CHAPTER 4

Finite-State Machines and Pushdown Automata

The finite-state machine (FSM) and the pushdown automaton (PDA) enjoy a special place in computer science. The FSM has proven to be a very useful model for many practical tasks and deserves to be among the tools of every practicing computer scientist. Many simple tasks, such as interpreting the commands typed into a keyboard or running a calculator, can be modeled by finite-state machines. The PDA is a model to which one appeals when writing compilers because it captures the essential architectural features needed to parse context-free languages, languages whose structure most closely resembles that of many programming languages.

In this chapter we examine the language recognition capability of FSMs and PDAs. We show that FSMs recognize exactly the regular languages, languages defined by regular expressions and generated by regular grammars. We also provide an algorithm to find a FSM that is equivalent to a given FSM but has the fewest states.

We examine language recognition by PDAs and show that PDAs recognize exactly the context-free languages, languages whose grammars satisfy less stringent requirements than regular grammars. Both regular and context-free grammar types are special cases of the phrase-structure grammars that are shown in Chapter 5 to be the languages accepted by Turing machines.

It is desirable not only to classify languages by the architecture of machines that recognize them but also to have tests to show that a language is not of a particular type. For this reason we establish so-called pumping lemmas whose purpose is to show how strings in one language can be elongated or "pumped up." Pumping up may reveal that a language does not fall into a presumed language category. We also develop other properties of languages that provide mechanisms for distinguishing among language types. Because of the importance of context-free languages, we examine how they are parsed, a key step in programming language translation.

4.1 Finite-State Machine Models

The deterministic finite-state machine (DFSM), introduced in Section 3.1, has a set of states, including an initial state and one or more final states. At each unit of time a DFSM is given a letter from its input alphabet. This causes the machine to move from its current state to a potentially new state. While in a state, the DFSM produces a letter from its output alphabet. Such a machine computes the function defined by the mapping from strings of input letters to strings of output letters. DFSMs can also be used to accept strings. A string is accepted by a DFSM if the last state entered by the machine on that input string is a final state. The language recognized by a DFSM is the set of strings that it accepts.

Although there are languages that cannot be accepted by any machine with a finite number of states, it is important to note that all realistic computational problems are finite in nature and can be solved by FSMs. However, important opportunities to simplify computations may be missed if we do not view them as requiring potentially infinite storage, such as that provided by pushdown automata, machines that store data on a pushdown stack. (Pushdown automata are formally introduced in Section 4.8.)

The nondeterministic finite-state machine (NFSM) was also introduced in Section 3.1. The NFSM has the property that for a given state and input letter there may be several states to which it could move. Also for some state and input letter there may be no possible move. We say that an NFSM accepts a string if there is a sequence of next-state choices (see Section 3.1.5) that can be made, when necessary, so that the string causes the NFSM to enter a final state. The language accepted by such a machine is the set of strings it accepts.

Although nondeterminism is a useful tool in describing languages and computations, nondeterministic computations are very expensive to simulate deterministically: the deterministic simulation time can grow as an exponential function of the nondeterministic computation time. We explore nondeterminism here to gain experience with it. This will be useful in Chapter 8 when we classify languages by the ability of nondeterministic machines of infinite storage capacity to accept them. However, as we shall see, nondeterminism offers no advantage for finite-state machines in that both DFSMs and NFSMs recognize the same set of languages.

We now begin our formal treatment of these machine models. Since this chapter is concerned only with language recognition, we give an abbreviated definition of the deterministic FSM that ignores the output function. We also give a formal definition of the nondeterministic finite-state machine that agrees with that given in Section 3.1.5. We recall that we interpreted such a machine as a deterministic FSM that possesses a choice input through which a choice agent specifies the state transition to take if more than one is possible.

DEFINITION 4.1.1 *A deterministic* **finite-state machine (DFSM)** M *is a five-tuple* $M = (\Sigma, Q, \delta, s, F)$ *where* Σ *is the* **input alphabet**, Q *is the finite* **set of states**, $\delta : Q \times \Sigma \mapsto Q$ *is the* **next-state function**, s *is the* **initial state**, *and* F *is the* **set of final states**. *The DFSM* M **accepts the input string** $w \in \Sigma^*$ *if the last state entered by* M *on application of* w *starting in state* s *is a member of the set* F. M **recognizes the language** $L(M)$ *consisting of all such strings.*

A **nondeterministic FSM (NFSM)** *is similarly defined except that the next-state function* δ *is replaced by a* **next-set function** $\delta : Q \times \Sigma \mapsto 2^Q$ *that* **associates a set of states with each state-input pair** (q, a). *The NFSM* M **accepts the string** $w \in \Sigma^*$ *if there are next-state choices, whenever more than one exists, such that the last state entered under the input string* w *is a member of* F. M **accepts the language** $L(M)$ *consisting of all such strings.*

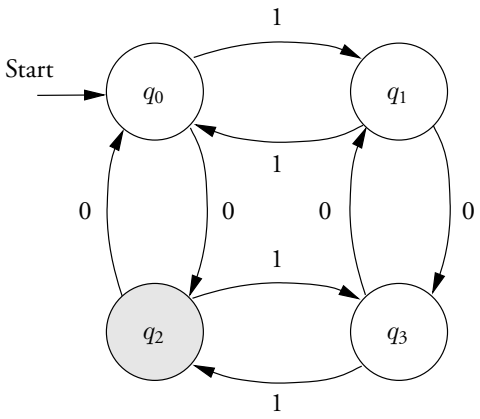


Figure 4.1 The deterministic finite-state machines $M_{\text{odd/even}}$ that accepts strings containing an odd number of 0's and an even number of 1's.

Figure 4.1 shows a DFSM $M_{\text{odd/even}}$ with initial state q_0. The final state is shown as a shaded circle; that is, $F = \{q_2\}$. $M_{\text{odd/even}}$ is in state q_0 or q_2 as long as the number of 1's in its input is even and is in state q_1 or q_3 as long as the number of 1's in its input is odd. Similarly, $M_{\text{odd/even}}$ is in state q_0 or q_1 as long as the number of 0's in its input is even and is in states q_2 or q_3 as long as the number of 0's in its input is odd. Thus, $M_{\text{odd/even}}$ recognizes the language of binary strings containing an odd number of 0's and an even number of 1's.

When the next-set function δ for an NFSM has value $\delta(q, a) = \emptyset$, the empty set, for state-input pair (q, a), no transition is specified from state q on input letter a.

Figure 4.2 shows a simple NFSM ND with initial state q_0 and final state set $F = \{q_0, q_3, q_5\}$. Nondeterministic transitions are possible from states q_0, q_3, and q_5. In addition, no transition is specified on input 0 from states q_1 and q_2 nor on input 1 from states q_0, q_3, q_4, or q_5.

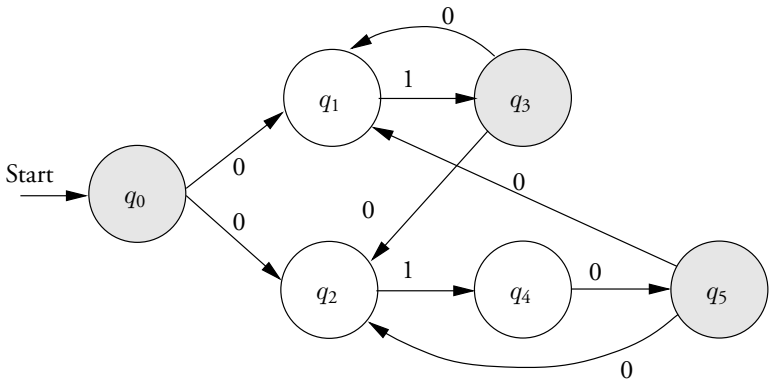


Figure 4.2 The nondeterministic machine ND.

4.2 Equivalence of DFSMs and NFSMs

Finite-state machines recognizing the same language are said to be **equivalent**. We now show that the class of languages accepted by DFSMs and NFSMs is the same. That is, for each NFSM there is an equivalent DFSM and vice versa. The proof has two symmetrical steps: a) given an arbitrary DFSM D_1 recognizing the language $L(D_1)$, we construct an NFSM N_1 that accepts $L(D_1)$, and b) given an arbitrary NFSM N_2 that accepts $L(N_2)$, we construct a DFSM D_2 that recognizes $L(N_2)$. The first half of this proof follows immediately from the fact that a DFSM is itself an NFSM. The second half of the proof is a bit more difficult and is stated below as a theorem. The method of proof is quite simple, however. We construct a DFSM D_2 that has one state for each set of states that the NFSM N_2 can reach on some input string and exhibit a next-state function for D_2. We illustrate this approach with the NFSM $N_2 = ND$ of Fig. 4.2.

Since the initial state of *ND* is q_0, the initial state of $D_2 = M_{\text{equiv}}$, the DFSM equivalent to *ND*, is the set $\{q_0\}$. In turn, because q_0 has two successor states on input 0, namely q_1 and q_2, we let $\{q_1, q_2\}$ be the successor to $\{q_0\}$ in M_{equiv} on input 0, as shown in the following table. Since q_0 has no successor on input 1, the successor to $\{q_0\}$ on input 1 is the empty set $\emptyset$. Building in this fashion, we find that the successor to $\{q_1, q_2\}$ on input 1 is $\{q_3, q_4\}$ whereas its successor on input 0 is $\emptyset$. The reader can complete the table shown below. Here q_{equiv} is the name of a state of the DFSM M_{equiv}.

q_{equiv}	a	$\delta_{M_{\text{equiv}}}(q_{\text{equiv}}, a)$
$\{q_0\}$	0	$\{q_1, q_2\}$
$\{q_0\}$	1	$\emptyset$
$\{q_1, q_2\}$	0	$\emptyset$
$\{q_1, q_2\}$	1	$\{q_3, q_4\}$
$\{q_3, q_4\}$	0	$\{q_1, q_2, q_5\}$
$\{q_3, q_4\}$	1	$\emptyset$
$\{q_1, q_2, q_5\}$	0	$\{q_1, q_2\}$
$\{q_1, q_2, q_5\}$	1	$\{q_3, q_4\}$

q_{equiv}	q
$\{q_0\}$	a
$\{q_1, q_2\}$	b
$\{q_3, q_4\}$	c
$\{q_1, q_2, q_5\}$	d
$\emptyset$	q_R

In the second table above, we provide a new label for each state q_{equiv} of M_{equiv}. In Fig. 4.3 we use these new labels to exhibit the DFSM M_{equiv} equivalent to the NFSM *ND* of Fig. 4.2. A final state of M_{equiv} is any set containing a final state of *ND* because a string takes M_{equiv} to such a set if and only if it can take *ND* to one of its final states. We now show that this method of constructing a DFSM from an NFSM always works.

THEOREM 4.2.1 *Let L be a language accepted by a nondeterministic finite-state machine M_1. There exists a deterministic finite-state machine M_2 that recognizes L.*

Proof Let $M_1 = (\Sigma, Q_1, \delta_1, s_1, F_1)$ be an NFSM that accepts the language L. We design a DFSM $M_2 = (\Sigma, Q_2, \delta_2, s_2, F_2)$ that also recognizes L. M_1 and M_2 have identical input alphabets, Σ. The states of M_2 are associated with subsets of the states of Q_1, which is denoted by $Q_2 \subseteq 2^{Q_1}$, where 2^{Q_1} is the power set of Q_1 containing all the subsets of Q_1, including the empty set. We let the initial state s_2 of M_2 be associated with the set $\{s_1\}$ containing the initial state of M_1. A state of M_2 is a set of states that M_1 can reach on a sequence of inputs. A final state of M_2 is a subset of Q_1 that contains a final state of M_1. For example, if $q_5 \in F_1$, then $\{q_2, q_5\} \in F_2$.

