It is pictured in Figure 3.24b.

q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, \Lambda)$	$\delta^*(q, a)$	$\delta^*(q, b)$
1	{1}	∅	{2, 4}	{1, 2, 3, 4, 5}	{4, 5}
2	{3}	{5}	∅	{3}	{5}
3	∅	{2}	∅	∅	{2}
4	{5}	{4}	∅	{5}	{4}
5	∅	∅	∅	∅	∅

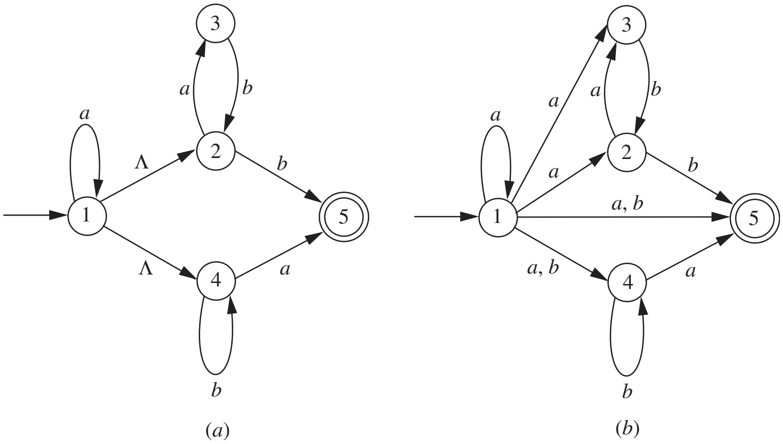


Figure 3.24 | Converting an NFA to an FA.

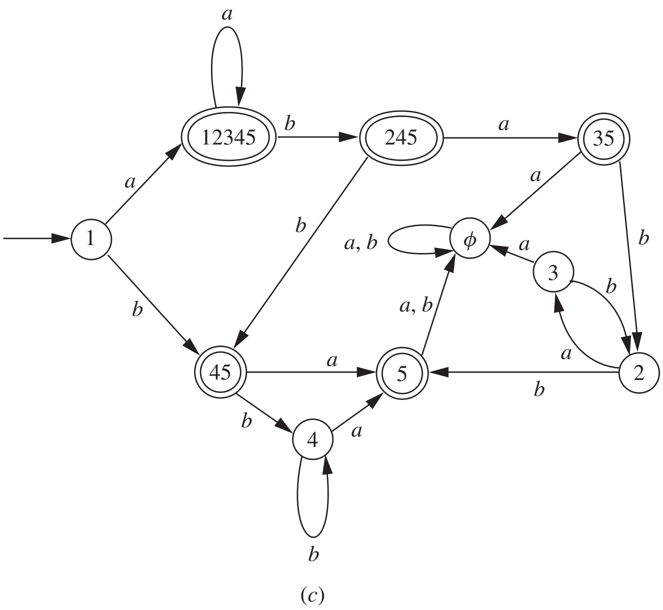


Figure 3.24 | Continued

The subset construction gives us a slightly greater variety of subsets this time, but still considerably fewer than the total number of subsets of Q. The final FA is shown in Figure 3.24c.

3.4 | KLEENE'S THEOREM, PART 1

If we are trying to construct a device that accepts a regular language L, we can proceed one state at a time, as in Example 2.22, deciding at each step which strings it is necessary to distinguish. Adding each additional state may get harder as the number of states grows, but if we know somehow that there is an FA accepting L, we can be sure that the procedure will eventually terminate and produce one.

We have examples to show that for certain regular expressions, nondeterminism simplifies the problem of drawing an accepting device. In this section we will use nondeterminism to show that we can do this for every regular expression. Furthermore, we now have algorithms to convert the resulting NFA to an FA. The conclusion will be that on the one hand, the state-by-state approach will always work; and on the other hand, there is a systematic procedure that is also guaranteed to work and may be more straightforward.

The general result is one half of Kleene's theorem, which says that regular languages are the languages that can be accepted by finite automata. We will discuss the first half in this section and the second in Section 3.5.

Theorem 3.25 Kleene's Theorem, Part 1

For every alphabet Σ, every regular language over Σ can be accepted by a finite automaton.

Proof

Because of Theorems 3.17 and 3.18, it's enough to show that every regular language over Σ can be accepted by an NFA. The set of regular languages over Σ is defined recursively in Definition 3.1, and we will prove the theorem by structural induction.

The languages $\emptyset$ and $\{\sigma\}$ (where $\sigma \in \Sigma$) can be accepted by the two NFAs in Figure 3.26, respectively. The induction hypothesis is that L_1 and L_2 are both regular languages over Σ and that for both $i = 1$ and $i = 2$, L_i can be accepted by an NFA $M_i = (Q_i, \Sigma, q_i, A_i, \delta_i)$. We can assume, by renaming states if necessary, that Q_1 and Q_2 are disjoint. In the induction step we must show that there are NFAs accepting the three languages $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, and $L(M_1)^*$.

In each case we will give an informal definition and a diagram showing the idea of the construction. For simplicity, each diagram shows the two NFAs M_1 and M_2 as having two accepting states, both distinct from the initial state.

An NFA M_u accepting $L(M_1) \cup L(M_2)$ is shown in Figure 3.27a. Its states are those of M_1 and M_2 and one additional state q_u that is the initial state. The transitions include all the ones in M_1 and M_2 as well as Λ-transitions from q_u to q_1 and q_2, the initial states of M_1 and M_2. Finally, the accepting states are simply the states in $A_1 \cup A_2$.

If $x \in L(M_1)$, for example, M_u can accept x by taking the Λ-transition from q_u to q_1 and then executing the moves that would allow M_1 to accept x. On the other hand, if x is any string accepted by M_u, there is a path from q_u to an element of A_1 or A_2. The first transition in the path must be a Λ-transition, which takes M_u to q_1 or q_2. Because $Q_1 \cap Q_2 = \emptyset$, the remainder of the path causes x to be accepted either by M_1 or by M_2.

