Chapter 3

Regular Languages and Regular Grammars

According to our definition, a language is regular if there exists a finite accepter for it. Therefore, every regular language can be described by some dfa or some nfa. Such a description can be very useful, for example, if we want to show the logic by which we decide if a given string is in a certain language. But in many instances, we need more concise ways of describing regular languages. In this chapter, we look at other ways of representing regular languages. These representations have important practical applications, a matter that is touched on in some of the examples and exercises.

3.1 Regular Expressions

One way of describing regular languages is via the notation of **regular expressions.** This notation involves a combination of strings of symbols from some alphabet Σ, parentheses, and the operators $+$, $\cdot$, and $*$. The simplest case is the language $\{a\}$, which will be denoted by the regular expression a. Slightly more complicated is the language $\{a, b, c\}$, for which, using the $+$ to denote union, we have the regular expression $a+b+c$. We use $\cdot$ for concatenation and $*$ for star-closure in a similar way. The expression $(a+b\cdot c)^*$ stands for the star-closure of $\{a\} \cup \{bc\}$, that is, the language $\{\lambda, a, bc, aa, abc, bca, bcbc, aaa, aabc, ...\}$.

Formal Definition of a Regular Expression

We construct regular expressions from primitive constituents by repeatedly applying certain recursive rules. This is similar to the way we construct familiar arithmetic expressions.

Definition 3.1

Let Σ be a given alphabet. Then

1. $\varnothing, \lambda$, and $a \in \Sigma$ are all regular expressions. These are called **primitive regular expressions.**

2. If r_1 and r_2 are regular expressions, so are $r_1 + r_2$, $r_1 \cdot r_2$, r_1^*, and (r_1).

3. A string is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules in (2).

Example 3.1 For $\Sigma = \{a, b, c\}$, the string

$$(a+b\cdot c)^* \cdot (c+\varnothing)$$

is a regular expression, since it is constructed by application of the above rules. For example, if we take $r_1 = c$ and $r_2 = \varnothing$, we find that $c+\varnothing$ and $(c+\varnothing)$ are also regular expressions. Repeating this, we eventually generate the whole string. On the other hand, $(a+b+)$ is not a regular expression, since there is no way it can be constructed from the primitive regular expressions. ■

Languages Associated with Regular Expressions

Regular expressions can be used to describe some simple languages. If r is a regular expression, we will let $L(r)$ denote the language associated with r. This language is defined as follows:

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. for every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1) L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

The last four rules of this definition are used to reduce $L(r)$ to simpler components recursively; the first three are the termination conditions for this recursion. To see what language a given expression denotes, we apply these rules repeatedly.

Example 3.2 Exhibit the language $L(a^* \cdot (a+b))$ in set notation.

$$\begin{aligned} L(a^* \cdot (a+b)) &= L(a^*) L(a+b) \\ &= (L(a))^* (L(a) \cup L(b)) \\ &= \{\lambda, a, aa, aaa, ...\} \{a, b\} \\ &= \{a, aa, aaa, ..., b, ab, aab, ...\} \end{aligned}$$

■

There is one problem with rules (4) to (7) in Definition 3.2. They define a language precisely if r_1 and r_2 are given, but there may be some ambiguity in breaking a complicated expression into parts. Consider, for example, the regular expression $a \cdot b + c$. We can consider this as being made up of $r_1 = a \cdot b$ and $r_2 = c$. In this case, we find $L(a \cdot b + c) = \{ab, c\}$. But there is nothing in Definition 3.2 to stop us from taking $r_1 = a$ and $r_2 = b + c$. We now get a different result, $L(a \cdot b + c) = \{ab, ac\}$. To overcome this, we could require that all expressions be fully parenthesized, but this gives cumbersome results. Instead, we use a convention familiar from mathematics and programming languages. We establish a set of precedence rules for evaluation in which star-closure precedes concatenation and concatenation precedes union. Also, the symbol for concatenation may be omitted, so we can write $r_1 r_2$ for $r_1 \cdot r_2$.

boring & hard

With a little practice, we can see quickly what language a particular regular expression denotes.

Example 3.3 For $\Sigma = \{a, b\}$, the expression

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

is regular. It denotes the language

$$L(r) = \{a, bb, aa, abb, ba, bbb, ...\}.$$

We can see this by considering the various parts of r. The first part, $(a+b)^*$, stands for any string of a's and b's. The second part, $(a + bb)$ represents either an a or a double b. Consequently, $L(r)$ is the set of all strings on $\{a, b\}$, terminated by either an a or a bb. ■

Example 3.4 The expression

$$r = (aa)^* (bb)^* b$$

denotes the set of all strings with an even number of a's followed by an odd number of b's; that is

$$L(r) = \left\{a^{2n} b^{2m+1} : n \geq 0,\ m \geq 0\right\}.$$

Going from an informal description or set notation to a regular expression tends to be a little harder. ■

Example 3.5 For $\Sigma = \{0, 1\}$, give a regular expression r such that

$$L(r) = \{w \in \Sigma^* : w \text{ has at least one pair of consequtive zeros}\}.$$

One can arrive at an answer by reasoning something like this: Every string in $L(r)$ must contain 00 somewhere, but what comes before and what goes after is completely arbitrary. An arbitrary string on $\{0, 1\}$ can be denoted by $(0+1)^*$. Putting these observations together, we arrive at the solution

$$r = (0+1)^* \, 00 \, (0+1)^* .$$

■

Example 3.6 Find a regular expression for the language

$$L = \left\{w \in \{0, 1\}^* : w \text{ has no pair of consecutive zeros}\right\}.$$