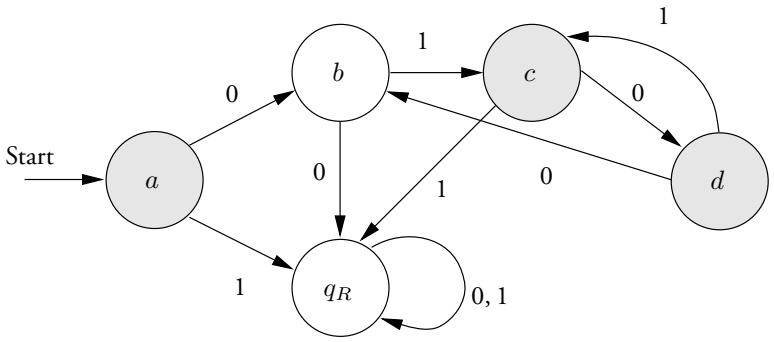


Figure 4.3 The DFSM M_{equiv} equivalent to the NFSM *ND*.

We first give an inductive definition of the states of M_2. Let $Q_2^{(k)}$ denote the sets of states of M_1 that can be reached from s_1 on input strings containing k or fewer letters. In the example given above, $Q_2^{(1)} = \{\{q_0\}, \{q_1, q_2\}, q_R\}$ and $Q_2^{(3)} = \{\{q_0\}, \{q_1, q_2\}, \{q_3, q_4\}, \{q_1, q_2, q_5\}, q_R\}$. To construct $Q_2^{(k+1)}$ from $Q_2^{(k)}$, we form the subset of Q_1 that can be reached on each input letter from a subset in $Q_2^{(k)}$, as illustrated above. If this is a new set, it is added to $Q_2^{(k)}$ to form $Q_2^{(k+1)}$. When $Q_2^{(k)}$ and $Q_2^{(k+1)}$ are the same, we terminate this process since no new subsets of Q_1 can be reached from s_1. This process eventually terminates because Q_2 has at most $2^{|Q_1|}$ elements. It terminates in at most $2^{|Q_1|} - 1$ steps because starting from the initial set $\{q_0\}$ at least one new subset must be added at each step.

The next-state function δ_2 of M_2 is defined as follows: for each state q of M_2 (a subset of Q_1), the value of $\delta_2(q, a)$ for input letter a is the state of M_2 (subset of Q_1) reached from q on input a. As the sets $Q_2^{(1)}, \ldots, Q_2^{(m)}$ are constructed, $m \leq 2^{|Q_1|} - 1$, we construct a table for δ_2.

We now show by induction on the length of an input string $\boldsymbol{z}$ that if $\boldsymbol{z}$ can take M_1 to a state in the set $S \subseteq Q_1$, then it takes M_2 to its state associated with S. It follows that if S contains a final state of M_1, then $\boldsymbol{z}$ is accepted by both M_1 and M_2.

The basis for the inductive hypothesis is the case of the empty input letter. In this case, s_1 is reached by M_1 if and only if $\{s_1\}$ is reached by M_2. The inductive hypothesis is that if $\boldsymbol{w}$ of length n can take M_1 to a state in the set S, then it takes M_2 to its state associated with S. We assume the hypothesis is true on inputs of length n and show that it remains true on inputs of length $n + 1$. Let $\boldsymbol{z} = \boldsymbol{w}a$ be an input string of length $n + 1$. To show that $\boldsymbol{z}$ can take M_1 to a state in S' if and only if it takes M_2 to the state associated with S', observe that by the inductive hypothesis there exists a set $S \subseteq Q_1$ such that $\boldsymbol{w}$ can take M_1 to a state in S if and only if it takes M_2 to the state associated with S. By the definition of δ_2, the input letter a takes the states of M_1 in S into states of M_1 in S' if and only if a takes the state of M_2 associated with S to the state associated with S'. It follows that the inductive hypothesis holds. ■

Up to this point we have shown equivalence between deterministic and nondeterministic FSMs. Another equivalence question arises in this context: It is, "Given an FSM, is there an equivalent FSM that has a smaller number of states?" The determination of an equivalent FSM

with the smallest number of states is called the **state minimization problem** and is explored in Section 4.7.

4.3 Regular Expressions

In this section we introduce regular expressions, algebraic expressions over sets of individual letters that describe the class of languages recognized by finite-state machines, as shown in the next section.

Regular expressions are formed through the concatenation, union, and Kleene closure of sets of strings. Given two sets of strings L_1 and L_2, their **concatenation** $L_1 \cdot L_2$ is the set $\{uv \mid u \in L_1 \text{ and } v \in L_2\}$; that is, the set of strings consisting of an arbitrary string of L_1 followed by an arbitrary string of L_2. (We often omit the concatenation operator $\cdot$, writing variables one after the other instead.) The **union** of L_1 and L_2, denoted $L_1 \cup L_2$, is the set of strings that are in L_1 or L_2 or both. The **Kleene closure** of a set L of strings, denoted L^* (also called the **Kleene star**), is defined in terms of the i-fold concatenation of L with itself, namely, $L^i = L \cdot L^{i-1}$, where $L^0 = \{\epsilon\}$, the set containing the empty string:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Thus, L^* is the union of strings formed by concatenating zero or more words of L. Finally, we define the **positive closure** of L to be the union of all i-fold products except for the zeroth, that is,

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

The positive closure is a useful shorthand in regular expressions.

An example is helpful. Let $L_1 = \{01, 11\}$ and $L_2 = \{0, aba\}$; then $L_1L_2 = \{010, 01aba, 110, 11aba\}$, $L_1 \cup L_2 = \{0, 01, 11, aba\}$, and

$$L_2^* = \{0, aba\}^* = \{\epsilon, 0, aba, 00, 0aba, aba0, abaaba, \ldots\}$$

Note that the definition given earlier for Σ^*, namely, the set of strings over the finite alphabet Σ, coincides with this new definition of the Kleene closure. We are now prepared to define regular expressions.

DEFINITION 4.3.1 **Regular expressions** *over the finite alphabet Σ and the languages they describe are defined recursively as follows:*

1. $\emptyset$ *is a regular expression denoting the empty set.*
2. ϵ *is a regular expression denoting the set $\{\epsilon\}$.*
3. *For each letter $a \in \Sigma$, a is a regular expression denoting the set $\{a\}$ containing a.*
4. *If r and s are regular expressions denoting the languages R and S, then (rs), $(r + s)$, and (r^*) are regular expressions denoting the languages $R \cdot S$, $R \cup S$, and R^*, respectively.*

The languages denoted by regular expressions are called **regular languages**. *(They are also often called regular sets.)*

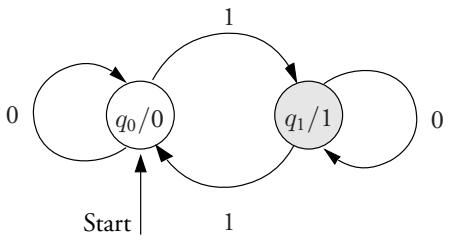


Figure 4.4 A finite-state machine computing the EXCLUSIVE OR of its inputs.

Some examples of regular expressions will clarify the definitions. The regular expression $(\mathbf{0}+\mathbf{1})^*$ denotes the set of all strings over the alphabet $\{0,1\}$. The expression $(\mathbf{0}^*)(\mathbf{1})$ denotes the strings containing zero or more 0's that end with a single 1. The expression $((\mathbf{1})(\mathbf{0}^*)(\mathbf{1})+\mathbf{0})^*$ denotes strings containing an even number of 1's. Thus, the expression $((\mathbf{0}^*)(\mathbf{1}))((\mathbf{1})(\mathbf{0}^*)(\mathbf{1})+\mathbf{0})^*$ denotes strings containing an odd number of 1's. This is exactly the class of strings recognized by the simple DFSM in Fig. 4.4. (So far we have set in boldface all regular expressions denoting sets containing letters. Since context will distinguish between a set containing a letter and the letter itself, we drop the boldface notation at this point.)

Some parentheses in regular expressions can be omitted if we give highest precedence to Kleene closure, next highest precedence to concatenation, and lowest precedence to union. For example, we can write $((0^*)(1))((1)(0^*)(1)+0)^*$ as $0^*1(10^*1+0)^*$.

Because regular expressions denote languages, certain combinations of union, concatenation, and Kleene closure operations on regular expressions can be rewritten as other combinations of operations. A regular expression will be treated as identical to the language it denotes. Two **regular expressions are equivalent** if they denote the same language. We now state properties of regular expressions, leaving their proof to the reader.

THEOREM 4.3.1 *Let $\emptyset$ and ϵ be the regular expressions denoting the empty set and the set containing the empty string and let r, s, and t be arbitrary regular expressions. Then the rules shown in Fig. 4.5 hold.*

We illustrate these rules with the following example. Let $a = 0^*1\cdot b+0^*$, where $b = c\cdot 10^+$ and $c = (0+10^+1)^*$. Using rule (16) of Fig. 4.5, we rewrite c as follows:

$$c = (0+10^+1)^* = (0^*10^+1)^*0^*$$

Then using rule (15) with $r = 0^*10^+$ and $s = 1$, we write b as follows:

$$b = (0^*10^+1)^*0^*10^+ = (rs)^*r = r(sr)^* = 0^*10^+(10^*10^+)^*$$

It follows that a satisfies

$$\begin{aligned} a &= 0^*1\cdot b+0^* \\ &= 0^*10^*10^+(10^*10^+)^*+0^* \\ &= 0^*(10^*10^+)^++0^* \\ &= 0^*((10^*10^+)^++\epsilon) \\ &= 0^*(10^*10^+)^* \end{aligned}$$

(1)	$r\emptyset$	$=$	$\emptyset r$	$=$	$\emptyset$
(2)	$r\epsilon$	$=$	ϵr	$=$	r
(3)	$r+\emptyset$	$=$	$\emptyset + r$	$=$	r
(4)	$r+r$	$=$	r		
(5)	$r+s$	$=$	$s+r$		
(6)	$r(s+t)$	$=$	$rs+rt$		
(7)	$(r+s)t$	$=$	$rt+st$		
(8)	$r(st)$	$=$	$(rs)t$		
(9)	$\emptyset^*$	$=$	ϵ		
(10)	ϵ^*	$=$	ϵ		
(11)	$(\epsilon+r)^+$	$=$	r^*		
(12)	$(\epsilon+r)^*$	$=$	r^*		
(13)	$r^*(\epsilon+r)$	$=$	$(\epsilon+r)r^*$	$=$	r^*
(14)	r^*s+s	$=$	r^*s		
(15)	$r(sr)^*$	$=$	$(rs)^*r$		
(16)	$(r+s)^*$	$=$	$(r^*s)^*r^*$	$=$	$(s^*r)^*s^*$

Figure 4.5 Rules that apply to regular expressions.

where we have simplified the expressions using the definition of the positive closure, namely $r(r^*) = r^+$ in the second equation and rules (6), (5), and (12) in the last three equations. Other examples of the use of the identities can be found in Section 4.4.

4.4 Regular Expressions and FSMs

Regular languages are exactly the languages recognized by finite-state machines, as we now show. Our two-part proof begins by showing (Section 4.4.1) that every regular language can be accepted by a nondeterministic finite-state machine. This is followed in Section 4.4.2 by a proof that the language recognized by an arbitrary deterministic finite-state machine can be described by a regular expression. Since by Theorem 4.2.1 the language recognition power of DFSMs and NFSMs are the same, the desired conclusion follows.

4.4.1 Recognition of Regular Expressions by FSMs

THEOREM 4.4.1 *Given a regular expression r over the set Σ, there is a nondeterministic finite-state machine that accepts the language denoted by r.*

Proof We show by induction on the size of a regular expression r (the number of its operators) that there is an NFSM that accepts the language described by r.

BASIS: If no operators are used, the regular expression is either ϵ, $\emptyset$, or a for some $a \in \Sigma$. The finite-state machines shown in Fig. 4.6 recognize these three languages.