An NFA M_c accepting $L(M_1)L(M_2)$ is shown in Figure 3.27b. No new states need to be added to those of M_1 and M_2. The initial state is q_1, and the accepting states are the elements of A_2. The transitions include all those of M_1 and M_2 and a new Λ-transition from every element of A_1 to q_2. If x is the string x_1x_2, where x_i is accepted by M_i for each i, then M_c can process x by moving from q_1 to a state in A_1 using Λ's and the symbols of x_1, taking the Λ-transition to q_2, and moving to a state in A_2 using Λ's and the symbols of x_2. Conversely, if x is a string accepted

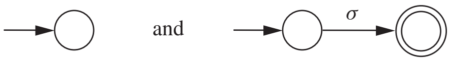


Figure 3.26

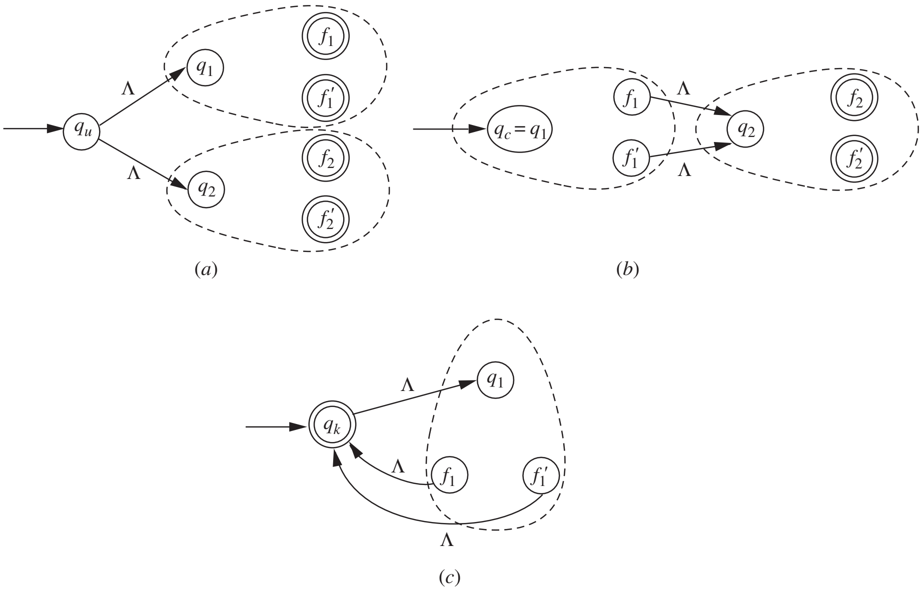


Figure 3.27 |
Schematic diagram for Kleene's theorem, Part 1.

by M_c, then at some point during the computation, M_c must execute the Λ-transition from an element of A_1 to q_2. If x_1 is the prefix of x whose symbols have been processed at that point, then x_1 must be accepted by M_1; the remaining suffix of x is accepted by M_2, because it corresponds to a path from q_2 to an element of A_2 that cannot involve any transitions other than those of M_2.

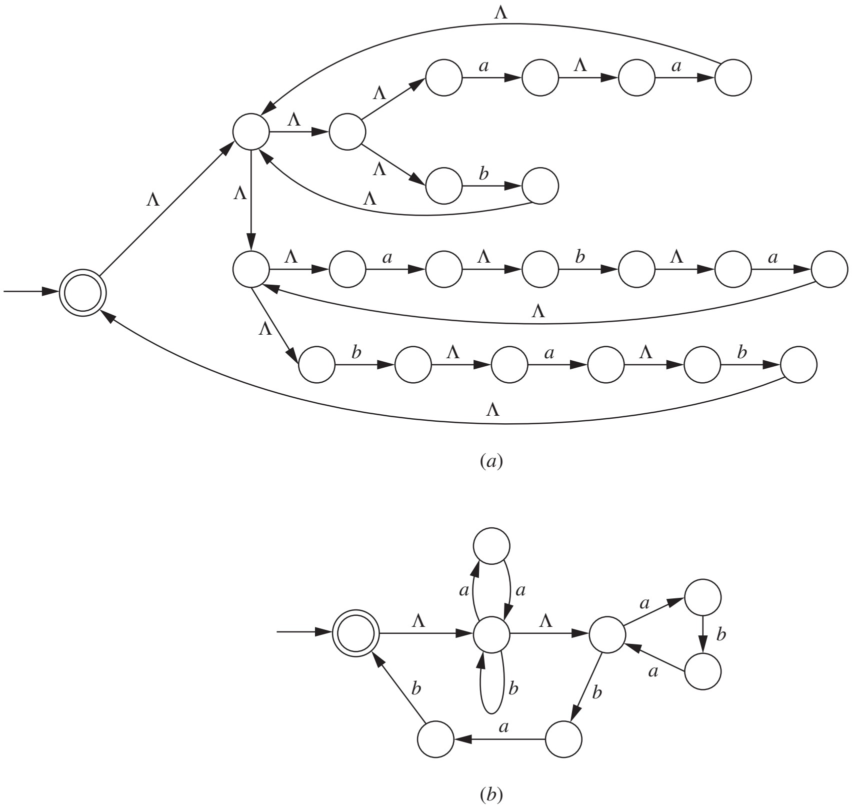
Finally, an NFA M_k accepting $L(M_1)^*$ is shown in Figure 3.27c. Its states are the elements of Q_1 and a new initial state q_k that is also the only accepting state. The transitions are those of M_1, a Λ-transition from q_k to q_1, and a Λ-transition from every element of A_1 to q_k. We can see by structural induction that every element of $L(M_1)^*$ is accepted. The null string is, because q_k is an accepting state. Now suppose that $x \in L(M_1)^*$ is accepted and that $y \in L(M_1)$. When M^* is in a state in A_1 after processing x, it can take a Λ-transition to q_k and another to q_1, process y so as to end up in an element of A_1, and finish up by returning to q_k with a Λ-transition. Therefore, xy is accepted by M^*.

We can argue in the opposite direction by using mathematical induction on the number of times M^* enters the state q_k in the process of accepting a string. If M^* visits q_k only once in accepting x, then $x = \Lambda$, which is an element of $L(M_1)^*$. If we assume that $n \geq 1$ and that every string accepted by M^* that causes M^* to enter q_k n or fewer times is in $L(M_1)^*$, then consider a string x that causes M^* to enter q_k $n + 1$ times and is accepted. Let x_1 be the prefix of x that is accepted when M^* enters q_k the nth time, and let x_2 be the remaining part of x. By the induction hypothesis, $x_1 \in L(M_1)^*$. In processing x_2, M^* moves to q_1 on a Λ-transition and then from q_1 to an element of A_1 using Λ-transitions in addition to the symbols of x_2. Therefore, $x_2 \in L(M_1)$, and it follows that $x \in L(M_1)^*$.

An NFA Corresponding to $((aa + b)^*(aba)^*bab)^*$ EXAMPLE 3.28

The three portions of the induction step in the proof of Theorem 3.25 provide algorithms for constructing an NFA corresponding to an arbitrary regular expression. These can be combined into a general algorithm that could be used to automate the process.