Even though this looks similar to Example 3.5, the answer is harder to construct. One helpful observation is that whenever a 0 occurs, it must be followed immediately by a 1. Such a substring may be preceded and followed by an arbitrary number of 1's. This suggests that the answer involves the repetition of strings of the form $1 \cdots 101 \cdots 1$, that is, the language denoted by the regular expression $(1^*011^*)^*$. However, the answer is still incomplete, since the strings ending in 0 or consisting of all 1's are unaccounted for. After taking care of these special cases we arrive at the answer

$$r = (1^*011^*)^* (0 + \lambda) + 1^* (0 + \lambda).$$

If we reason slightly differently, we might come up with another answer. If we see L as the repetition of the strings 1 and 01, the shorter expression

$$r = (1 + 01)^* (0 + \lambda)$$

might be reached. Although the two expressions look different, both answers are correct, as they denote the same language. Generally, there are an unlimited number of regular expressions for any given language.

Note that this language is the complement of the language in Example 3.5. However, the regular expressions are not very similar and do not suggest clearly the close relationship between the languages.

■

The last example introduces the notion of equivalence of regular expressions. We say the two regular expressions are equivalent if they denote the same language. One can derive a variety of rules for simplifying regular expressions (see Exercise 18 in the following exercise section), but since we have little need for such manipulations we will not pursue this.

EXERCISES

1. Find all strings in $L((a+b)^* b(a+ab)^*)$ of length less than four.

2. Does the expression $((0+1)(0+1)^*)^* 00(0+1)^*$ denote the language in Example 3.5? ●

3. Show that $r = (1+01)^*(0+1^*)$ also denotes the language in Example 3.6. Find two other equivalent expressions.

4. Find a regular expression for the set $\{a^n b^m : (n+m) \text{ is even}\}$.

5. Give regular expressions for the following languages.

(a) $L_1 = \{a^n b^m, n \geq 4, m \leq 3\}$, ●

(b) $L_2 = \{a^n b^m : n < 4, m \leq 3\}$,

(c) The complement of L_1, ●

(d) The complement of L_2.

6. What languages do the expressions $(\varnothing^*)^*$ and $a\varnothing$ denote?

7. Give a simple verbal description of the language $L((aa)^* b(aa)^* + a(aa)^* ba(aa)^*)$.

8. Give a regular expression for L^R, where L is the language in Exercise 1.

9. Give a regular expression for $L = \{a^n b^m : n \geq 1, m \geq 1, nm \geq 3\}$. ●

10. Find a regular expression for $L = \{ab^n w : n \geq 3, w \in \{a,b\}^+\}$.

11. Find a regular expression for the complement of the language in Example 3.4.

12. Find a regular expression for $L = \{vwv : v, w \in \{a,b\}^*, |v| = 2\}$. ●

13. Find a regular expression for

$$L = \{w \in \{0,1\}^* : w \text{ has exactly one pair of consecutive zeros}\}.$$

14. Give regular expressions for the following languages on $\Sigma = \{a,b,c\}$.

(a) all strings containing exactly one a,

(b) all strings containing no more than three a's,

(c) all strings that contain at least one occurrence of each symbol in Σ, ●

(d) all strings that contain no run of a's of length greater than two,

★ (e) all strings in which all runs of a's have lengths that are multiples of three.

15. Write regular expressions for the following languages on $\{0, 1\}$.

(a) all strings ending in 01,

(b) all strings not ending in 01,

(c) all strings containing an even number of 0's, ●

(d) all strings having at least two occurrences of the substring 00 (Note that with the usual interpretation of a substring, 000 contains two such occurrences),

(e) all strings with at most two occurrences of the substring 00,

★ (f) all strings not containing the substring 101.

16. Find regular expressions for the following languages on $\{a, b\}$.

(a) $L = \{w : |w| \bmod 3 = 0\}$ ●

(b) $L = \{w : n_a(w) \bmod 3 = 0\}$

(c) $L = \{w : n_a(w) \bmod 5 > 0\}$

17. Repeat parts (a), (b), and (c) of Exercise 16, with $\Sigma = \{a, b, c\}$.

18. Determine whether or not the following claims are true for all regular expressions r_1 and r_2. The symbol $\equiv$ stands for equivalence of regular expressions in the sense that both expressions denote the same language.

(a) $(r_1^*)^* \equiv r_1^*$,

(b) $r_1^*(r_1 + r_2)^* \equiv (r_1 + r_2)^*$,

(c) $(r_1 + r_2)^* \equiv (r_1^* r_2^*)^*$, ●

(d) $(r_1 r_2)^* \equiv r_1^* r_2^*$.

19. Give a general method by which any regular expression r can be changed into $\widehat{r}$ such that $(L(r))^R = L(\widehat{r})$.

20. Prove rigorously that the expressions in Example 3.6 do indeed denote the specified language.

21. For the case of a regular expression r that does not involve λ or $\varnothing$, give a set of necessary and sufficient conditions that r must satisfy if $L(r)$ is to be infinite. ●

22. Formal languages can be used to describe a variety of two-dimensional figures. Chain-code languages are defined on the alphabet $\Sigma = \{u, d, r, l\}$, where these symbols stand for unit-length straight lines in the directions up, down, right, and left, respectively. An example of this notation is $urdl$, which stands for the square with sides of unit length. Draw pictures of the figures denoted by the expressions $(rd)^*$, $(urddru)^*$, and $(ruldr)^*$.