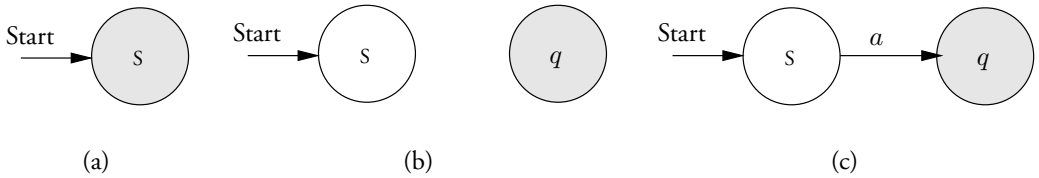


Figure 4.6 Finite-state machines recognizing the regular expressions ϵ, $\emptyset$, and a, respectively. In b) an output state is shown even though it cannot be reached.

INDUCTION: Assume that the hypothesis holds for all regular expressions r with at most k operators. We show that it holds for $k+1$ operators. Since k is arbitrary, it holds for all k. The outermost operator (the $k+1$st) is either concatenation, union, or Kleene closure. We argue each case separately.

CASE 1: Let $r = (r_1 \cdot r_2)$. M_1 and M_2 are the NFSMs that accept r_1 and r_2, respectively. By the inductive hypothesis, such machines exist. Without loss of generality, assume that the states of these machines are distinct and let them have initial states s_1 and s_2, respectively. As suggested in Fig. 4.7, create a machine M that accepts r as follows: for each input letter σ, final state f of M_1, and state q of M_2 reached by an edge from s_2 labeled σ, add an edge with the same label σ from f to q. If s_2 is not a final state of M_2, remove the final state designations from states of M_1.

It follows that every string accepted by M either terminates on a final state of M_1 (when M_2 accepts the empty string) or exits a final state of M_1 (never to return to a state of M_1), enters a state of M_2 reachable on one input letter from the initial state of M_2, and terminates on a final state of M_2. Thus, M accepts exactly the strings described by r.

CASE 2: Let $r = (r_1 + r_2)$. Let M_1 and M_2 be NFSMs with distinct sets of states and let initial states s_1 and s_2 accept r_1 and r_2, respectively. By the inductive hypothesis, M_1 and M_2 exist. As suggested in Fig. 4.8, create a machine M that accepts r as follows: a) add a new initial state s_0; b) for each input letter σ and state q of M_1 or M_2 reached by an edge

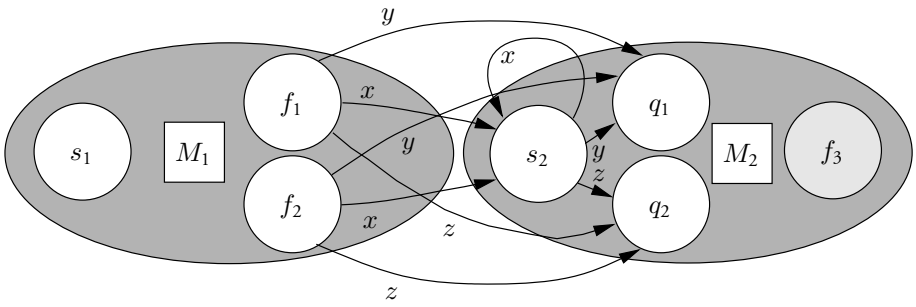


Figure 4.7 A machine M recognizing $r_1 \cdot r_2$. M_1 and M_2 are the NFSMs that accept r_1 and r_2, respectively. An edge with label a is added between each final state of M_1 and each state of M_2 reached on input a from its start state, s_2. The final states of M_2 are final states of M, as are the final states of M_1 if s_2 is a final of M_2. It follows that this machine accepts the strings beginning with a string in r_1 followed by one in r_2.

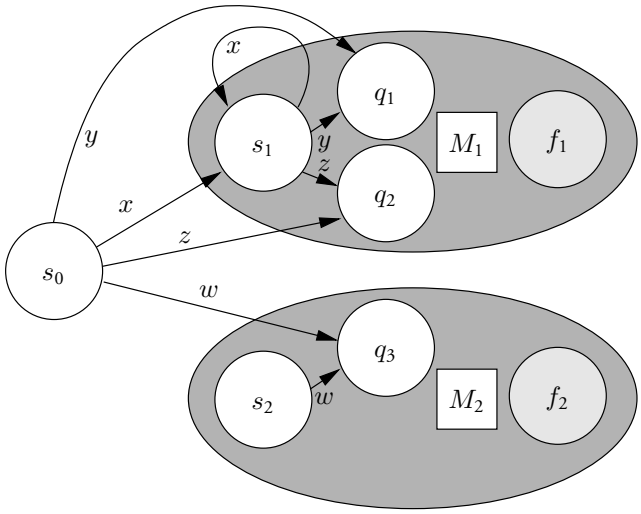


Figure 4.8 A machine M accepting $r_1 + r_2$. M_1 and M_2 are the NFSMs that accept r_1 and r_2, respectively. The new start state s_0 has an edge labeled a for each edge with this label from the initial state of M_1 or M_2. The final states of M are the final states of M_1 and M_2 as well as s_0 if either s_1 or s_2 is a final state. After the first input choice, the new machine acts like either M_1 or M_2. Therefore, it accepts strings denoted by $r_1 + r_2$.

from s_1 or s_2 labeled σ, add an edge with the same label from s_0 to q. If either s_1 or s_2 is a final state, make s_0 a final state.

It follows that if either M_1 or M_2 accepts the empty string, so does M. On the first non-empty input letter M enters and remains in either the states of M_1 or those of M_2. It follows that it accepts either the strings accepted by M_1 or those accepted by M_2 (or both), that is, the union of r_1 and r_2.

CASE 3: Let $r = (r_1)^*$. Let M_1 be an NFSM with initial state s_1 that accepts r_1, which, by the inductive hypothesis, exists. Create a new machine M, as suggested in Fig. 4.9, as follows: a) add a new initial state s_0; b) for each input letter σ and state q reached on σ from s_1, add an edge with label σ between s_0 and state q with label σ, as in Case 2; c) add such edges from each final state to these same states. Make the new initial state a final state and remove the initial-state designation from s_1.

It follows that M accepts the empty string, as it should since $r = (r_1)^*$ contains the empty string. Since the edges leaving each final state are those directed away from the initial state s_0, it follows that M accepts strings that are the concatenation of strings in r_1, as it should. ■

We now illustrate this construction of an NFSM from a regular expression. Consider the regular expression $r = 10^* + 0$, which we decompose as $r = (r_1 r_2 + r_3)$ where $r_1 = 1$, $r_2 = (r_4)^*$, $r_3 = 0$, and $r_4 = 0$. Shown in Fig. 4.10(a) is a NFSM accepting the languages denoted by the regular expressions r_3 and r_4, and in (b) is an NFSM accepting r_1. Figure 4.11 shows an NFSM accepting the closure of r_4 obtained by adding a new initial state (which is also made a final state) from which is directed a copy of the edge directed away from the initial

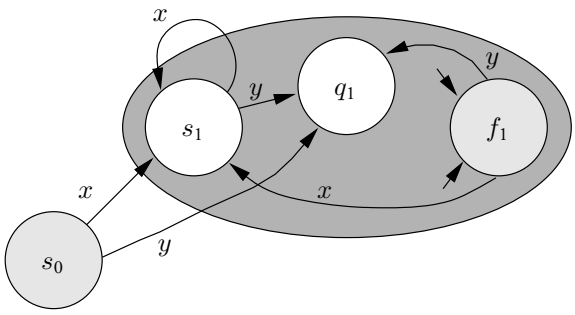


Figure 4.9 A machine M accepts r_1^*. M_1 accepts r_1. Make s_0 the initial state of M. For each input letter a, add an edge labeled a from s_0 and each final of M_1 to each state reached on input a from s_1, the initial state of M_1. The final states of M are s_0 and the final states of M_1. Thus, M accepts ϵ and all states reached by the concatenation of strings accepted by M_1; that is, it realizes the closure r_1^*.

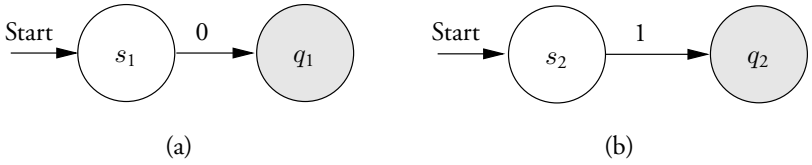


Figure 4.10 Nondeterministic machines accepting 0 and 1.

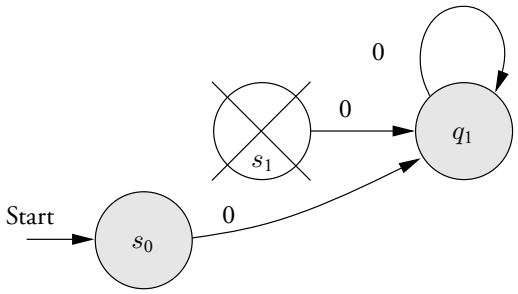


Figure 4.11 An NFSM accepting the Kleene closure of $\{0\}$.

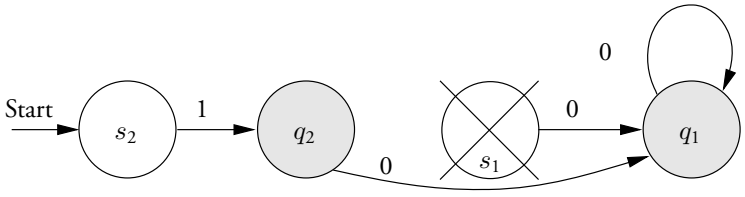


Figure 4.12 A nondeterministic machine accepting 10^*.

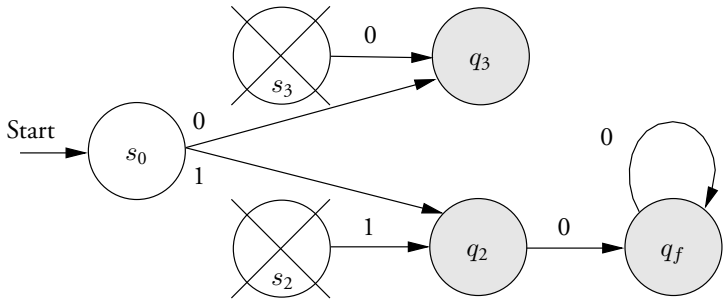


Figure 4.13 A nondeterministic machine accepting $10^* + 0$.

state of M_0, the machine accepting r_4. (The state s_1 is marked as inaccessible.) Figure 4.12 (page 163) shows an NFSM accepting r_1r_2 constructed by **concatenating** the machine M_1 accepting r_1 with M_2 accepting r_2. (s_1 is inaccessible.) Figure 4.13 gives an NFSM accepting the language denoted by $r_1r_2+r_3$, designed by forming the **union** of machines for r_1r_2 and r_3. (States s_2 and s_3 are inaccessible.) Figure 4.14 shows a DFSM recognizing the same language as that accepted by the machine in Fig. 4.13. Here we have added a reject state q_R to which all states move on input letters for which no state transition is defined.

4.4.2 Regular Expressions Describing FSM Languages

We now give the second part of the proof of equivalence of FSMs and regular expressions. We show that every language recognized by a DFSM can be described by a regular expression. We illustrate the proof using the DFSM of Fig. 4.3, which is the DFSM given in Fig. 4.15 except for a relabeling of states.

THEOREM 4.4.2 *If the language L is recognized by a DFSM $M = (\Sigma, Q, \delta, s, F)$, then L can be represented by a regular expression.*

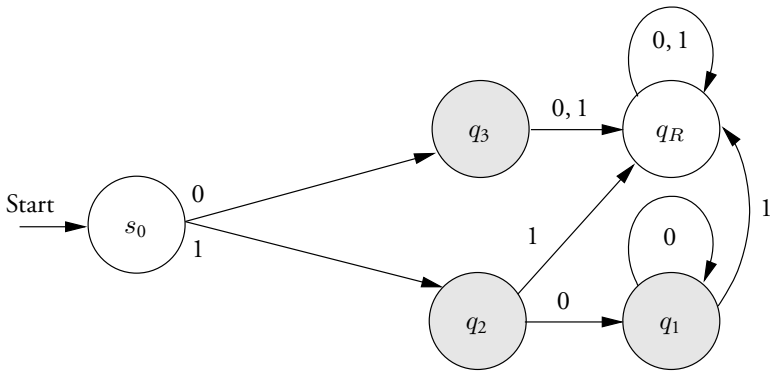


Figure 4.14 A deterministic machine accepting $10^* + 0$.