The transition diagram in Figure 3.29a shows a literal application of the three algorithms in the case of the regular expression $((aa + b)^*(aba)^*bab)^*$. In this case there is no need for all the Λ-transitions that are called for by the algorithms, and a simplified NFA is



(*a*)

(*b*)

Figure 3.29 |
Constructing an NFA for the regular expression $((aa + b)^*(aba)^*bab)^*$.

shown in Figure 3.29b. At least two Λ-transitions are still helpful in order to preserve the resemblance between the transition diagram and the regular expression. The algorithms can often be shortened in examples, but for each step where one of them calls for an extra state and/or a Λ-transition, there are examples to show that dispensing with the extra state or the transition doesn't always work (see Exercises 3.45–3.48).

3.5 | KLEENE'S THEOREM, PART 2

In this section we prove that if L is accepted by a finite automaton, then L is regular. The proof will provide an algorithm for starting with an FA that accepts L and finding a regular expression that describes L.

Theorem 3.30 Kleene's Theorem, Part 2

For every finite automaton $M = (Q, \Sigma, q_0, A, \delta)$, the language $L(M)$ is regular.

Proof

For states p and q, we introduce the notation $L(p, q)$ for the language

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

If we can show that for every p and q in Q, $L(p, q)$ is regular, then it will follow that $L(M)$ is, because

$$L(M) = \bigcup\{L(q_0, q) \mid q \in A\}$$

and the union of a finite collection of regular languages is regular.

We will show that each language $L(p, q)$ is regular by expressing it in terms of simpler languages that are regular. Strings in $L(p, q)$ cause M to move from p to q in any manner whatsoever. One way to think about simpler ways of moving from p to q is to think about the number of transitions involved; the problem with this approach is that there is no upper limit to this number, and so no obvious way to obtain a final regular expression. A similar approach that is more promising is to consider the distinct states through which M passes as it moves from p to q. We can start by considering how M can go from p to q without going through any states, and at each step add one more state to the set through which M is allowed to go. This procedure will terminate when we have enlarged the set to include all possible states.

If $x \in L(p, q)$, we say x causes M to go from p to q *through* a state r if there are nonnull strings x_1 and x_2 such that $x = x_1x_2$, $\delta^*(p, x_1) = r$, and $\delta^*(r, x_2) = q$. In using a string of length 1 to go from p to q, M does not go through any state. (If M loops from p back to p on the symbol a, it does not go through p even though the string a causes it to leave p and enter p.) In using a string of length $n \geq 2$, it goes through a state $n - 1$ times, but if $n > 2$ these states may not be distinct.

Now we assume that Q has n elements and that they are numbered from 1 to n. For $p, q \in Q$ and $j \geq 0$, we let $L(p, q, j)$ be the set of strings in $L(p, q)$ that cause M to go from p to q without going through any state numbered higher than j.

The set $L(p, q, 0)$ is the set of strings that allow M to go from p to q without going through any state at all. This includes the set of alphabet symbols σ for which $\delta(p, \sigma) = q$, and in the case when $p = q$ it also includes the string Λ. In any case, $L(p, q, 0)$ is a finite set of strings and therefore regular.

Suppose that for some number $k \geq 0$, $L(p, q, k)$ is regular for every p and every q in Q, and consider how a string can be in $L(p, q, k + 1)$. The easiest way is for it to be in $L(p, q, k)$, because if M goes through no state numbered higher than k, it certainly goes through no state numbered higher than $k + 1$. The other strings in $L(p, q, k + 1)$ are those that cause M to go from p to q by going through state $k + 1$ and no higher-numbered states. A path of this type goes from p to $k + 1$; it may return to $k + 1$ one or more times; and it finishes by going from $k + 1$ to q (see Figure 3.31). On each of these individual portions, the path starts or stops at state $k + 1$ but doesn't go through any state numbered higher than k.

Every string in $L(p, q, k + 1)$ can be described in one of these two ways, and every string that has one of these two forms is in $L(p, q, k + 1)$. The resulting formula is

$$\begin{aligned} L(p, q, k+1) = {} & L(p, q, k) \cup L(p, k+1, k) L(k+1, k+1, k)^* \\ & L(k+1, q, k) \end{aligned}$$

We have the ingredients, both for a proof by mathematical induction that $L(p, q)$ is regular and for an algorithm to obtain a regular expression for this language. $L(p, q, 0)$ can be described by a regular expression; for each $k < n$, $L(p, q, k + 1)$ is described by the formula above; and $L(p, q, n) = L(p, q)$, because the condition that the path go through no state numbered higher than n is no restriction at all if there are no states numbered higher than n. As we observed at the beginning of the proof, the last step in obtaining a regular expression for $L(M)$ is to use the + operation to combine the expressions for the languages $L(q_0, q)$, where $q \in A$.

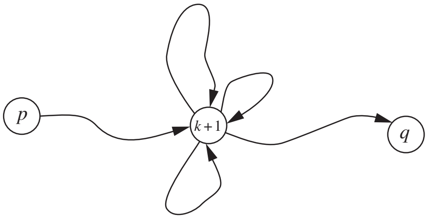


Figure 3.31 |
Going from p to q by going through $k + 1$.

EXAMPLE 3.32 Finding a Regular Expression Corresponding to an FA

Let M be the finite automaton pictured in Figure 3.33.

If we let $r(i, j, k)$ denote a regular expression corresponding to the language $L(i, j, k)$ described in the proof of Theorem 3.30, then $L(M)$ is described by the regular expression $r(M)$, where

$$r(M) = r(1, 1, 3) + r(1, 2, 3)$$

We might try calculating this expression from the top down, at least until we can see how many of the terms $r(i, j, k)$ we will need that involve smaller values of k. The recursive formula in the proof of the theorem tells us that

$$r(1, 1, 3) = r(1, 1, 2) + r(1, 3, 2)r(3, 3, 2)^*r(3, 1, 2)$$
$$r(1, 2, 3) = r(1, 2, 2) + r(1, 3, 2)r(3, 3, 2)^*r(3, 2, 2)$$