23. In Exercise 22, what are sufficient conditions on the expression so that the picture is a closed contour in the sense that the beginning and ending point are the same? Are these conditions also necessary? ●

24. Find an nfa that accepts the language $L(aa^*(a+b))$.

25. Find a regular expression that denotes all bit strings whose value, when interpreted as a binary integer, is greater than or equal to 40. ●

26. Find a regular expression for all bit strings, with leading bit 1, interpreted as a binary integer, with values not between 10 and 30.

3.2 Connection Between Regular Expressions and Regular Languages

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some dfa. Because of the equivalence of nfa's and dfa's, a language is also regular if it is accepted by some nfa. We now show that if we have any regular expression r, we can construct an nfa that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of Definition 3.2 on page 73, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Theorem 3.1 Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof: We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure 3.1(a), (b), and (c), respectively. Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in Figure 3.2. In this schema, the graph vertex at the left represents the initial state, the one on the right the final state. In Exercise 7, Section 2.3 we claimed that for every nfa there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, r_1r_2, and r_1^*. The constructions are shown in Figures 3.3 to 3.5. As indicated

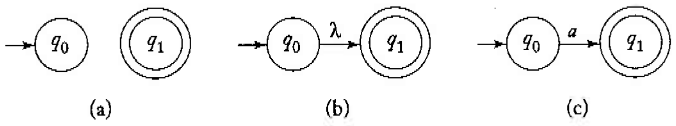


Figure 3.1
(a) nfa accepts $\varnothing$.
(b) nfa accepts $\{\lambda\}$.
(c) nfa accepts $\{a\}$.

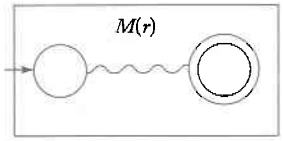


Figure 3.2
Schematic representation of an nfa accepting $L(r)$.

in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

It should be clear from the interpretation of the graphs in Figures 3.3 to 3.5 that this construction works. To argue more rigorously, we can give a formal method for constructing the states and transitions of the combined machine from the states and transitions of the parts, then prove by induction on the number of operators that the construction yields an automaton that accepts the language denoted by any particular regular expression. We will not belabor this point, as it is reasonably obvious that the results are always correct. ■

Example 3.7 Find an nfa which accepts $L(r)$, where

$$r = (a + bb)^* (ba^* + \lambda).$$

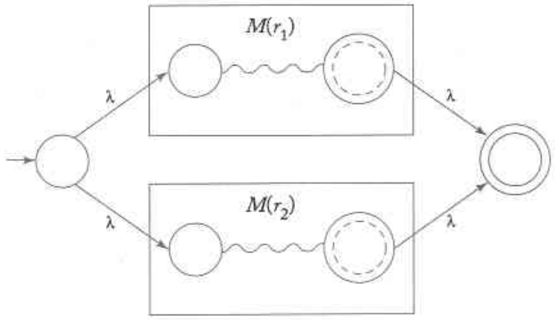


Figure 3.3
Automaton for $L(r_1 + r_2)$.

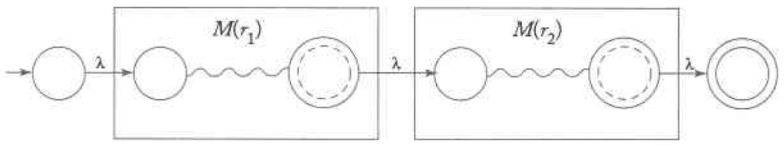


Figure 3.4 Automaton for $L(r_1r_2)$.

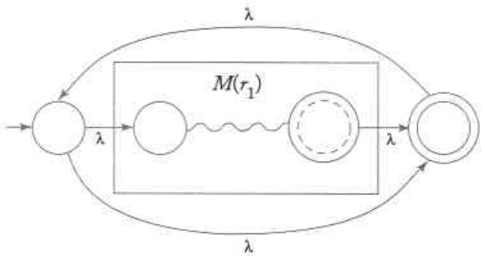


Figure 3.5 Automaton for $L(r_1^*)$.

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in Figure 3.6. Putting these together using the construction in Theorem 3.1, we get the solution in Figure 3.7 ■

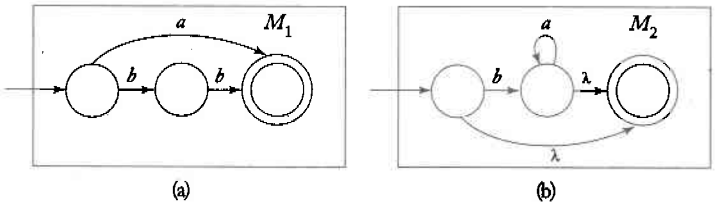


Figure 3.6 (a) M_1 accepts $L(a + bb)$. (b) M_2 accepts $L(ba^* + \lambda)$.