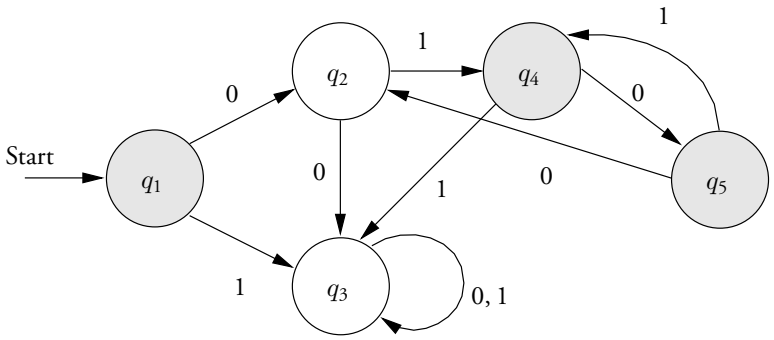


Figure 4.15 The DFSM of Figure 4.3 with a relabeling of states.

Proof Let $Q = \{q_1, q_2, \ldots, q_n\}$ and $F = \{q_{j_1}, q_{j_2}, \ldots, q_{j_p}\}$ be the final states. The proof idea is the following. For every pair of states (q_i, q_j) of M we construct a regular expression $r_{i,j}^{(0)}$ denoting the set $R_{i,j}^{(0)}$ containing input letters that take M from q_i to q_j without passing through any other states. If $i = j$, $R_{i,j}^{(0)}$ contains the empty letter ϵ because M can move from q_i to q_i without reading an input letter. (These definitions are illustrated in the table $T^{(0)}$ of Fig. 4.16.) For $k = 1, 2, \ldots, m$ we proceed to define the set $R_{i,j}^{(k)}$ of strings that take M from q_i to q_j without passing through any state except possibly one in $Q^{(k)} = \{q_1, q_2, \ldots, q_k\}$. We also associate a regular expression $r_{i,j}^{(k)}$ with the set $R_{i,j}^{(k)}$. Since $Q^{(n)} = Q$, the input strings that carry M from $s = q_t$, the initial state, to a final state in F are the strings accepted by M. They can be described by the following regular expression:

$$r_{t,j_1}^{(n)} + r_{t,j_2}^{(n)} + \cdots + r_{t,j_p}^{(n)}$$

This method of proof provides a **dynamic programming** algorithm to construct a regular expression for L.

$T^{(0)} = \{r_{i,j}^{(0)}\}$

$i \setminus j$	1	2	3	4	5
1	ϵ	0	1	$\emptyset$	$\emptyset$
2	$\emptyset$	ϵ	0	1	$\emptyset$
3	$\emptyset$	$\emptyset$	$\epsilon + 0 + 1$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	1	ϵ	0
5	$\emptyset$	0	$\emptyset$	1	ϵ

Figure 4.16 The table $T^{(0)}$ containing the regular expressions $\{r_{i,j}^{(0)}\}$ associated with the DFSM in shown in Fig. 4.15.

$R_{i,j}^{(0)}$ is formally defined below.

$$R_{i,j}^{(0)} = \begin{cases} \{a \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{a \mid \delta(q_i, a) = q_j\} \cup \{\epsilon\} & \text{if } i = j \end{cases}$$

Since $R_{i,j}^{(k)}$ is defined as the set of strings that take M from q_i to q_j without passing through states outside of $Q^{(k)}$, it can be recursively defined as the strings that take M from q_i to q_j without passing through states outside of $Q^{(k-1)}$ plus those that take M from q_i to q_k without passing through states outside of $Q^{(k-1)}$, followed by strings that take M from q_k to q_k zero or more times without passing through states outside $Q^{(k-1)}$, followed by strings that take M from q_k to q_j without passing through states outside of $Q^{(k-1)}$. This is represented by the formula below and suggested in Fig. 4.17:

$$R_{i,j}^{(k)} = R_{i,j}^{(k-1)} \cup R_{i,k}^{(k-1)} \cdot \left(R_{k,k}^{(k-1)}\right)^* \cdot R_{k,j}^{(k-1)}$$

It follows by induction on k that $R_{i,j}^{(k)}$ correctly describes the strings that take M from q_i to q_j without passing through states of index higher than k.

We now exhibit the set $\{r_{i,j}^{(k)}\}$ of regular expressions that describe the sets $\{R_{i,j}^{(k)} \mid 1 \leq i, j, k \leq m\}$ and establish the correspondence by induction. If the set $R_{i,j}^{(0)}$ contains the letters $x_1, x_2, \ldots, x_l$ (which might include the empty letter ϵ), then we let $r_{i,j}^{(0)} = x_1 + x_2 + \cdots + x_l$. Assume that $r_{i,j}^{(k-1)}$ correctly describes $R_{i,j}^{(k-1)}$. It follows that the regular expression

$$r_{i,j}^{(k)} = r_{i,j}^{(k-1)} + r_{i,k}^{(k-1)} \left(r_{k,k}^{(k-1)}\right)^* r_{k,j}^{(k-1)} \tag{4.1}$$

correctly describes $R_{i,j}^{(k)}$. This concludes the proof. ■

The dynamic programming algorithm given in the above proof is illustrated by the DFSM in Fig. 4.15. Because this algorithm can produce complex regular expressions even for small DFSMs, we display almost all of its steps, stopping when it is obvious which results are needed for the regular expression that describes the strings recognized by the DFSM. For $1 \leq k \leq 6$,

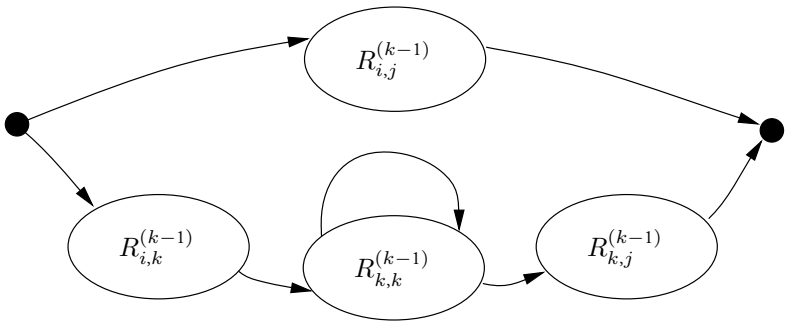


Figure 4.17 A recursive decomposition of the set $R_{i,j}^{(k)}$ of strings that cause an FSM to move from state q_i to q_j without passing through states q_l for $l > k$.

let $T^{(k)}$ denote the table of values of $\{r_{i,j}^{(k)} \mid 1 \leq i,j \leq 6\}$. Table $T^{(0)}$ in Fig. 4.16 describes the next-state function of this DFSM. The remaining tables are constructed by invoking the definition of $r_{i,j}^{(k)}$ in (4.1). Entries in table $T^{(1)}$ are formed using the following facts:

$$r_{i,j}^{(1)} = r_{i,j}^{(0)} + r_{i,1}^{(0)} \left(r_{1,1}^{(0)}\right)^* r_{1,j}^{(0)}; \quad \left(r_{1,1}^{(0)}\right)^* = \epsilon^* = \epsilon; \quad r_{i,1}^{(0)} = \emptyset \quad \text{for } i \geq 2$$

It follows that $r_{i,j}^{(1)} = r_{i,j}^{(0)}$ or that $T^{(1)}$ is identical to $T^{(0)}$. Invoking the identity $r_{i,j}^{(2)} = r_{i,j}^{(1)} + r_{i,2}^{(1)} \left(r_{2,2}^{(1)}\right)^* r_{2,j}^{(1)}$ and using $\left(r_{2,2}^{(1)}\right)^* = \epsilon$, we construct the table $T^{(2)}$ below:

$T^{(2)} = \{r_{i,j}^{(2)}\}$

$i \setminus j$	1	2	3	4	5
1	ϵ	0	$1+00$	01	$\emptyset$
2	$\emptyset$	ϵ	0	1	$\emptyset$
3	$\emptyset$	$\emptyset$	$\epsilon + 0 + 1$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	1	ϵ	0
5	$\emptyset$	0	00	$1+01$	ϵ

The fourth table $T^{(3)}$ is shown below. It is constructed using the identity $r_{i,j}^{(3)} = r_{i,j}^{(2)} + r_{i,3}^{(2)} \left(r_{3,3}^{(2)}\right)^* r_{3,j}^{(2)}$ and the fact that $\left(r_{3,3}^{(2)}\right)^* = (0+1)^*$.

$T^{(3)} = \{r_{i,j}^{(3)}\}$

$i \setminus j$	1	2	3	4	5
1	ϵ	0	$(1+00)(0+1)^*$	01	$\emptyset$
2	$\emptyset$	ϵ	$0(0+1)^*$	1	$\emptyset$
3	$\emptyset$	$\emptyset$	$(0+1)^*$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$1(0+1)^*$	ϵ	0
5	$\emptyset$	0	$00(0+1)^*$	$1+01$	ϵ

The fifth table $T^{(4)}$ is shown below. It is constructed using the identity $r_{i,j}^{(4)} = r_{i,j}^{(3)} + r_{i,4}^{(3)} \left(r_{4,4}^{(3)}\right)^* r_{4,j}^{(3)}$ and the fact that $\left(r_{4,4}^{(3)}\right)^* = \epsilon$.

$T^{(4)} = \{r_{i,j}^{(4)}\}$

$i \setminus j$	1	2	3	4	5
1	ϵ	0	$(1+00+011)(0+1)^*$	01	010
2	$\emptyset$	ϵ	$(0+11)(0+1)^*$	1	10
3	$\emptyset$	$\emptyset$	$(0+1)^*$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$1(0+1)^*$	ϵ	0
5	$\emptyset$	0	$(00+11+011)(0+1)^*$	$1+01$	$\epsilon+10+010$

Instead of building the sixth table, $T^{(5)}$, we observe that the regular expression that is needed is $r = r_{1,1}^{(5)} + r_{1,4}^{(5)} + r_{1,5}^{(5)}$. Since $r_{i,j}^{(5)} = r_{i,j}^{(4)} + r_{i,5}^{(4)} \left(r_{5,5}^{(4)}\right)^* r_{5,j}^{(4)}$ and $\left(r_{5,5}^{(4)}\right)^* = (10 + 010)^*$, we have the following expressions for $r_{1,1}^{(5)}$, $r_{1,4}^{(5)}$, and $r_{1,5}^{(5)}$:

$$
\begin{aligned}
r_{1,1}^{(5)} &= \epsilon \\
r_{1,4}^{(5)} &= 01 + (010)(10 + 010)^*(1 + 01) \\
r_{1,5}^{(5)} &= 010 + (010)(10 + 010)^*(\epsilon + 10 + 010) = (010)(10 + 010)^*
\end{aligned}
$$

Thus, the DFSM recognizes the language denoted by the regular expression $r = \epsilon + 01 + (010)(10+010)^*(\epsilon+1+01)$. It can be shown that this expression denotes the same language as does $\epsilon + 01 + (01)(01 + 001)^*(\epsilon + 0) = (01 + 010)^*$. (See Problem 4.12.)

4.4.3 grep—Searching for Strings in Files

Many operating systems provide a command to find strings in files. For example, the Unix `grep` command prints all lines of a file containing a string specified by a regular expression. `grep` is invoked as follows:

```
grep regular-expression file_name
```

Thus, the command `grep 'o+' file_name` returns each line of the file `file_name` that contains o^+ somewhere in the line. `grep` is typically implemented with a nondeterministic algorithm whose behavior can be understood by considering the construction of the preceding section.