Applying the formula to the expressions $r(i, j, 2)$ that we apparently need, we obtain

$$r(1, 1, 2) = r(1, 1, 1) + r(1, 2, 1)r(2, 2, 1)^*r(2, 1, 1)$$
$$r(1, 3, 2) = r(1, 3, 1) + r(1, 2, 1)r(2, 2, 1)^*r(2, 3, 1)$$
$$r(3, 3, 2) = r(3, 3, 1) + r(3, 2, 1)r(2, 2, 1)^*r(2, 3, 1)$$
$$r(3, 1, 2) = r(3, 1, 1) + r(3, 2, 1)r(2, 2, 1)^*r(2, 1, 1)$$
$$r(1, 2, 2) = r(1, 2, 1) + r(1, 2, 1)r(2, 2, 1)^*r(2, 2, 1)$$
$$r(3, 2, 2) = r(3, 2, 1) + r(3, 2, 1)r(2, 2, 1)^*r(2, 2, 1)$$

At this point it is clear that we need every one of the expressions $r(i, j, 1)$, and we now start at the bottom and work our way up. The three tables below show the expressions $r(i, j, 0)$, $r(i, j, 1)$, and $r(i, j, 2)$ for all combinations of i and j. (Only six of the nine entries in the last table are required.)

p	$r(p, 1, 0)$	$r(p, 2, 0)$	$r(p, 3, 0)$
1	$a + \Lambda$	b	$\emptyset$
2	a	Λ	b
3	a	b	Λ

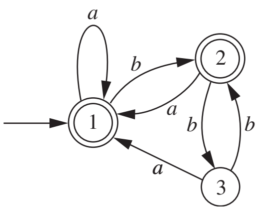


Figure 3.33 | An FA for which we want an equivalent regular expression.

p	$r(p,1,1)$	$r(p,2,1)$	$r(p,3,1)$
1	a^*	a^*b	$\emptyset$
2	aa^*	$\Lambda + aa^*b$	b
3	aa^*	a^*b	Λ

p	$r(p,1,2)$	$r(p,2,2)$	$r(p,3,2)$
1	$a^*(baa^*)^*$	$a^*(baa^*)^*b$	$a^*(baa^*)^*bb$
2	$aa^*(baa^*)^*$	$(aa^*b)^*$	$(aa^*b)^*b$
3	$aa^* + a^*baa^*(baa^*)^*$	$a^*b(aa^*b)^*$	$\Lambda + a^*b(aa^*b)^*b$

For example,

$$\begin{aligned} r(2,2,1) &= r(2,2,0) + r(2,1,0)r(1,1,0)^*r(1,2,0) \\ &= \Lambda + (a)(a+\Lambda)^*(b) \\ &= \Lambda + aa^*b \\ r(3,1,2) &= r(3,1,1) + r(3,2,1)r(2,2,1)^*r(2,1,1) \\ &= aa^* + (a^*b)(\Lambda + aa^*b)^*(aa^*) \\ &= aa^* + a^*b(aa^*b)^*aa^* \\ &= aa^* + a^*baa^*(baa^*)^* \end{aligned}$$

The terms required for the final regular expression can now be obtained from the last table. As you can see, these expressions get very involved, even though we have already made some attempts to simplify them. There is no guarantee that the final regular expression is the simplest possible (it seems clear in this case that it is not), but at least we have a systematic way of generating a regular expression corresponding to $L(M)$.

EXERCISES

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

a. $b^*(ab)^*a^*$

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

c. $a^*(baa^*)^*b^*$

d. $b^*(a + ba)^*b^*$

3.2. Consider the two regular expressions

$$r = a^* + b^* \qquad s = ab^* + ba^* + b^*a + (a^*b)^*$$

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 $\{a, b\}^*$ corresponding to neither r nor s.

3.3. Let r and s be arbitrary regular expressions over the alphabet Σ. In each case below, find a simpler equivalent regular expression.

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

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

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

3.4. It is not difficult to show using mathematical induction that for every integer $n \geq 2$, there are nonnegative integers i and j such that $n = 2i + 3j$. With this in mind, simplify the regular expression $(aa + aaa)(aa + aaa)^*$.

3.5. In each case below, give a simple description of the smallest set of languages that contains all the "basic" languages $\emptyset$, $\{\Lambda\}$, and $\{\sigma\}$ (for every $\sigma \in \Sigma$) and is closed under the specified operations.

a. union

b. concatenation

c. union and concatenation

3.6. Suppose w and z are strings in $\{a, b\}^*$. Find regular expressions corresponding to each of the languages defined recursively below.

a. $\Lambda \in L$; for every $x \in L$, then wx and xz are elements of L.

b. $a \in L$; for every $x \in L$, wx, xw, and xz are elements of L.

c. $\Lambda \in L$; $a \in L$; for every $x \in L$, wx and zx are in L.

3.7. Find a regular expression corresponding to each of the following subsets of $\{a, b\}^*$.

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

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

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

d. The language of all strings that begin or end with aa or bb.

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

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

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

h. The language of all strings in which every a is followed immediately by bb.

i. The language of all strings containing both bb and aba as substrings.

j. The language of all strings not containing the substring aaa.

k. The language of all strings not containing the substring bba.

l. The language of all strings containing both bab and aba as substrings.

m. The language of all strings in which the number of a's is even and the number of b's is odd.

n. The language of all strings in which both the number of a's and the number of b's are odd.

3.8. a. The regular expression $(b + ab)^*(a + ab)^*$ describes the set of all strings in $\{a, b\}^*$ not containing the substring _______x_______ for any x. (Fill in the blanks appropriately.)

b. The regular expression $(a + b)^*(aa^*bb^*aa^* + bb^*aa^*bb^*)$ $(a + b)^*$ describes the set of all strings in $\{a, b\}^*$ containing both the substrings _______ and _______. (Fill in the blanks appropriately.)

3.9. Show that every finite language is regular.

3.10. a. If L is the language corresponding to the regular expression $(aab + bbaba)^*baba$, find a regular expression corresponding to $L^r = \{x^r \mid x \in L\}$.

b. Using the example in part (a) as a model, give a recursive definition (based on Definition 3.1) of the reverse e^r of a regular expression e.

c. Show that for every regular expression e, if the language L corresponds to e, then L^r corresponds to e^r.

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

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

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

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

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

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

3.13. Let c and d be regular expressions over Σ.

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 Λ is not in the language corresponding to d, then any regular expression r satisfying $r = c + rd$ corresponds to the same language as cd^*.

3.14. 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.

3.15. Describe an algorithm that could be used to eliminate the symbol Λ from any regular expression whose corresponding language does not contain the null string.

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

a. Show that the order of L is finite if and only if there is an integer k such that $L^k = L^*$, and that in this case the order of L is the smallest k such 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 $(\Lambda + b^*a)(b + ab^*ab^*a)^*$?

3.17. †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 $\{aaa\}^*$ be described this way? Give reasons for your answer.