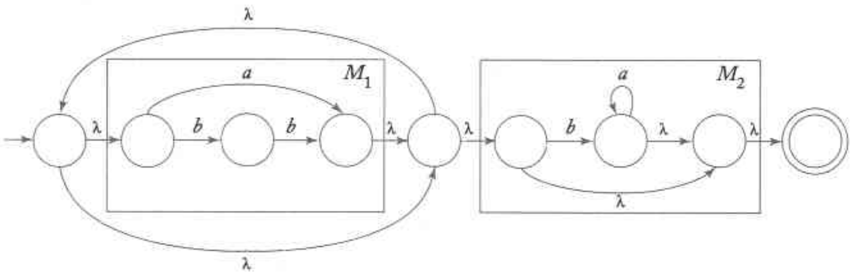


Figure 3.7 Automaton accepts $L((a + bb)^*(ba^* + \lambda))$.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated nfa and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called **generalized transition graphs.** Since this idea is used here in a limited way and plays no role in our further discussion, we will deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Example 3.8 Figure 3.8 represents a generalized transition graph. The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph. ■

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a

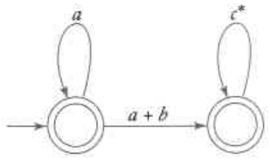
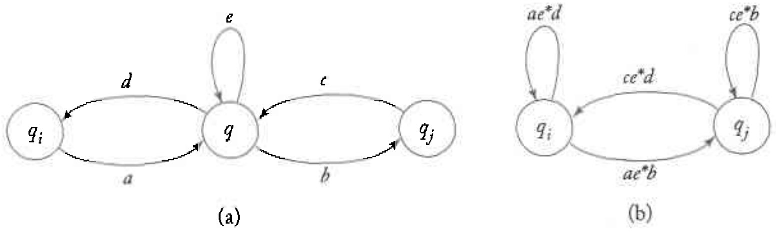


Figure 3.8

Figure 3.9



generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1. However, there are some subtleties in the argument; we will not pursue them here, but refer the reader instead to Exercise 16, Section 4.3 for details.

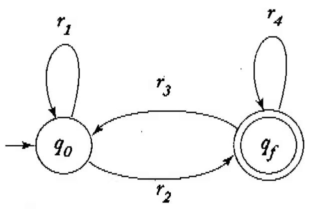
Equivalence for generalized transition graphs is defined in terms of the language accepted. Consider a generalized transition graph with states $\{q, q_i, q_j, ...\}$, where q is neither a final nor an initial state, and for which we want to create an equivalent generalized transition graph with one less state by removing q. We can do this if we do not change the language denoted by the set of labels that can be generated as we go from q_0 to q_f. The construction that achieves this is illustrated in Figure 3.9, where the state q is to be removed and the edge labels $a, b, ...$ stand for general expressions. The case depicted is the most general in the sense that q has outgoing edges to all three vertices q_i, q_j, q. In cases where an edge is missing in (a), we omit the corresponding edge in (b).

The construction in Figure 3.9 shows which edges have to be introduced so that the language of the generalized transition graph does not change when we remove q and all its incoming and outgoing edges. The complete process requires that this be done for all pairs (q_i, q_j) in $Q - \{q\}$ before removing q. Although we will not formally prove this, it can be shown that the construction yields an equivalent generalized transition graph. Accepting this, we are ready to show how any nfa can be associated with a regular expression.

Theorem 3.2 Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof: Let M be an nfa that accepts L. We can assume without any loss of generality that M has only one final state and that $q_0 \notin F$. We interpret the graph of M as a generalized transition graph and apply the above construction to it. To remove a vertex labeled q, we use the scheme in Figure 3.9 for all pairs (q_i, q_j). After all the new edges have been added,

Figure 3.10



$r_1^* r_2 (r_4 + r_3 r_1^* r_2)^*$

q with all its incident edges can be removed. We continue this process, removing one vertex after the other, until we reach the situation shown in Figure 3.10. A regular expression that denotes the language accepted by this graph is

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* . \tag{3.1}$$

Since the sequence of generalized transition graphs are all equivalent to the initial one, we can prove by an induction on the number of states in the generalized transition graph that the regular expression in (3.1) denotes L. ■

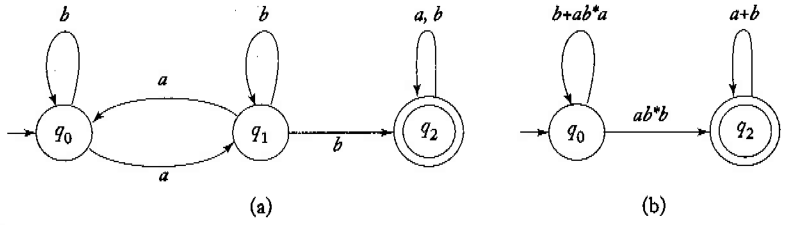
Example 3.9 Consider the nfa in Figure 3.11(a). The corresponding generalized transition graph after removal of state q_1 is shown in Figure 3.11(b). Making the identification $r_1 = b + ab^*a$, $r_2 = ab^*b$, $r_3 = \varnothing$, $r_4 = a + b$, we arrive at the regular expression

$$r = (b + ab^*a)^* \, ab^*b \, (a + b)^*$$

for the original automaton. The construction involved in Theorem 3.2 is tedious and tends to give very lengthy answers, but it is completely routine and always works.

■

Figure 3.11



Example 3.10 Find a regular expression for the language

$$L = \{w \in \{a,b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding a nfa for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution in Figure 3.12.

We can now apply the conversion to a regular expression in a mechanical way. First, we remove the state labeled OE, giving the generalized transition graph in Figure 3.13.