In Section 4.4.1 we describe a procedure to construct NFSMs accepting strings denoted by regular expressions. Each such machine starts in its initial state before processing an input string. Since `grep` finds lines containing a string that starts anywhere in the lines, these NFSMs have to be modified to implement `grep`. The modifications required for this purpose are straightforward and left as an exercise for the reader. (See Problem 4.19.)

4.5 The Pumping Lemma for FSMs

It is not surprising that some languages are not regular. In this section we provide machinery to show this. It is given in the form of the pumping lemma, which demonstrates that if a regular language contains long strings, it must contain an infinite set of strings of a particular form. We show the existence of languages that do not contain strings of this form, thereby demonstrating that they are not regular.

The **pigeonhole principle** is used to prove the pumping lemma. It states that if there are n pigeonholes and $n + 1$ pigeons, each of which occupies a hole, then at least one hole has two pigeons. This principle, whose proof is obvious (see Section 1.3), enjoys a hallowed place in combinatorial mathematics.

The pigeonhole principle is applied as follows. We first note that if a regular language L is infinite, it contains a string $\boldsymbol{w}$ with at least as many letters as there are states in a DFSM M recognizing L. Including the initial state, it follows that M visits at least one more state while processing $\boldsymbol{w}$ than it has different states. Thus, at least one state is visited at least twice. The substring of $\boldsymbol{w}$ that causes M to move from this state back to itself can be repeated zero or

more times to give other strings in the language. We use the notation $\boldsymbol{u}^n$ to mean the string repeated n times and let $\boldsymbol{u}^0 = \epsilon$.

LEMMA 4.5.1 *Let L be a regular language over the alphabet Σ recognized by a DFSM with m states. If $\boldsymbol{w} \in L$ and $|\boldsymbol{w}| \geq m$, then there are strings $\boldsymbol{r}$, $\boldsymbol{s}$, and $\boldsymbol{t}$ with $|\boldsymbol{s}| \geq 1$ and $|\boldsymbol{rs}| \leq m$ such that $\boldsymbol{w} = \boldsymbol{rst}$ and for all integers $n \geq 0$, $\boldsymbol{rs}^n\boldsymbol{t}$ is also in L.*

Proof Let L be recognized by the DFSM M with m states. Let $k = |\boldsymbol{w}| \geq m$ be the length of $\boldsymbol{w}$ in L. Let $q_0, q_1, q_2, \ldots, q_k$ denote the initial and k successive states that M enters after receiving each of the letters in $\boldsymbol{w}$. By the pigeonhole principle, some state q' in the sequence $q_0, \ldots, q_m$ $(m \leq k)$ is repeated. Let $q_i = q_j = q'$ for $i < j$. Let $\boldsymbol{r} = w_1 \ldots w_i$ be the string that takes M from q_0 to $q_i = q'$ (this string may be empty) and let $\boldsymbol{s} = w_{i+1} \ldots w_j$ be the string that takes M from $q_i = q'$ to $q_j = q'$ (this string is non-empty). It follows that $|\boldsymbol{rs}| \leq m$. Finally, let $\boldsymbol{t} = w_{j+1} \ldots w_k$ be the string that takes M from q_j to q_k. Since $\boldsymbol{s}$ takes M from state q' to state q', the final state entered by M is the same whether $\boldsymbol{s}$ is deleted or repeated one or more times. (See Fig. 4.18.) It follows that $\boldsymbol{rs}^n\boldsymbol{t}$ is in L for all $n \geq 0$. ■

As an application of the pumping lemma, consider the language $L = \{0^p1^p \mid p \geq 1\}$. We show that it is not regular. Assume it is regular and is recognized by a DFSM with m states. We show that a contradiction results. Since L is infinite, it contains a string $\boldsymbol{w}$ of length $k = 2p \geq 2m$, that is, with $p \geq m$. By Lemma 4.5.1 L also contains $\boldsymbol{rs}^n\boldsymbol{t}$, $n \geq 0$, where $\boldsymbol{w} = \boldsymbol{rst}$ and $|\boldsymbol{rs}| \leq m \leq p$. That is, $\boldsymbol{s} = 0^d$ where $d \leq p$. Since $\boldsymbol{rs}^n\boldsymbol{t} = 0^{p+(n-1)d}1^p$ for $n \geq 0$ and this is not of the form 0^p1^p for $n = 0$ and $n \geq 2$, the language is not regular.

The pumping lemma allows us to derive specific conditions under which a language is finite or infinite, as we now show.

LEMMA 4.5.2 *Let L be a regular language recognized by a DFSM with m states. L is non-empty if and only if it contains a string of length less than m. It is infinite if and only if it contains a string of length at least m and at most $2m - 1$.*

Proof If L contains a string of length less than m, it is not empty. If it is not empty, let $\boldsymbol{w}$ be a shortest string in L. This string must have length at most $m - 1$ or we can apply the pumping lemma to it and find another string of smaller length that is also in L. But this would contradict the assumption that $\boldsymbol{w}$ is a shortest string in L. Thus, L contains a string of length at most $m - 1$.

If L contains a string $\boldsymbol{w}$ of length $m \leq |\boldsymbol{w}| \leq 2m - 1$, as shown in the proof of the pumping lemma, $\boldsymbol{w}$ can be "pumped up" to produce an infinite set of strings. Suppose now that L is infinite. Either it contains a string $\boldsymbol{w}$ of length $m \leq |\boldsymbol{w}| \leq 2m - 1$ or it does not.

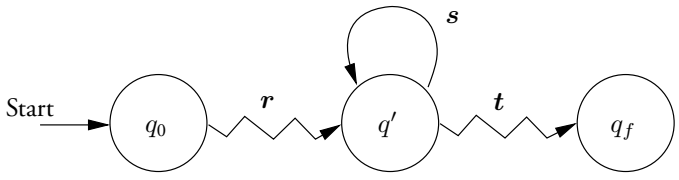


Figure 4.18 Diagram illustrating the pumping lemma.

In the first case, we are done. In the second case, $|w| \geq 2m$ and we apply the pumping lemma to it to find another shorter string that is also in L, contradicting the hypothesis that it was the shortest string of length greater than or equal to $2m$. ■

4.6 Properties of Regular Languages

Section 4.4 established the equivalence of regular languages (recognized by finite-state machines) and the languages denoted by regular expressions. We now present properties satisfied by regular languages. We say that **a class of languages is closed under an operation** if applying that operation to a language (or languages) in the class produces another language in the class. For example, as shown below, the union of two regular languages is another regular language. Similarly, the Kleene closure applied to a regular language returns another regular language.

Given a language L over an alphabet Σ, **the complement of L** is the set $\overline{L} = \Sigma^* - L$, the strings that are in Σ^* but not in L. (This is also called the **difference** between Σ^* and L.) The **intersection** of two languages L_1 and L_2, denoted $L_1 \cap L_2$, is the set of strings that are in both languages.

THEOREM 4.6.1 *The class of regular languages is closed under the following operations:*

- *concatenation*
- *union*
- *Kleene closure*
- *complementation*
- *intersection*

Proof In Section 4.4 we showed that the languages denoted by regular expressions are exactly the languages recognized by finite-state machines (deterministic or nondeterministic). Since regular expressions are defined in terms of concatenation, union, and Kleene closure, they are closed under each of these operations.

The proof of closure of regular languages under complementation is straightforward. If L is regular and has an associated FSM M that recognizes it, make all final states of M nonfinal and all non-final states final. This new machine then recognizes exactly the complement of L. Thus, $\overline{L}$ is also regular.

The proof of closure of regular languages under intersection follows by noting that if L_1 and L_2 are regular languages, then

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

that is, the intersection of two sets can be obtained by complementing the union of their complements. Since each of $\overline{L_1}$ and $\overline{L_2}$ is regular, as is their union, it follows that $\overline{L_1} \cup \overline{L_2}$ is regular. (See Fig. 4.19(a).) Finally, the complement of a regular set is regular. ■

When we come to study Turing machines in Chapter 5, we will show that there are well-defined languages that have no machine to recognize them, even if the machine has an infinite amount of storage available. Thus, it is interesting to ask if there are algorithms that solve certain decision problems about regular languages in a finite number of steps. (Machines that halt on all input are said to **implement algorithms**.) As shown above, there are algorithms

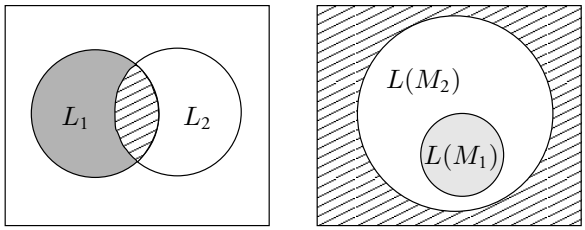


Figure 4.19 (a) The intersection $L_1 \cap L_2$ of two sets L_1 and L_2 can be obtained by taking the complement $\overline{\overline{L_1} \cup \overline{L_2}}$ of the union $\overline{L_1} \cup \overline{L_2}$ of their complements. (b) If $L(M_1) \subseteq L(M_2)$, then $L(M_1) \cap \overline{L(M_2)} = \emptyset$.

that can recognize the concatenation, union and Kleene closure of regular languages. We now show that algorithms exist for a number of decision problems concerning finite-state machines.

THEOREM 4.6.2 *There are algorithms for each of the following decision problems:*

a) For a finite-state machine M and a string $\boldsymbol{w}$, determine if $\boldsymbol{w} \in L(M)$.
b) For a finite-state machine M, determine if $L(M) = \emptyset$.
c) For a finite-state machine M, determine if $L(M) = \Sigma^*$.
d) For finite-state machines M_1 and M_2, determine if $L(M_1) \subseteq L(M_2)$.
e) For finite-state machines M_1 and M_2, determine if $L(M_1) = L(M_2)$.

Proof To answer (a) it suffices to supply $\boldsymbol{w}$ to a deterministic finite-state machine equivalent to M and observe the final state after it has processed all letters in $\boldsymbol{w}$. The number of steps executed by this machine is the length of $\boldsymbol{w}$. Question (b) is answered in Lemma 4.5.2. We need only determine if the language contains strings of length less than m, where m is the number of states of M. This can be done by trying all inputs of length less than m. The answer to question (c) is the same as the answer to "Is $\overline{L(M)} = \emptyset$?" The answer to question (d) is the same as the answer to "Is $L(M_1) \cap \overline{L(M_2)} = \emptyset$?" (See Fig. 4.19(b).) Since FSMs that recognize the complement and intersection of regular languages can be constructed in a finite number of steps (see the proof of Theorem 4.6.1), we can use the procedure for (b) to answer the question. Finally, the answer to question (e) is "yes" if and only if $L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$. ■

4.7 State Minimization*

Given a finite-state machine M, it is often useful to have a potentially different DFSM $M_{\min}$ with the smallest number of states (a minimal-state machine) that recognizes the same language $L(M)$. In this section we develop a procedure to find such a machine recognizing a regular language L. As a step in this direction, we define a natural equivalence relation R_L for each language L and show that L is regular if and only if R_L has a finite number of equivalence classes.

4.7.1 Equivalence Relations on Languages and States

The relation R_L is used to define a machine M_L. When L is regular, we show that M_L is a minimal-state DFSM. We also give an explicit procedure to construct a minimal-state DFSM

recognizing a regular language L. The approach is the following: a) given a regular expression, an NFSM is constructed (Theorem 4.4.1); b) an equivalent DFSM is then produced (Theorem 4.2.1); c) equivalent states of this DFSM are discovered and coalesced, thereby producing the minimal machine. We begin our treatment with a discussion of equivalence relations.