3.18. Figure 3.34, at the bottom of this page, shows a transition diagram for an NFA. For each string below, say whether the NFA accepts it.

a. aba

b. $abab$

c. $aaabbb$

3.19. Find a regular expression corresponding to the language accepted by the NFA pictured in Figure 3.34. You should be able to do it without applying Kleene's theorem: First find a regular expression describing the most general way of reaching state 4 the first time, and then find a regular expression describing the most general way, starting in state 4, of moving to state 4 the next time.

3.20. For each of the NFAs shown in Figure 3.35 on the next page, find a regular expression corresponding to the language it accepts.

3.21. On the next page, after Figure 3.35, is the transition table for an NFA with states 1–5 and input alphabet $\{a, b\}$. There are no Λ-transitions.

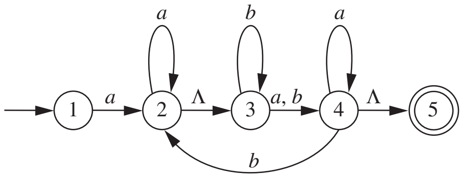
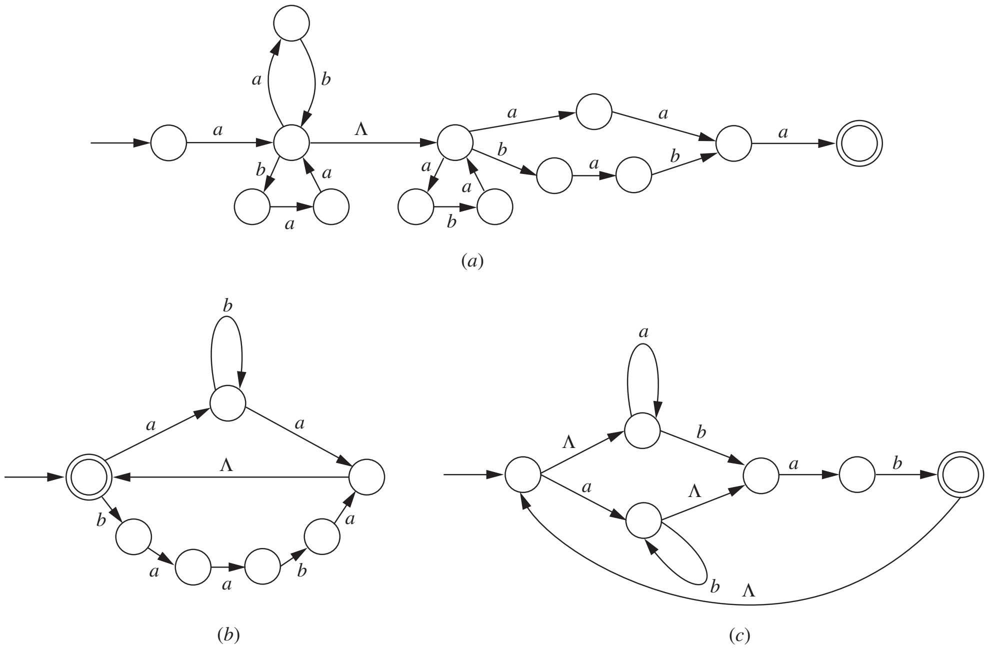


Figure 3.34 |



(*a*)

(*b*)

(*c*)

Figure 3.35 |

q	δ(q, a)	δ(q, b)
1	{1, 2}	{1}
2	{3}	{3}
3	{4}	{4}
4	{5}	∅
5	∅	{5}

a. Draw a transition diagram.

b. Calculate $\delta^*(1, ab)$.

c. Calculate $\delta^*(1, abaab)$.

3.22. A transition table is given for an NFA with seven states.

q	δ(q, a)	δ(q, b)	δ(q,Λ)
1	∅	∅	{2}
2	{3}	∅	{5}
3	∅	{4}	∅
4	{4}	∅	{1}
5	∅	{6, 7}	∅
6	{5}	∅	∅
7	∅	∅	{1}

Find:

a. $\Lambda(\{2, 3\})$

b. $\Lambda(\{1\})$

c. $\Lambda(\{3, 4\})$

d. $\delta^*(1, ba)$

e. $\delta^*(1, ab)$

f. $\delta^*(1, ababa)$

3.23. A transition table is given for another NFA with seven states.

q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, \Lambda)$
1	$\{5\}$	$\emptyset$	$\{4\}$
2	$\{1\}$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\{2\}$	$\emptyset$
4	$\emptyset$	$\{7\}$	$\{3\}$
5	$\emptyset$	$\emptyset$	$\{1\}$
6	$\emptyset$	$\{5\}$	$\{4\}$
7	$\{6\}$	$\emptyset$	$\emptyset$

Calculate $\delta^*(1, ba)$.

3.24. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA with no Λ-transitions. Show that for every $q \in Q$ and every $\sigma \in \Sigma$, $\delta^*(q, \sigma) = \delta(q, \sigma)$.

3.25. It is easy to see that if $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting L, then the FA $M' = (Q, \Sigma, q_0, Q - A, \delta)$ accepts L' (the FA obtained from Theorem 2.15 by writing $L' = \Sigma^* - L$ is essentially M'). Does this still work if M is an NFA? If so, prove it. If not, find a counterexample.

3.26. In Definition 3.14, δ^* is defined recursively in an NFA by first defining $\delta^*(q, \Lambda)$ and then defining $\delta^*(q, y\sigma)$, where $y \in \Sigma^*$ and $\sigma \in \Sigma$. Give an acceptable recursive definition in which the recursive part of the definition defines $\delta^*(q, \sigma y)$ instead.

3.27. Which of the following, if any, would be a correct substitute for the second part of Definition 3.14? Give reasons for your answer.

a. $\delta^*(q, \sigma y) = \Lambda(\bigcup\{\delta^*(r, y) \mid r \in \delta(q, \sigma)\})$

b. $\delta^*(q, \sigma y) = \bigcup\{\Lambda(\delta^*(r, y)) \mid r \in \delta(q, \sigma)\}$

c. $\delta^*(q, \sigma y) = \bigcup\{\delta^*(r, y) \mid r \in \Lambda(\delta(q, \sigma))\}$

d. $\delta^*(q, \sigma y) = \bigcup\{\Lambda(\delta^*(r, y)) \mid r \in \Lambda(\delta(q, \sigma))\}$

3.28. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. This exercise involves properties of the Λ-closure of a set S. Since $\Lambda(S)$ is defined recursively, structural induction can be used to show that $\Lambda(S)$ is a subset of some other set.

a. Show that if S and T are subsets of Q for which $S \subseteq T$, then $\Lambda(S) \subseteq \Lambda(T)$.

b. Show that for any $S \subseteq Q$, $\Lambda(\Lambda(S)) = \Lambda(S)$.

c. Show that if $S, T \subseteq Q$, then $\Lambda(S \cup T) = \Lambda(S) \cup \Lambda(T)$.

d. Show that if $S \subseteq Q$, then $\Lambda(S) = \bigcup\{\Lambda(\{p\}) \mid p \in S\}$.

e. Draw a transition diagram to illustrate the fact that $\Lambda(S \cap T)$ and $\Lambda(S) \cap \Lambda(T)$ are not always the same. Which is always a subset of the other?

f. Draw a transition diagram illustrating the fact that $\Lambda(S')$ and $\Lambda(S)'$ are not always the same. Which is always a subset of the other? Under what circumstances are they equal?