Next, we remove the vertex labeled OO. This gives Figure 3.14. Finally, we apply (3.1) with

$$\begin{aligned} r_1 &= aa + ab(bb)^* ba, \\ r_2 &= b + ab(bb)^* a, \\ r_3 &= b + a(bb)^* ba, \\ r_4 &= a(bb)^* a. \end{aligned}$$

Figure 3.12

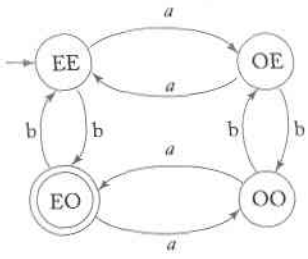


Figure 3.13

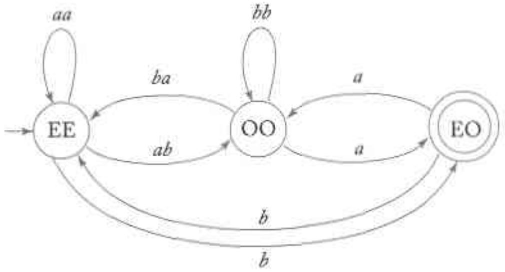
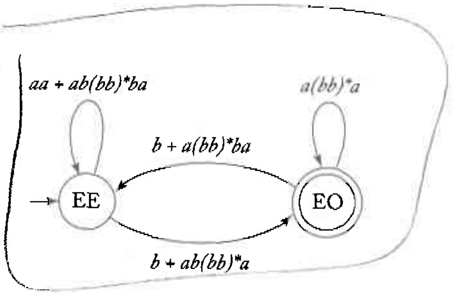


Figure 3.14



The final expression is long and complicated, but the way to get it is relatively straightforward. ■

Regular Expressions for Describing Simple Patterns

In Example 1.15 and in Exercise 15, Section 2.1, we explored the connection between finite accepters and some of the simpler constituents of programming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, the set of all acceptable Pascal integers is defined by the regular expression

sign digit

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9.

Pascal integers are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The example below is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Example 3.11 An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the editor *ed* in the UNIX operating system recognizes the command

$$\boxed{/aba^*c/}$$

as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example that the UNIX editor can recognize regular expressions (although it uses a somewhat different convention for specifying regular expressions than the one used here).

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent nfa using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a dfa. This dfa, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must be considered also. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Section 2.4 is helpful. ■

Exercises

1. Use the construction in Theorem 3.1 to find an nfa that accepts the language $L(ab^*aa + bba^*ab)$.

2. Find an nfa that accepts the complement of the language in Exercise 1.

3. Give an nfa that accepts the language $L((a+b)^* b (a+bb)^*)$. ●

4. Find dfa's that accept the following languages.

   (a) $L(aa^* + aba^*b^*)$ ●

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

   (c) $L((abab)^* + (aaa^* + b)^*)$

   (d) $L(((aa^*)^* b)^*)$

5. Find dfa's that accept the following languages.

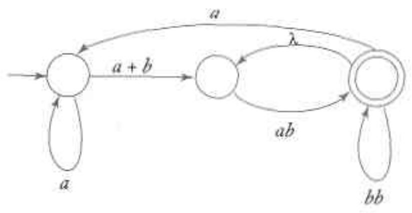
   (a) $L = L(ab^*a^*) \cup L((ab)^*\, ba)$,

   (b) $L = L(ab^*a^*) \cap L((ab)^*\, ba)$.

6. Find an nfa for Exercise 15(f), Section 3.1. Use this to derive a regular expression for that language.

7. Give explicit rules for the construction suggested in Figure 3.9 when various edges in 3.9(a) are missing. ●

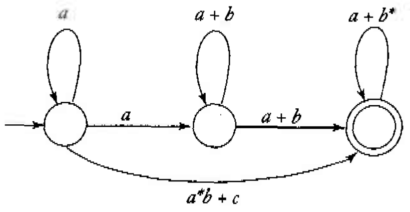
8. Consider the following generalized transition graph.



   (a) Find an equivalent generalized transition graph with only two states. ●

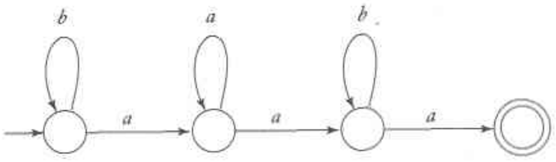
   (b) What is the language accepted by this graph? ●

9. What language is accepted by the following generalized transition graph?

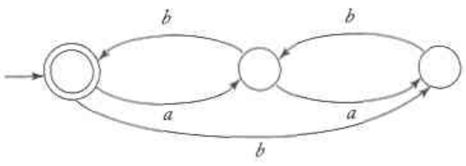


10. Find regular expressions for the languages accepted by the following automata.

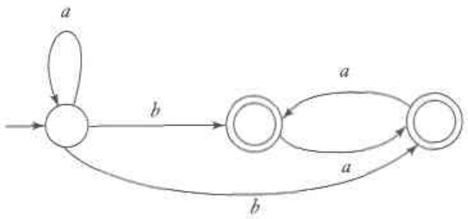
(a)



(b)



(c)



11. Rework Example 3.10, this time eliminating the state OO first.

12. Find a regular expression for the following languages on $\{a, b\}$.

(a) $L = \{w : n_a(w) \text{ and } n_b(w) \text{ are both even}\}$

(b) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 = 1\}$

(c) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 \neq 0\}$

(d) $L = \{w : 2n_a(w) + 3n_b(w) \text{ is even}\}$

13. Find a regular expression that generates the set of all strings of triplets defining correct binary addition as in Exercise 23, Section 2.1.