DEFINITION 4.7.1 *An* **equivalence relation** R *on a set* A *is a partition of the elements of* A *into disjoint subsets called* **equivalence classes**. *If two elements* a *and* b *are in the same equivalence class under relation* R, *we write* aRb. *If* a *is an element of an equivalence class, we represent its equivalence class by* $[a]$. *An equivalence relation is represented by its equivalence classes.*

An example of equivalence relation on the set $A = \{0, 1, 2, 3\}$ is the set of equivalence classes $\{\{0, 2\}, \{1, 3\}\}$. Then, $[0]$ and $[2]$ denote the same equivalence class, namely $\{0, 2\}$, whereas $[1]$ and $[2]$ denote different equivalence classes.

Equivalence relations can be defined on any set, including the set of strings over a finite alphabet (a language). For example, let the partition $\{0^*,\ 0(0^*10^*)^+,\ 1(0+1)^*\}$ of the set $(0+1)^*$ denote the equivalence relation R. The equivalence classes consist of strings containing zero or more 0's, strings starting with 0 and containing at least one 1, and strings beginning with 1. It follows that $00R000$ and $1001R11$ but not that $10R01$.

Additional conditions can be put on equivalence relations on languages. An important restriction is that an equivalence relation be right-invariant (with respect to concatenation).

DEFINITION 4.7.2 *An equivalence relation* R *over the alphabet* Σ *is* **right-invariant** *(with respect to concatenation) if for all* $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *in* Σ^*, $\boldsymbol{u}R\boldsymbol{v}$ *implies* $\boldsymbol{uz}R\boldsymbol{vz}$ *for all* $\boldsymbol{z} \in \Sigma^*$.

For example, let $R = \{(10^*1 + 0)^*, 0^*1(10^*1 + 0)^*\}$. That is, R consists of two equivalence classes, the set containing strings with an even number of 1's and the set containing strings with an odd number of 1's. R is right-invariant because if $\boldsymbol{u}R\boldsymbol{v}$; that is, if the numbers of 1's in $\boldsymbol{u}$ and $\boldsymbol{v}$ are both even or both odd, then the same is true of $\boldsymbol{uz}$ and $\boldsymbol{vz}$ for each $\boldsymbol{z} \in \Sigma^*$, that is, $\boldsymbol{uz}R\boldsymbol{vz}$.

To each language L, whether regular or not, we associate the natural equivalence relation R_L defined below. Problem 4.30 shows that for some languages R_L has an unbounded number of equivalence classes.

DEFINITION 4.7.3 *Given a language* L *over* Σ, *the equivalence relation* R_L *is defined as follows: strings* $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$ *are equivalent, that is,* $\boldsymbol{u}R_L\boldsymbol{v}$, *if and only if for each* $\boldsymbol{z} \in \Sigma^*$, *either both* $\boldsymbol{uz}$ *and* $\boldsymbol{vz}$ *are in* L *or both are not in* L.

The equivalence relation $R = \{(10^*1+0)^*, 0^*1(10^*1+0)^*\}$ given above is the equivalence relation R_L for both the language $L = (10^*1 + 0)^*$ and the language $L = 0^*1(10^*1 + 0)^*$.

A natural right-invariant equivalence relation on strings can also be associated with each DFSM, as shown below. This relation defines two strings as equivalent if they carry the machine from its initial state to the same state. Thus, for each state there is an equivalence class of strings that take the machine to that state. For this purpose we extend the state transition function δ to strings $\boldsymbol{a} \in \Sigma^*$ recursively by $\delta(q, \epsilon) = q$ and $\delta(q, \sigma\boldsymbol{a}) = \delta(\delta(q, \sigma), \boldsymbol{a})$ for $\sigma \in \Sigma$.

DEFINITION 4.7.4 *Given a DFSM* $M = (\Sigma, Q, \delta, s, F)$, R_M *is the equivalence relation defined as follows: for all* $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$, $\boldsymbol{u}R_M\boldsymbol{v}$ *if and only if* $\delta(s, \boldsymbol{u}) = \delta(s, \boldsymbol{v})$. *(Note that* $\delta(q, \epsilon) = q$.*)*

It is straightforward to show that the equivalence relations R_L and R_M are right-invariant. (See Problems 4.28 and 4.29.) It is also clear that R_M has as many equivalence classes as there are accessible states of M.

Before we present the major results of this section we define a special machine M_L that will be seen to be a minimal machine recognizing the language L.

DEFINITION 4.7.5 *Given the language L over the alphabet Σ with finite R_L, the DFSM $M_L = (\Sigma, Q_L, \delta_L, s_L, F_L)$ is defined in terms of the right-invariant equivalence relation R_L as follows: a) the states Q_L are the equivalence classes of R_L; b) the initial state s_L is the equivalence class $[\epsilon]$; c) the final states F_L are the equivalence classes containing strings in the language L; d) for an arbitrary equivalence class $[\boldsymbol{u}]$ with representative element $\boldsymbol{u} \in \Sigma^*$ and an arbitrary input letter $a \in \Sigma$, the next-state transition function $\delta_L : Q_L \times \Sigma \mapsto Q_L$ is defined by $\delta_L([\boldsymbol{u}], a) = [\boldsymbol{u}a]$.*

For this definition to make sense we must show that condition c) does not contradict the facts about R_L: that an equivalence class containing a string in L does not also contain a string that is not in L. But by the definition of R_L, if we choose $\boldsymbol{z} = \epsilon$, we have that $\boldsymbol{u} R_L \boldsymbol{v}$ only if both $\boldsymbol{u}$ and $\boldsymbol{v}$ are in L. We must also show that the next-state function definition is consistent: it should not matter which representative of the equivalence class $[\boldsymbol{u}]$ is used. In particular, if we denote the class $[\boldsymbol{u}]$ by $[\boldsymbol{v}]$ for $\boldsymbol{v}$ another member of the class, it should follow that $[\boldsymbol{u}a] = [\boldsymbol{v}a]$. But this is a consequence of the definition of R_L.

Figure 4.20 shows the machine M_L associated with $L = (10^*1 + 0)^*$. The initial state is associated with $[\epsilon]$, which is in the language. Thus, the initial state is also a final state. The state associated with $[0]$ is also $[\epsilon]$ because ϵ and 0 are both in L. Thus, the transition from state $[\epsilon]$ on input 0 is back to state $[\epsilon]$. Problem 4.31 asks the reader to complete the description of this machine.

We need the notion of a refinement of an equivalence relation before we establish conditions for a language to be regular.

DEFINITION 4.7.6 *An equivalence relation R over a set A is a* **refinement** *of an equivalence relation S over the same set if aRb implies that aSb. A refinement R of S is* **strict** *if there exist $a, b \in A$ such that aSb but it is not true that aRb.*

Over the set $A = \{a, b, c, d\}$, the relation $R = \{\{a\}, \{b\}, \{c, d\}\}$ is a strict refinement of the relation $S = \{\{a, b\}, \{c, d\}\}$. Clearly, if R is a refinement of S, R has no fewer equivalence classes than does S. If the refinement R of S is strict, R has more equivalence classes than does S.

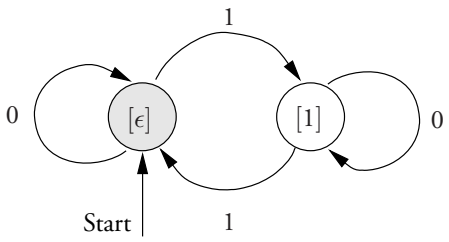


Figure 4.20 The machine M_L associated with $L = (10^*1 + 0)^*$.

4.7.2 The Myhill-Nerode Theorem

The following theorem uses the notion of refinement to give conditions under which a language is regular.

THEOREM 4.7.1 (Myhill-Nerode) *L is a regular language if and only if R_L has a finite number of equivalence classes. Furthermore, if L is regular, it is the union of some of the equivalence classes of R_L.*

Proof We begin by showing that if L is regular, R_L has a finite number of equivalence classes. Let L be recognized by the DFSM $M = (\Sigma, Q, \delta, s, F)$. Then the number of equivalence classes of R_M is finite. Consider two strings $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$ that are equivalent under R_M. By definition, $\boldsymbol{u}$ and $\boldsymbol{v}$ carry M from its initial state to the same state, whether final or not. Thus, $\boldsymbol{uz}$ and $\boldsymbol{vz}$ also carry M to the same state. It follows that R_M is right-invariant. Because $\boldsymbol{u} R_M \boldsymbol{v}$, either $\boldsymbol{u}$ and $\boldsymbol{v}$ take M to a final state and are in L or they take M to a non-final state and are not in L. It follows from the definition of R_L that $\boldsymbol{u} R_L \boldsymbol{v}$. Thus, R_M is a refinement of R_L. Consequently, R_L has no more equivalence classes than does R_M and this number is finite.

Now let R_L have a finite number of equivalence classes. We show that the machine M_L recognizes L. Since it has a finite number of states, we are done. The proof that M_L recognizes L is straightforward. If $[\boldsymbol{w}]$ is a final state, it is reached by applying to M_L in its initial state a string in $[\boldsymbol{w}]$. Since the final states are the equivalence classes containing exactly those strings that are in L, M_L recognizes L. It follows that if L is regular, it is the union of some of the equivalence classes of R_L. ■

We now state an important corollary of this theorem that identifies a minimal machine recognizing a regular language L. Two DFSMs are **isomorphic** if they differ only in the names given to states.

COROLLARY 4.7.1 *If L is regular, the machine M_L is a minimal DFSM recognizing L. All other such minimal machines are isomorphic to M_L.*

Proof From the proof of Theorem 4.7.1, if M is any DFSM recognizing L, it has no fewer states than there are equivalence classes of R_L, which is the number of states of M_L. Thus, M_L has a minimal number of states.

Consider another minimal machine $M_0 = (\Sigma, Q_0, \delta_0, s_0, F_0)$. Each state of M_0 can be identified with some state of M_L. Equate the initial states of M_L and M_0 and let q be an arbitrary state of M_0. There is some string $\boldsymbol{u} \in \Sigma^*$ such that $q = \delta_0(s_0, \boldsymbol{u})$. (If not, M_0 is not minimal.) Equate state q with state $\delta_L(s_L, \boldsymbol{u}) = [\boldsymbol{u}]$ of M_L. Let $\boldsymbol{v} \in [\boldsymbol{u}]$. If $\delta_0(s_0, \boldsymbol{v}) \neq q$, M_0 has more states than does M_L, which is a contradiction. Thus, the identification of states in these two machines is consistent. The final states F_0 of M_0 are identified with those equivalence classes of M_L that contain strings in L.

Consider now the next-state function δ_0 of M_0. Let state q of M_0 be identified with state $[\boldsymbol{u}]$ of M_L and let a be an input letter. Then, if $\delta_0(q, a) = p$, it follows that p is associated with state $[\boldsymbol{u}a]$ of M_L because the input string ua maps s_0 to state p in M_0 and maps s_L to $[\boldsymbol{u}a]$ in M_L. Thus, the next-state functions of the two machines are identical up to a renaming of the states of the two machines. ■

4.7.3 A State Minimization Algorithm

The above approach does not offer a direct way to find a minimal-state machine. In this section we give a procedure for this purpose. Given a regular language, we construct an NFSM that recognizes it (Theorem 4.4.1) and then convert the NFSM to an equivalent DFSM (Theorem 4.2.1). Once we have such a DFSM M, we give a procedure to minimize the number of states based on combining equivalence classes of the right-invariant equivalence relation R_M that are indistinguishable. (These equivalence classes are sets of states of M.) The resulting machine is isomorphic to M_L, the minimal-state machine.