3.29. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. A set $S \subseteq Q$ is called Λ-*closed* if $\Lambda(S) = S$.

a. Show that the union of two Λ-closed sets is Λ-closed.

b. Show that the intersection of two Λ-closed sets is Λ-closed.

c. Show that for any subset S of Q, $\Lambda(S)$ is the smallest Λ-closed set of which S is a subset.

3.30. †Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. Show that for every $q \in Q$ and every $x, y \in \Sigma^*$,

$$\delta^*(q, xy) = \bigcup\{\delta^*(r, y) \mid r \in \delta^*(q, x)\}$$

3.31. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA, and let $M_1 = (Q, \Sigma, q_0, A, \delta_1)$ be the NFA with no Λ-transitions for which $\delta_1(q, \sigma) = \{\delta(q, \sigma)\}$ for every $q \in Q$ and $\sigma \in \Sigma$. Show that for every $q \in Q$ and $x \in \Sigma^*$, $\delta_1^*(q, x) = \{\delta(q, x)\}$. Recall that the two functions δ^* and δ_1^* are defined differently.

3.32. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA accepting a language L. Assume that there are no transitions to q_0, that A has only one element, q_f, and that there are no transitions from q_f.

a. Let M_1 be obtained from M by adding Λ-transitions from q_0 to every state that is reachable from q_0 in M. (If p and q are states, q is reachable from p if there is a string $x \in \Sigma^*$ such that $q \in \delta^*(p, x)$.) Describe (in terms of L) the language accepted by M_1.

b. Let M_2 be obtained from M by adding Λ-transitions to q_f from every state from which q_f is reachable in M. Describe in terms of L the language accepted by M_2.

c. Let M_3 be obtained from M by adding both the Λ-transitions in (a) and those in (b). Describe the language accepted by M_3.

3.33. Give an example of a regular language L containing Λ that cannot be accepted by any NFA having only one accepting state and no Λ-transitions, and show that your answer is correct.

3.34. Can every regular language not containing Λ be accepted by an NFA having only one accepting state and no Λ-transitions? Prove your answer.

3.35. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA, let m be the maximum size of any of the sets $\delta^*(q, \sigma)$ for $q \in Q$ and $\sigma \in \Sigma$, and let x be a string of length n over the input alphabet.