14. Prove that the constructions suggested by Figure 3.9 generate equivalent generalized transition graphs.

15. Write a regular expression for the set of all Pascal real numbers.

16. Find a regular expression for Pascal sets whose elements are integer numbers.

17. In some applications, such as programs that check spelling, we may not need an exact match of the pattern, only an approximate one. Once the notion

of an approximate match has been made precise, automata theory can be applied to construct approximate pattern matchers. As an illustration of this, consider patterns derived from the original ones by insertion of one symbol. Let L be a regular language on Σ and define

$$insert\,(L) = \{uav : a \in \Sigma, uv \in L\}.$$

In effect, $insert\,(L)$ contains all the words created from L by inserting a spurious symbol anywhere in a word.

★ (a) Given an nfa for L, show how one can construct an nfa for $insert\,(L)$. ●

★★ (b) Discuss how you might use this to write a pattern-recognition program for $insert\,(L)$, using as input a regular expression for L.

★ 18. Analogous to the previous exercise, consider all words that can be formed from L by dropping a single symbol of the string. Formally define this operation *drop* for languages. Construct an nfa for $drop\,(L)$, given an nfa for L.

19. Use the construction in Theorem 3.1 to find nfa's for $L\,(a\varnothing)$ and $L\,(\varnothing^*)$. Is the result consistent with the definition of these languages?

3.3 Regular Grammars

A third way of describing regular languages is by means of certain simple grammars. Grammars are often an alternative way of specifying languages. Whenever we define a language family through an automaton or in some other way, we are interested in knowing what kind of grammar we can associate with the family. First, we look at grammars that generate regular languages.

Right- and Left-Linear Grammars

Definition 3.3

A grammar $G = (V, T, S, P)$ is said to be **right-linear** if all productions are of the form

$$A \to xB,$$
$$A \to x,$$

where $A, B \in V$, and $x \in T^*$. A grammar is said to be **left-linear** if all productions are of the form

$$A \to Bx,$$

or

$$A \to x.$$

A **regular grammar** is one that is either right-linear or left-linear.

Note that in a regular grammar, at most one variable appears on the right side of any production. Furthermore, that variable must consistently be either the rightmost or leftmost symbol of the right side of any production.

Example 3.12 The grammar $G_1 = (\{S\}, \{a, b\}, S, P_1)$, with P_1 given as

$$S \to abS|a$$

is right-linear. The grammar $G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$, with productions

$$S \to S_1ab,$$
$$S_1 \to S_1ab|S_2,$$
$$S_2 \to a,$$

is left-linear. Both G_1 and G_2 are regular grammars.

The sequence

$$S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa$$

is a derivation with G_1. From this single instance it is easy to conjecture that $L(G_1)$ is the language denoted by the regular expression $r = (ab)^* a$. In a similar way, we can see that $L(G_2)$ is the regular language $L\left(aab(ab)^*\right)$. ■

Example 3.13 The grammar $G = (\{S, A, B\}, \{a, b\}, S, P)$ with productions

$$S \to A,$$
$$A \to aB|\lambda,$$
$$B \to Ab,$$

is not regular. Although every production is either in right-linear or left-linear form, the grammar itself is neither right-linear nor left-linear, and therefore is not regular. The grammar is an example of a **linear grammar.** A linear grammar is a grammar in which at most one variable can occur on the right side of any production, without restriction on the position of this variable. Clearly, a regular grammar is always linear, but not all linear grammars are regular.

■

Our next goal will be to show that regular grammars are associated with regular languages and that for every regular language there is a regular grammar. Thus, regular grammars are another way of talking about regular languages.

Right-Linear Grammars Generate Regular Languages

First, we show that a language generated by a right-linear grammar is always regular. To do so, we construct an nfa that mimics the derivations of a right-linear grammar. Note that the sentential forms of a right-linear grammar have the special form in which there is exactly one variable and it occurs as the rightmost symbol. Suppose now that we have a step in a derivation

$$ab \cdots cD \Rightarrow ab \cdots cdE,$$

arrived at by using a production $D \rightarrow dE$. The corresponding nfa can imitate this step by going from state D to state E when a symbol d is encountered. In this scheme, the state of the automaton corresponds to the variable in the sentential form, while the part of the string already processed is identical to the terminal prefix of the sentential form. This simple idea is the basis for the following theorem.

Theorem 3.3 Let $G = (V, T, S, P)$ be a right-linear grammar. Then $L(G)$ is a regular language.

Proof: We assume that $V = \{V_0, V_1, ...\}$, that $S = V_0$, and that we have productions of the form $V_0 \rightarrow v_1 V_i, V_i \rightarrow v_2 V_j, ...$ or $V_n \rightarrow v_l, ...$ If w is a string in $L(G)$, then because of the form of the productions in G, the derivation must have the form

$$\begin{aligned} V_0 &\Rightarrow v_1 V_i \\ &\Rightarrow v_1 v_2 V_j \\ &\overset{*}{\Rightarrow} v_1 v_2 \cdots v_k V_n \\ &\Rightarrow v_1 v_2 \cdots v_k v_l = w. \end{aligned} \tag{3.2}$$

The automaton to be constructed will reproduce the derivation by "consuming" each of these v's in turn. The initial state of the automaton will be labeled V_0, and for each variable V_i there will be a nonfinal state labeled V_i. For each production

$$V_i \rightarrow a_1 a_2 \cdots a_m V_j,$$

the automaton will have transitions to connect V_i and V_j that is, δ will be defined so that

$$\delta^* (V_i, a_1 a_2 \cdots a_m) = V_j.$$

For each production

$$V_i \rightarrow a_1 a_2 \cdots a_m,$$

the corresponding transition of the automaton will be

$$\delta^* (V_i, a_1 a_2 \cdots a_m) = V_f,$$

where V_f is a final state. The intermediate states that are needed to do this are of no concern and can be given arbitrary labels. The general scheme is shown in Figure 3.15. The complete automaton is assembled from such individual parts.