DEFINITION 4.7.7 *Let $M = (\Sigma, Q, \delta, s, F)$ be a DFSM. The equivalence relation $\equiv_n$ on states in Q is defined as follows: two states p and q of M are* **n-indistinguishable** *(denoted $p \equiv_n q$) if and only if for all input strings $\boldsymbol{u} \in \Sigma^*$ of length $|\boldsymbol{u}| \leq n$ either both $\delta(p, \boldsymbol{u})$ and $\delta(q, \boldsymbol{u})$ are in F or both are not in F. (We write $p \not\equiv_n q$ if p and q are not n-indistinguishable.) Two states p and q are* **equivalent** *(denoted $p \equiv q$) if they are n-indistinguishable for all $n \geq 0$.*

For arbitrary states q_1, q_2, and q_3, if q_1 and q_2 are n-indistinguishable and q_2 and q_3 are n-indistinguishable, then q_1 and q_3 are n-indistinguishable. Thus, all three states are in the same set of the partition and $\equiv_n$ is an equivalence relation. By an extension of this type of reasoning to all values of n, it is also clear that $\equiv$ is an equivalence relation.

The following lemma establishes that $\equiv_{j+1}$ refines $\equiv_j$ and that for some k and all $j \geq k$, $\equiv_j$ is identical to $\equiv_k$, which is in turn equal to $\equiv$.

LEMMA 4.7.1 *Let $M = (\Sigma, Q, \delta, s, F)$ be an arbitrary DFSM. Over the set Q the equivalence relation $\equiv_{n+1}$ is a refinement of the relation $\equiv_n$. Furthermore, if for some $k \leq |Q| - 2$, $\equiv_{k+1}$ and $\equiv_k$ are equal, then so are $\equiv_{j+1}$ and $\equiv_j$ for all $j \geq k$. In particular, $\equiv_k$ and $\equiv$ are identical.*

Proof If $p \equiv_{n+1} q$ then $p \equiv_n q$ by definition. Thus, for $n \geq 0$ $\equiv_{n+1}$ refines $\equiv_n$.

We now show that if $\equiv_{k+1}$ and $\equiv_k$ are equal, then $\equiv_{j+1}$ and $\equiv_j$ are equal for all $j \geq$k. Suppose not. Let l be the smallest value of j for which $\equiv_{j+1}$ and $\equiv_j$ are equal but $\equiv_{j+2}$ and $\equiv_{j+1}$ are not equal. It follows that there exist two states p and q that are indistinguishable for input strings of length $l + 1$ or less but are distinguishable for some input string $\boldsymbol{v}$ of length $|\boldsymbol{v}| = l + 2$. Let $\boldsymbol{v} = a\boldsymbol{u}$ where $a \in \Sigma$ and $|\boldsymbol{u}| = l + 1$. Since $\delta(p, \boldsymbol{v}) = \delta(\delta(p, a), \boldsymbol{u})$ and $\delta(q, \boldsymbol{v}) = \delta(\delta(q, a), \boldsymbol{u})$, it follows that the states $\delta(p, a)$ and $\delta(q, a)$ are distinguishable by some string $\boldsymbol{u}$ of length $l + 1$ but not by any string of length l. But this contradicts the assumption that $\equiv_{l+1}$ and $\equiv_l$ are equal.

The relation $\equiv_0$ has two equivalence classes, the final states and all other states. For each integer $j \leq k$, where k is the smallest integer such that $\equiv_{k+1}$ and $\equiv_k$ are equal, $\equiv_j$ has at least one more equivalence class than does $\equiv_{j-1}$. That is, it has at least $j + 2$ classes. Since $\equiv_k$ can have at most $|Q|$ equivalence classes, it follows that $k + 2 \leq |Q|$.

Clearly, $\equiv_k$ and $\equiv$ are identical because if two states cannot be distinguished by input strings of length k or less, they cannot be distinguished by input strings of any length. ■

The proof of this lemma provides an algorithm to compute the equivalence relation $\equiv$, namely, compute the relations $\equiv_j$, $0 \leq j \leq |Q| - 2$ in succession until we find two relations that are identical. We find $\equiv_{j+1}$ from $\equiv_j$ as follows: for every pair of states (p, q) in an equivalence class of $\equiv_j$, we find their successor states $\delta(p, a)$ and $\delta(q, a)$ under input letter a for each such letter. If for all letters a, $\delta(p, a) \equiv_j \delta(q, a)$ and $p \equiv_j q$, then $p \equiv_{j+1} q$ because we cannot distinguish between p and q on inputs of length $j + 1$ or less. Thus, the

algorithm compares each pair of states in an equivalence class of $\equiv_j$ and forms equivalence classes of $\equiv_{j+1}$ by grouping together states whose successors under input letters are in the same equivalence class of $\equiv_j$.

To illustrate these ideas, consider the DFSM of Fig. 4.14. The equivalence classes of $\equiv_0$ are $\{\{s_0, q_R\}, \{q_1, q_2, q_3\}\}$. Since $\delta(s_0, 0)$ and $\delta(q_R, 0)$ are different, s_0 and q_R are in different equivalence classes of $\equiv_1$. Also, because $\delta(q_3, 0) = q_R$ and $\delta(q_1, 0) = \delta(q_2, 0) = q_1 \in F$, q_3 is in a different equivalence class of $\equiv_1$ from q_1 and q_2. The latter two states are in the same equivalence class because $\delta(q_1, 1) = \delta(q_2, 1) = q_R \notin F$. Thus, $\equiv_1 = \{\{s_0\}, \{q_R\}, \{q_3\}, \{q_1, q_2\}\}$. The only one of these equivalence classes that could be refined is the last one. However, since we cannot distinguish between the two states in this class under any input, no further refinement is possible and $\equiv = \equiv_1$.

We now show that if two states are equivalent under $\equiv$, they can be combined, but if they are distinguishable under $\equiv$, they cannot. Applying this procedure provides a minimal-state DFSM.

DEFINITION 4.7.8 *Let $M = (\Sigma, Q, \delta, s, F)$ be a DFSM and let $\equiv$ be the equivalence relation defined above over Q. The DFSM $M_{\equiv} = (\Sigma, Q_{\equiv}, \delta_{\equiv}, [s], F_{\equiv})$ associated with the relation $\equiv$ is defined as follows: a) the states $Q_{\equiv}$ are the equivalence classes of $\equiv$; b) the initial state of $M_{\equiv}$ is $[s]$; c) the final states $F_{\equiv}$ are the equivalence classes containing states in F; d) for an arbitrary equivalence class $[q]$ with representative element $q \in Q$ and an arbitrary input letter $a \in \Sigma$, the next-state function $\delta_{\equiv} : Q_{\equiv} \times \Sigma \mapsto Q_{\equiv}$ is defined by $\delta_{\equiv}([q], a) = [\delta(q, a)]$.*

This definition is consistent; no matter which representative of the equivalence class $[q]$ is used, the next state on input a is $[\delta(q, a)]$. It is straightforward to show that $M_{\equiv}$ recognizes the same language as does M. (See Problem 4.27.) We now show that $M_{\equiv}$ is a minimal-state machine.

THEOREM 4.7.2 *$M_{\equiv}$ is a minimal-state machine.*

Proof Let $M = (\Sigma, Q, \delta, s, F)$ be a DFSM recognizing L and let $M_{\equiv}$ be the DFSM associated with the equivalence relation $\equiv$ on Q. Without loss of generality, we assume that all states of $M_{\equiv}$ are accessible from the initial state. We now show that $M_{\equiv}$ has no more states than M_L. Suppose it has more states. That is, suppose $M_{\equiv}$ has more states than there are equivalence classes of R_L. Then, there must be two states p and q of M such that $[p] \neq [q]$ but that $\boldsymbol{u} R_L \boldsymbol{v}$, where $\boldsymbol{u}$ and $\boldsymbol{v}$ carry M from its initial state to p and q, respectively. (If this were not the case, any strings equivalent under R_L would carry M from its initial state s to equivalent states, contradicting the assumption that $M_{\equiv}$ has more states than M_L.) But if $\boldsymbol{u} R_L \boldsymbol{v}$, then since R_L is right-invariant, $\boldsymbol{uw} R_L \boldsymbol{vw}$ for all $\boldsymbol{w} \in \Sigma^*$. However, because $[p] \neq [q]$, there is some $\boldsymbol{z} \in \Sigma^*$ such that $[p]$ and $[q]$ can be distinguished. This is equivalent to saying that $\boldsymbol{uz} R_L \boldsymbol{vz}$ does not hold, a contradiction. Thus, $M_{\equiv}$ and M_L have the same number of states. Since $M_{\equiv}$ recognizes L, it is a minimal-state machine equivalent to M. ■

As shown above, the equivalence relation $\equiv$ for the DFSM of Fig. 4.14 is $\equiv$ is $\{\{s_0\}, \{q_R\}, \{q_3\}, \{q_1, q_2\}\}$. The DFSM associated with this relation, $M_{\equiv}$, is shown in Fig. 4.21. It clearly recognizes the language $10^* + 0$. It follows that the equivalent DFSM of Fig. 4.14 is not minimal.

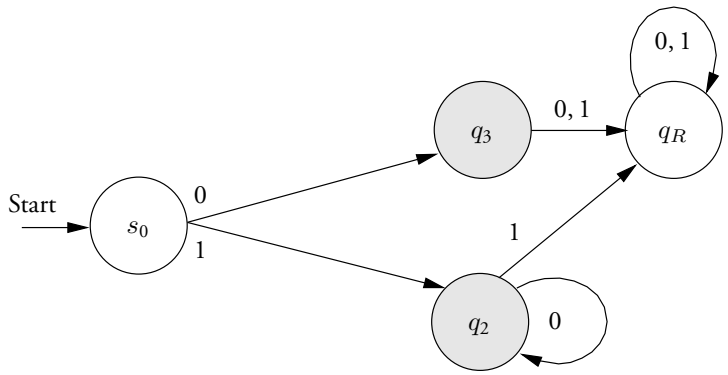


Figure 4.21 A minimal-state DFSM equivalent to the DFSM in Fig. 4.14.

4.8 Pushdown Automata

The pushdown automaton (PDA) has a one-way, read-only, potentially infinite input tape on which an input string is written (see Fig. 4.22); its head either advances to the right from the leftmost cell or remains stationary. It also has a stack, a storage medium analogous to the stack of trays in a cafeteria. The **stack** is a potentially infinite ordered collection of initially blank cells with the property that data can be pushed onto it or popped from it. Data is **pushed** onto the top of the stack by moving all existing entries down one cell and inserting the new element in the top location. Data is **popped** by removing the top element and moving all other entries up one cell. The control unit of a pushdown automaton is a finite-state machine. The full power of the PDA is realized only when its control unit is nondeterministic.

DEFINITION 4.8.1 *A* **pushdown automaton (PDA)** *is a six-tuple* $M = (\Sigma, \Gamma, Q, \Delta, s, F)$, *where* Σ *is the* **tape alphabet** *containing the blank symbol* β, Γ *is the* **stack alphabet** *containing the blank symbol* γ, Q *is the finite* **set of states**, $\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q \times (\Gamma \cup \{\epsilon\}))$ *is the set of* **transitions**, s *is the* **initial state**, *and* F *is the* **set of final states**. *We now describe transitions.*

If for state p, *tape symbol* x, *and stack symbol* y *the transition* $(p, x, y; q, z) \in \Delta$, *then if* M *is in state* p, $x \in \Sigma$ *is under its tape head, and* $y \in \Gamma$ *is at the top of its stack,* M *may pop* y *from its stack, enter state* $q \in Q$, *and push* $z \in \Gamma$ *onto its stack. However, if* $x = \epsilon$, $y = \epsilon$ *or* $z = \epsilon$, *then* M *does not read its tape, pop its stack or push onto its stack, respectively. The head on the tape either remains stationary if* $x = \epsilon$ *or advances one cell to the right if* $x \neq \epsilon$.

If at each point in time a unique transition $(p, x, y; q, z)$ *may be applied, the PDA is* **deterministic**. *Otherwise it is* **nondeterministic**.