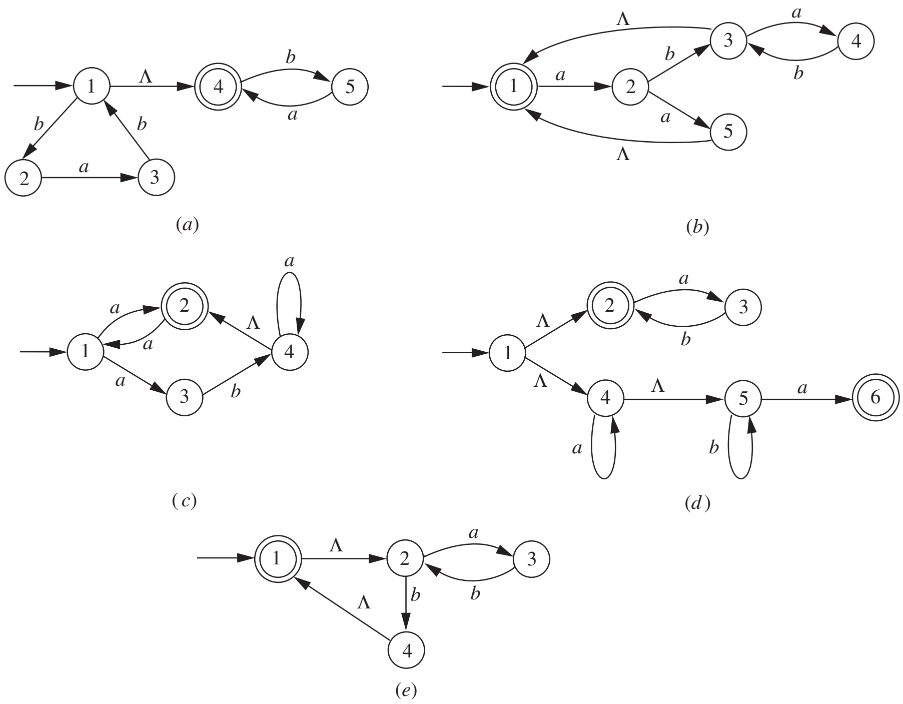


Figure 3.36 |

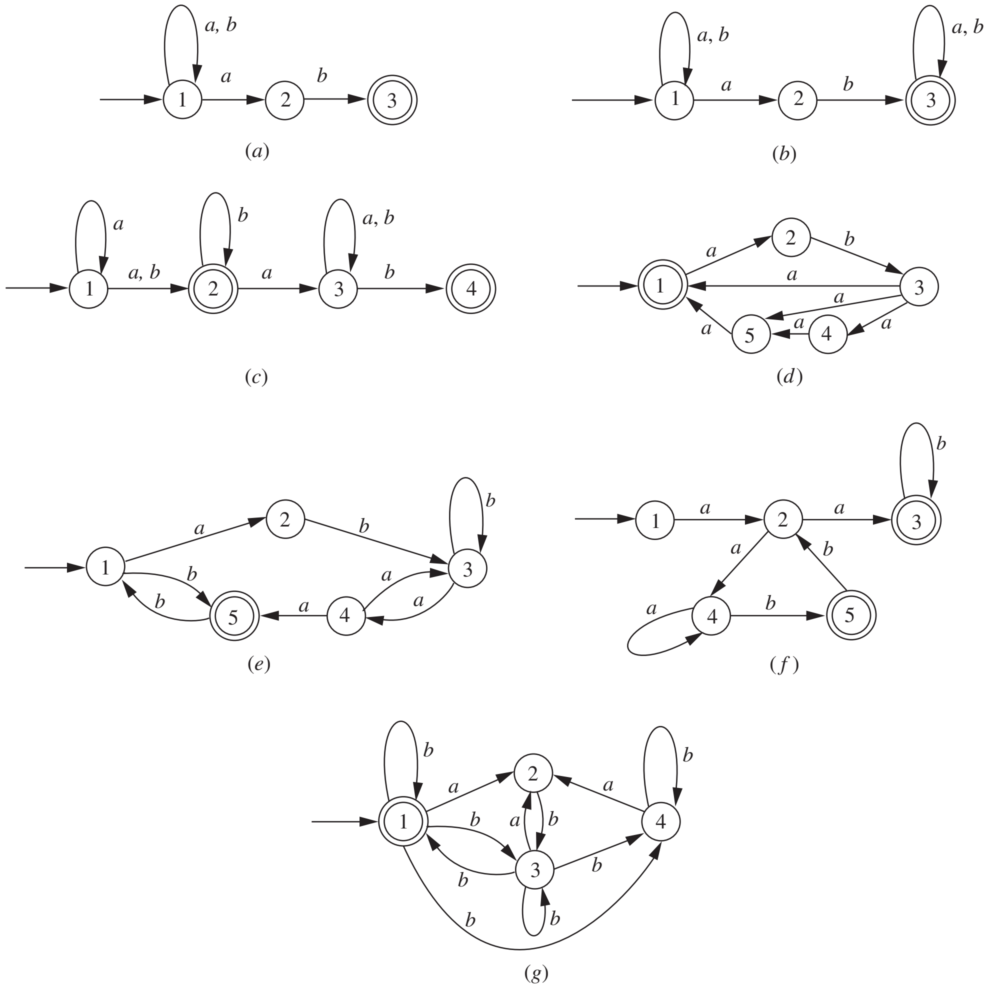
a. What is the maximum number of distinct paths that there might be in the computation tree corresponding to x?

b. In order to determine whether x is accepted by M, it is sufficient to replace the complete computation tree by one that is perhaps smaller, obtained by "pruning" the original one so that no level of the tree contains more nodes than the number of states in M (and no level contains more nodes than there are at that level of the original tree). Explain why this is possible, and how it might be done.

3.36. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. The NFA M_1 obtained by eliminating Λ-transitions from M might have more accepting states than M, because the initial state q_0 is made an accepting state if $\Lambda(\{q_0\}) \cap A \neq \emptyset$. Explain why it is not necessary to make *all* the states q for which $\Lambda(\{q\}) \cap A \neq \emptyset$ accepting states in M_1.

3.37. In each part of Figure 3.36 is pictured an NFA. Use the algorithm described in the proof of Theorem 3.17 to draw an NFA with no Λ-transitions accepting the same language.

3.38. Each part of Figure 3.37 pictures an NFA. Using the subset construction, draw an FA accepting the same language. Label the final picture so as to make it clear how it was obtained from the subset construction.



(*a*) (*b*) (*c*) (*d*) (*e*) (*f*) (*g*)

Figure 3.37 |

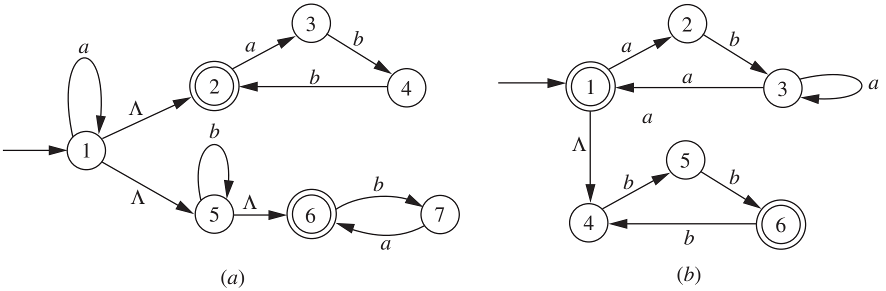
3.39. Suppose $L \subseteq \Sigma^*$ is a regular language. If every FA accepting L has at least n states, then every NFA accepting L has at least ____ states. (Fill in the blank, and explain your answer.)

3.40. Each part of Figure 3.38 shows an NFA. Draw an FA accepting the same language.

3.41. For each of the following regular expressions, draw an NFA accepting the corresponding language, so that there is a recognizable correspondence between the regular expression and the transition diagram.

a. $(b + bba)^*a$

b. $(a + b)^*(abb + ababa)(a + b)^*$



(*a*)

(*b*)

Figure 3.38 |

c. $(a + b)(ab)^*(abb)^*$

d. $(a + b)^*(abba^* + (ab)^*ba)$

e. $(a^*bb)^* + bb^*a^*$

3.42. For part (e) of Exercise 3.41, draw the NFA that is obtained by a literal application of Kleene's theorem, without any simplifications.

3.43. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA accepting a language L. Let M_1 be the NFA obtained from M by adding Λ-transitions from each element of A to q_0. Describe (in terms of L) the language $L(M_1)$.

3.44. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA accepting a language L.

a. Describe how to construct an NFA M_1 with no transitions to its initial state so that M_1 also accepts L.

b. Describe how to construct an NFA M_2 with exactly one accepting state and no transitions from that state, so that M_2 also accepts L.

3.45. Suppose M is an NFA with exactly one accepting state q_f that accepts the language $L \subseteq \{a, b\}^*$. In order to find NFAs accepting the languages $\{a\}^*L$ and $L\{a\}^*$, we might try adding a-transitions from q_0 to itself and from q_f to itself, respectively. Draw transition diagrams to show that neither technique always works.

3.46. In the construction of M_u in the proof of Theorem 3.25, consider this alternative to the construction described: Instead of a new state q_u and Λ-transitions from it to q_1 and q_2, make q_1 the initial state of the new NFA, and create a Λ-transition from it to q_2. Either prove that this works in general, or give an example in which it fails.

3.47. In the construction of M_c in the proof of Theorem 3.25, consider the simplified case in which M_1 has only one accepting state. Suppose that we eliminate the Λ-transition from the accepting state of M_1 to q_2, and merge these two states into one. Either show that this would always work in this case, or give an example in which it fails.