Suppose now that $w \in L(G)$ so that (3.2) is satisfied. In the nfa there is, by construction, a path from V_0 to V_i labeled v_1, a path from V_i to V_j labeled v_2, and so on, so that clearly

$$V_f \in \delta^* (V_0, w),$$

and w is accepted by M.

Conversely, assume that w is accepted by M. Because of the way in which M was constructed, to accept w the automaton has to pass through a sequence of states $V_0, V_i, ...$ to V_f, using paths labeled $v_1, v_2, ...$ Therefore, w must have the form

$$w = v_1 v_2 \cdots v_k v_l$$

Figure 3.15

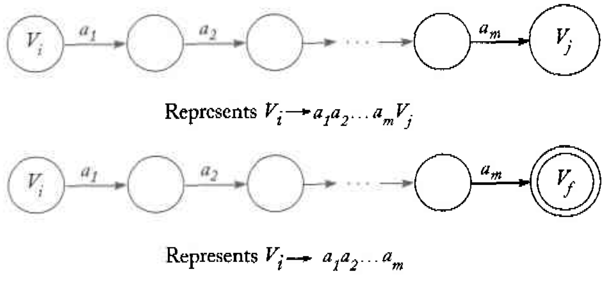
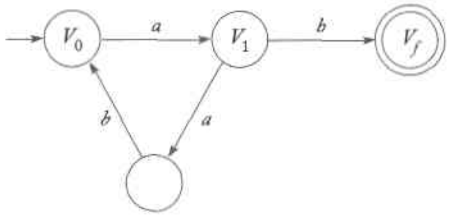

V_i
a_1
a_2
a_m
V_j
Represents $V_i \rightarrow a_1 a_2 \ldots a_m V_j$
V_i
a_1
a_2
a_m
V_f
Represents $V_i \rightarrow a_1 a_2 \ldots a_m$

Figure 3.16



and the derivation

$$V_o \Rightarrow v_1 V_i \Rightarrow v_1 v_2 V_j \overset{*}{\Rightarrow} v_1 v_2 \cdots v_k V_k \Rightarrow v_1 v_2 \cdots v_k v_l$$

is possible. Hence w is in $L(G)$, and the theorem is proved. ■

Example 3.14 Construct a finite automaton that accepts the language generated by the grammar

$$V_0 \rightarrow aV_1,$$
$$V_1 \rightarrow abV_0 | b.$$

We start the transition graph with vertices V_0, V_1, and V_f. The first production rule creates an edge labeled a between V_0 and V_1. For the second rule, we need to introduce an additional vertex so that there is a path labeled ab between V_1 and V_0. Finally, we need to add an edge labeled b between V_1 and V_f, giving the automaton shown in Figure 3.16. The language generated by the grammar and accepted by the automaton is the regular language $L\left((aab)^* ab\right)$.

■

Right-Linear Grammars for Regular Languages

To show that every regular language can be generated by some right-linear grammar, we start from the dfa for the language and reverse the construction shown in Theorem 3.3. The states of the dfa now become the variables of the grammar, and the symbols causing the transitions become the terminals in the productions.

Theorem 3.4 If L is a regular language on the alphabet Σ, then there exists a right-linear grammar $G = (V, \Sigma, S, P)$ such that $L = L(G)$.

Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a dfa that accepts L. We assume that $Q = \{q_0, q_1, ..., q_n\}$ and $\Sigma = \{a_1, a_2, ..., a_m\}$. Construct the right-linear grammar $G = (V, \Sigma, S, P)$ with

$$V = \{q_0, q_1, ..., q_n\}$$

and $S = q_0$. For each transition

$$\delta(q_i, a_j) = q_k$$

of M, we put in P the production

$$q_i \rightarrow a_j q_k. \tag{3.3}$$

In addition, if q_k is in F, we add to P the production

$$q_k \rightarrow \lambda. \tag{3.4}$$

We first show that G defined in this way can generate every string in L. Consider $w \in L$ of the form

$$w = a_i a_j \cdots a_k a_l.$$

For M to accept this string it must make moves via

$$\begin{aligned} \delta(q_0, a_i) &= q_p, \\ \delta(q_p, a_j) &= q_r, \\ &\vdots \\ \delta(q_s, a_k) &= q_t, \\ \delta(q_t, a_l) &= q_f \in F. \end{aligned}$$

By construction, the grammar will have one production for each of these δ's. Therefore we can make the derivation

$$\begin{aligned} q_0 &\Rightarrow a_i q_p \Rightarrow a_i a_j q_r \overset{*}{\Rightarrow} a_i a_j \cdots a_k q_t \\ &\Rightarrow a_i a_j \cdots a_k a_l q_f \Rightarrow a_i a_j \cdots a_k a_l, \end{aligned} \tag{3.5}$$

with the grammar G, and $w \in L(G)$.

Conversely, if $w \in L(G)$, then its derivation must have the form (3.5). But this implies that

$$\delta^*(q_0, a_i a_j \cdots a_k a_l) = q_f,$$

completing the proof. ■

Figure 3.17

$\delta(q_0, a) = \{q_1\}$	$q_0 \rightarrow aq_1$
$\delta(q_1, a) = \{q_2\}$	$q_1 \rightarrow aq_2$
$\delta(q_2, b) = \{q_2\}$	$q_2 \rightarrow bq_2$
$\delta(q_2, a) = \{q_f\}$	$q_2 \rightarrow aq_f$
$q_f \in F$	$q_f \rightarrow \lambda$

For the purpose of constructing a grammar, it is useful to note that the restriction that M be a dfa is not essential to the proof of Theorem 3.4. With minor modification, the same construction can be used if M is an nfa.