3.48. In the construction of M^* in the proof of Theorem 3.25, suppose that instead of adding a new state q_0, with Λ-transitions from it to q_1 and to it

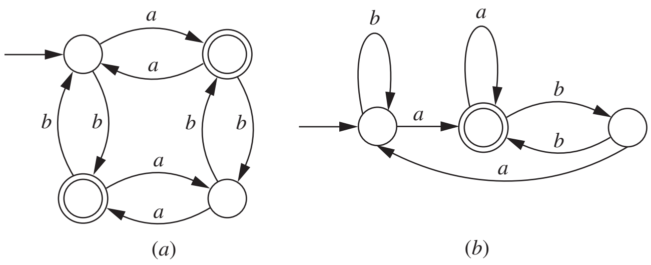


Figure 3.39 |

from each accepting state of Q_1, we make q_1 both the initial state and the accepting state, and create Λ-transitions from each accepting state of M_1 to q_0. Either show that this works in general, or give an example in which it fails.

3.49. Figure 3.39 shows FAs M_1 and M_2 accepting languages L_1 and L_2, respectively. Draw NFAs accepting each of the following languages, using the constructions in the proof of Theorem 3.25.

a. $L_2^* \cup L_1$

b. $L_2 L_1^*$

c. $L_1 L_2 \cup (L_2 L_1)^*$

3.50. Draw NFAs with no Λ-transitions accepting $L_1 L_2$ and $L_2 L_1$, where L_1 and L_2 are as in Exercise 3.49. Do this by connecting the two given diagrams directly, by arrows with appropriate labels.

3.51. Use the algorithm of Theorem 3.30 to find a regular expression corresponding to each of the FAs shown in Figure 3.40. In each case, if the FA has n states, construct tables showing $L(p, q, j)$ for each j with $0 \le j \le n - 1$.

3.52. Suppose M is an FA with the three states 1, 2, and 3, and 1 is both the initial state and the only accepting state. The expressions $r(p, q, 2)$ corresponding to the languages $L(p, q, 2)$ are shown in the table below. Write a regular expression describing $L(M)$.

p	$r(\mathbf{p}, \mathbf{1}, \mathbf{2})$	$r(\mathbf{p}, \mathbf{2}, \mathbf{2})$	$r(\mathbf{p}, \mathbf{3}, \mathbf{2})$
1	Λ	aa^*	$b + aa^*b$
2	$\emptyset$	a^*	a^*b
3	a	aaa^*	$\Lambda + b + ab + aaa^*b$

3.53. Suppose Σ_1 and Σ_2 are alphabets, and the function $f : \Sigma_1^* \to \Sigma_2^*$ is a *homomorphism*; i.e., $f(xy) = f(x)f(y)$ for every $x, y \in \Sigma_1^*$.

a. Show that $f(\Lambda) = \Lambda$.

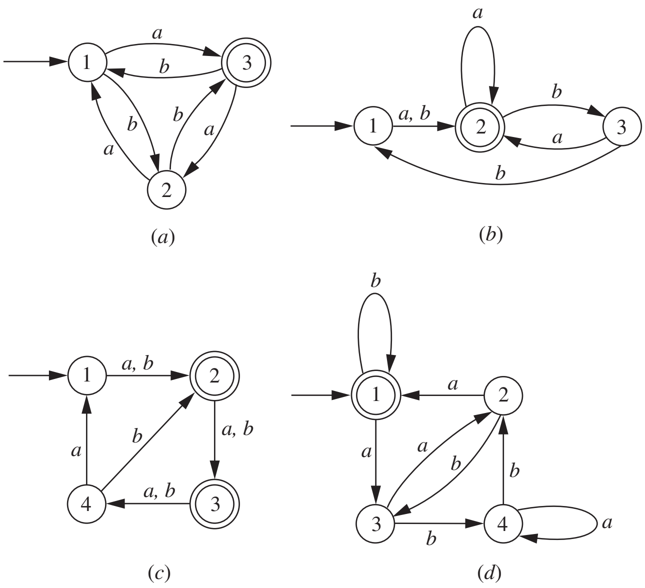


Figure 3.40 |

b. Show that if $L \subseteq \Sigma_1^*$ is regular, then $f(L)$ is regular. ($f(L)$ is the set $\{y \in \Sigma_2^* \mid y = f(x) \text{ for some } x \in L\}$.)

c. Show that if $L \subseteq \Sigma_2^*$ is regular, then $f^{-1}(L)$ is regular. ($f^{-1}(L)$ is the set $\{x \in \Sigma_1^* \mid f(x) \in L\}$.)

3.54. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA. For two (not necessarily distinct) states p and q, we define the regular expression $e(p, q)$ as follows: $e(p, q) = l + r_1 + r_2 + \cdots + r_k$, where l is either Λ (if $\delta(p, \Lambda)$ contains q) or $\emptyset$, and the r_i's are all the elements σ of Σ for which $\delta(p, \sigma)$ contains q. It's possible for $e(p, q)$ to be $\emptyset$, if there are no transitions from p to q; otherwise, $e(p, q)$ represents the "most general" transition from p to q.

If we generalize this by allowing $e(p, q)$ to be an arbitrary regular expression over Σ, we get what is called an *expression graph*. If p and q are two states in an expression graph G, and $x \in \Sigma^*$, we say that x allows G to move from p to q if there are states $p_0, p_1, \ldots, p_m$, with $p_0 = p$ and $p_m = q$, such that x corresponds to the regular expression $e(p_0, p_1)e(p_1, p_2)\ldots e(p_{n-1}, p_n)$. This allows us to say how G accepts a string x (x allows G to move from the initial state to an accepting state), and therefore to talk about the language accepted by G. It is easy to see that in the special case where G is simply an NFA, the two definitions for the language accepted by G coincide. It is also not hard to convince yourself, using Theorem 3.25, that for any expression graph G, the language accepted by G can be accepted by an NFA.

We can use the idea of an expression graph to obtain an alternate proof of Theorem 3.30, as follows. Starting with an FA M accepting L, we may easily convert it to an NFA M_1 accepting L, so that M_1 has no transitions to its initial state q_0, exactly one accepting state q_f (which is different from q_0), and no transitions from q_f. The remainder of the proof is to specify a reduction technique to reduce by one the number of states other than q_0 and q_f, obtaining an equivalent expression graph at each step, until q_0 and q_f are the only states remaining. The regular expression $e(q_0, q_f)$ then describes the language accepted. If p is the state to be eliminated, the reduction step involves redefining $e(q, r)$ for every pair of states q and r other than p.

Describe in more detail how this reduction can be done. Then apply this technique to the FAs in Figure 3.40 to obtain regular expressions corresponding to their languages.