Example 3.15 Construct a right-linear grammar for $L(aab^*a)$. The transition function for an nfa, together with the corresponding grammar productions, is given in Figure 3.17. The result was obtained by simply following the construction in Theorem 3.4. The string $aaba$ can be derived with the constructed grammar by

$$q_0 \Rightarrow aq_1 \Rightarrow aaq_2 \Rightarrow aabq_2 \Rightarrow aabaq_f \Rightarrow aaba.$$

■

Equivalence Between Regular Languages and Regular Grammars

The previous two theorems establish the connection between regular languages and right-linear grammars. One can make a similar connection between regular languages and left-linear grammars, thereby showing the complete equivalence of regular grammars and regular languages.

Theorem 3.5 A language L is regular if and only if there exists a left-linear grammar G such that $L = L(G)$.

Proof: We only outline the main idea. Given any left-linear grammar with productions of the form

$$A \rightarrow Bv,$$

or

$$A \rightarrow v,$$

we construct from it a right-linear grammar $\widehat{G}$ by replacing every such production of G with

$$A \to v^R B,$$

or

$$A \to v^R,$$

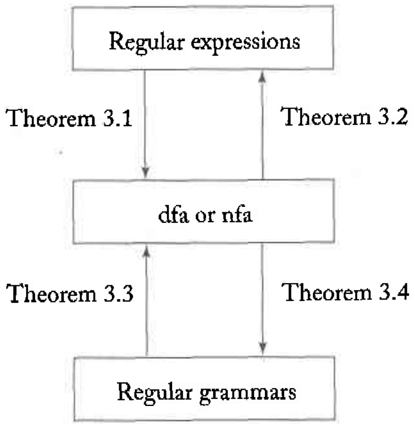
respectively. A few examples will make it clear quickly that $L(G) = \left(L\left(\widehat{G}\right)\right)^R$. Next, we use Exercise 12, Section 2.3, which tells us that the reverse of any regular language is also regular. Since $\widehat{G}$ is right-linear, $L\left(\widehat{G}\right)$ is regular. But then so are $L\left(\left(\widehat{G}\right)\right)^R$ and $L(G)$. ■

Putting Theorems 3.4 and 3.5 together, we arrive at the equivalence of regular languages and regular grammars.

Theorem 3.6 A language L is regular if and only if there exists a regular grammar G such that $L = L(G)$.

We now have several ways of describing regular languages: dfa's, nfa's, regular expressions, and regular grammars. While in some instance one or the other of these may be most suitable, they are all equally powerful. They all give a complete and unambiguous definition of a regular language. The connection between all these concepts is established by the four theorems in this chapter, as shown in Figure 3.18.

Figure 3.18



Exercises

1. Construct a dfa that accepts the language generated by the grammar

$$S \to abA,$$
$$A \to baB,$$
$$B \to aA|bb.$$

2. Find a regular grammar that generates the language $L(aa^*(ab+a)^*)$.

3. Construct a left-linear grammar for the language in Exercise 1.

۳ **4.** Construct right- and left-linear grammars for the language

$$L = \{a^n b^m : n \geq 2, m \geq 3\}. \; ●$$

5. Construct a right-linear grammar for the language $L((aab^*ab)^*)$.

۶ **6.** Find a regular grammar that generates the language on $\Sigma = \{a, b\}$ consisting of all strings with no more than three a's.

7. In Theorem 3.5, prove that $L(\widehat{G}) = (L(G))^R$. ●

۸ **8.** Suggest a construction by which a left-linear grammar can be obtained from an nfa directly.

9. Find a left-linear grammar for the language in Exercise 5.

۹ **10.** Find a regular grammar for the language $L = \{a^n b^m : n + m \text{ is even}\}$. ●

11. Find a regular grammar that generates the language

$$L = \{w \in \{a, b\}^* : n_a(w) + 3n_b(w) \text{ is even}\}.$$

۱۰ **12.** Find regular grammars for the following languages on $\{a, b\}$.

(a) $L = \{w : n_a(w) \text{ and } n_b(w) \text{ are both even}\}$ ●

(b) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 = 1\}$

(c) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 \neq 0\}$

(d) $L = \{w : |n_a(w) - n_b(w)| \text{ is odd}\}$.

۱۱ **13.** Show that for every regular language not containing λ there exists a right-linear grammar whose productions are restricted to the forms

$$A \to aB$$

or

$$A \to a,$$

where A, $B \in V$ and $a \in T$.

14. Show that any regular grammar G for which $L(G) \neq \varnothing$ must have at least one production of the form

$$A \to x,$$

where $A \in V$ and $x \in T^*$.

15. Find a regular grammar that generates the set of all Pascal real numbers.

16. Let $G_1 = (V_1, \Sigma, S_1, P_1)$ be right-linear and $G_2 = (V_2, \Sigma, S_2, P_2)$ be a left linear grammar, and assume that V_1 and V_2 are disjoint. Consider the linear grammar $G = (\{S\} \cup V_1 \cup V_2, \Sigma, S, P)$, where S is not in $V_1 \cup V_2$ and $P = \{S \to S_1 | S_2\} \cup P_1 \cup P_2$. Show that $L(G)$ is regular. ●