The PDA M **accepts the input string** $w \in \Sigma^*$ *if when started in state* s *with an empty stack (its cells contain the* **blank stack symbol** γ*) and* w *placed left-adjusted on its otherwise blank tape (its blank cells contain the* **blank tape symbol** β*), the last state entered by* M *after reading the components of* w *and no other tape cells is a member of the set* F. M **accepts the language** $L(M)$ *consisting of all such strings.*

Some of the special cases for the action of the PDA M on empty tape or stack symbols are the following: if $(p, x, \epsilon; q, z)$, x is read, state q is entered, and z is pushed onto

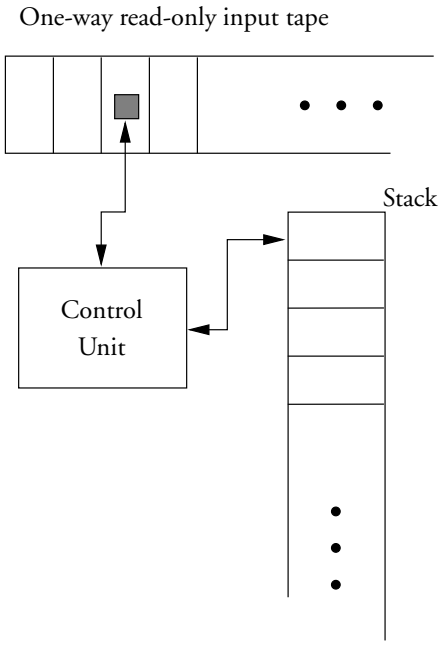


Figure 4.22 The control unit, one-way input tape, and stack of a pushdown automaton.

the stack; if $(p, x, y; q, \epsilon)$, x is read, state q is entered, and y is popped from the stack; if $(p, \epsilon, y; q, z)$, no input is read, y is popped, z is pushed and state q is entered. Also, if $(p, \epsilon, \epsilon; q, \epsilon)$, M moves from state p to q without reading input, or pushing or popping the stack.

Observe that if every transition is of the form $(p, x, \epsilon; q, \epsilon)$, the PDA ignores the stack and simulates an FSM. Thus, the languages accepted by PDAs include the regular languages.

We emphasize that a PDA is nondeterministic if for some state q, tape symbol x, and top stack item y there is more than one transition that M can make. For example, if Δ contains $(s, a, \epsilon; s, a)$ and $(s, a, a; r, \epsilon)$, M has the choice of ignoring or popping the top of the stack and of moving to state s or r. If after reading all symbols of $\boldsymbol{w}$ M enters a state in F, then M accepts $\boldsymbol{w}$.

We now give two examples of PDAs and the languages they accept. The first accepts palindromes of the form $\{\boldsymbol{w}c\boldsymbol{w}^R\}$, where $\boldsymbol{w}^R$ is the reverse of $\boldsymbol{w}$ and $\boldsymbol{w} \in \{a, b\}^*$. The state diagram of its control unit is shown in Fig. 4.23. The second PDA accepts those strings over $\{a, b\}$ of the form $a^n b^m$ for which $n \geq m$.

EXAMPLE 4.8.1 *The PDA* $M = (\Sigma, \Gamma, Q, \Delta, s, F)$, *where* $\Sigma = \{a, b, c, \beta\}$, $\Gamma = \{a, b, \gamma\}$, $Q = \{s, p, r, f\}$, $F = \{f\}$ *and* Δ *contains the transitions shown in Fig. 4.24, accepts the language* $L = \{\boldsymbol{w}c\boldsymbol{w}^R\}$.

The PDA M of Figs. 4.23 and 4.24 remains in the **stacking state** s while encountering a's and b's on the input tape, pushing these letters (the order of these letters on the stack is the reverse of their order on the input tape) onto the stack (Rules (a) and (b)). If it encounters an

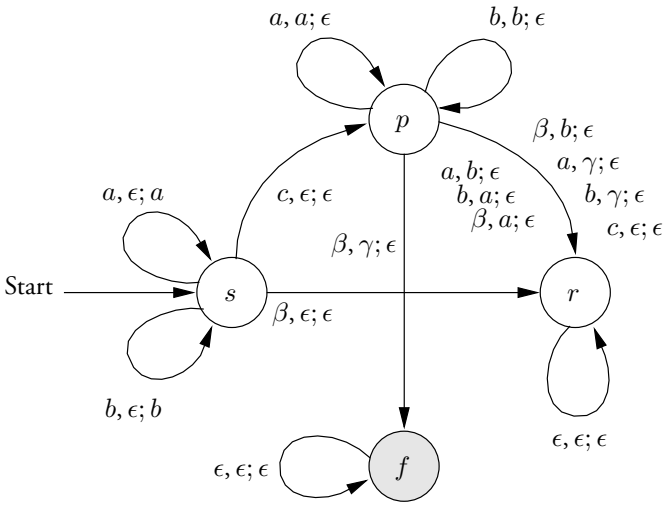


Figure 4.23 State diagram for the pushdown automaton of Fig. 4.24 which accepts $\{\boldsymbol{w}c\boldsymbol{w}^R\}$. An edge label $a, b; c$ between states p and q corresponds to the transition $(p, a, b; q, c)$.

instance of letter c while in state s, it enters the **possible accept state** p (Rule (c)) but enters the **reject state** r if it encounters a blank on the input tape (Rule (d)). While in state p it pops an a or b that matches the same letter on the input tape (Rules (e) and (f)). If the PDA discovers blank tape and stack symbols, it has identified a palindrome and enters the **accept state** f (Rule (g)). On the other hand, if while in state p the tape symbol and the symbol on the top of the stack are different or the letter c is encountered, the PDA enters the reject state r (Rules (h)–(n)). Finally, the PDA does not exit from either the reject or accept states (Rules (o) and (p)).

	Rule	*Comment*		*Rule*	*Comment*
(a)	$(s, a, \epsilon; s, a)$	push a	(i)	$(p, b, a; r, \epsilon)$	reject
(b)	$(s, b, \epsilon; s, b)$	push b	(j)	$(p, \beta, a; r, \epsilon)$	reject
(c)	$(s, c, \epsilon; p, \epsilon)$	accept?	(k)	$(p, \beta, b; r, \epsilon)$	reject
(d)	$(s, \beta, \epsilon; r, \epsilon)$	reject	(l)	$(p, a, \gamma; r, \epsilon)$	reject
(e)	$(p, a, a; p, \epsilon)$	accept?	(m)	$(p, b, \gamma; r, \epsilon)$	reject
(f)	$(p, b, b; p, \epsilon)$	accept?	(n)	$(p, c, \epsilon; r, \epsilon)$	reject
(g)	$(p, \beta, \gamma; f, \epsilon)$	accept	(o)	$(r, \epsilon, \epsilon; r, \epsilon)$	stay in reject state
(h)	$(p, a, b; r, \epsilon)$	reject	(p)	$(f, \epsilon, \epsilon; f, \epsilon)$	stay in accept state

Figure 4.24 Transitions for the PDA described by the state diagram of Fig. 4.23.

	Rule	*Comment*
(a)	$(s, \beta, \epsilon; f, \epsilon)$	accept
(b)	$(s, a, \epsilon; s, a)$	push a
(c)	$(s, b, \gamma; r, \epsilon)$	reject
(d)	$(s, b, a; p, \epsilon)$	pop a, enter pop state
(e)	$(p, b, a; p, \epsilon)$	pop a
(f)	$(p, b, \gamma; r, \epsilon)$	reject

	Rule	*Comment*
(g)	$(p, \beta, a; f, \epsilon)$	accept
(h)	$(p, \beta, \gamma; f, \epsilon)$	accept
(i)	$(p, a, \epsilon; r, \epsilon)$	reject
(j)	$(f, \epsilon, \epsilon; f, \epsilon)$	stay in accept state
(k)	$(r, \epsilon, \epsilon; r, \epsilon)$	stay in reject state

Figure 4.25 Transitions for a PDA that accepts the language $\{a^n b^m \mid n \geq m \geq 0\}$.

EXAMPLE 4.8.2 *The PDA $M = (\Sigma, \Gamma, Q, \Delta, s, F)$, where $\Sigma = \{a, b, \beta\}$, $\Gamma = \{a, b, \gamma\}$, $Q = \{s, p, r, f\}$, $F = \{f\}$ and Δ contains the transitions shown in Fig. 4.25, accepts the language $L = \{a^n b^m \mid n \geq m \geq 0\}$. The state diagram for this machine is shown in Fig. 4.26.*

The rules of Fig. 4.25 work as follows. An empty input in the **stacking state** s is accepted (Rule (a)). If a string of a's is found, the PDA remains in state s and the a's are pushed onto the stack (Rule (b)). At the first discovery of a b in the input while in state s, if the stack is empty, the input is rejected by entering the **reject state** (Rule (c)). If the stack is not empty, the a at the top is popped and the PDA enters the **pop state** p (Rule (d)). If while in p a b is discovered on the input tape when an a is found at the top of the stack (Rule(e)), the PDA pops the a and stays in this state because it remains possible that the input contains no more b's than a's. On the other hand, if the stack is empty when a b is discovered, the PDA enters the reject state (Rule (f)). If in state p the PDA discovers that it has more a's than b's by reading

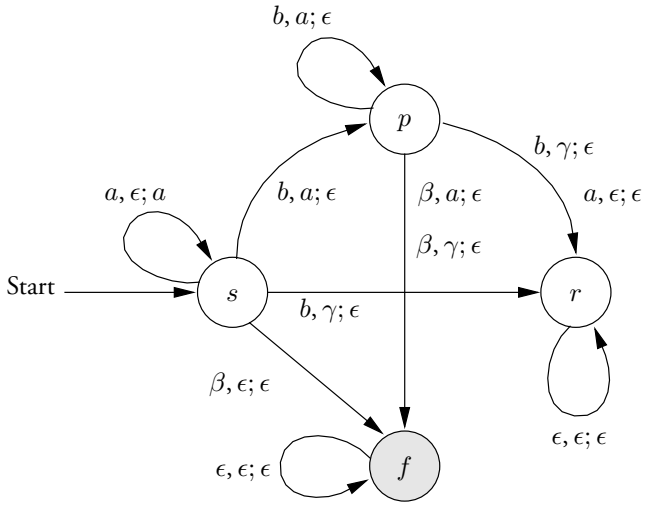


Figure 4.26 The state diagram for the PDA defined by the tables in Fig. 4.25.

the blank tape letter β when the stack is not empty, it enters the **accept state** f (Rule (g)). If the PDA encounters an a on its input tape when in state p, an a has been received after a b and the input is rejected (Rule (i)). After the PDA enters either the accept or reject states, it remains there (Rules (j) and (k)).

In Section 4.12 we show that the languages recognized by pushdown automata are exactly the languages defined by the context-free languages described in the next section.

4.9 Formal Languages

Languages are introduced in Section 1.2.3. A **language** is a set of strings over a finite set Σ, with $|\Sigma| \geq 2$, called an **alphabet**. Σ^* is the language of all strings over Σ including the **empty string** ϵ, which has zero length. The empty string has the property that for an arbitrary string $\boldsymbol{w}$, $\epsilon\boldsymbol{w} = \boldsymbol{w} = \boldsymbol{w}\epsilon$. Σ^+ is the set Σ^* without the empty string.

In this section we introduce grammars for languages, rules for **rewriting strings** through the substitution of substrings. A **grammar** consists of alphabets $\mathcal{T}$ and $\mathcal{N}$ of terminal and non-terminal symbols, respectively, a designated non-terminal start symbol, plus a set of rules $\mathcal{R}$ for rewriting strings. Below we define four types of language in terms of their grammars: the phrase-structure, context-sensitive, context-free, and regular grammars.

The role of grammars is best illustrated with an example for a small fragment of English. Consider a grammar G whose non-terminals $\mathcal{N}$ contain a start symbol S denoting a generic sentence and NP and VP denoting generic noun and verb phrases, respectively. In turn, assume that $\mathcal{N}$ also contains non-terminals for adjectives and adverbs, namely AJ and AV. Thus, $\mathcal{N} = \{\text{S, NP, VP, AJ, AV, N, V}\}$. We allow the grammar to have the following words as terminals: $\mathcal{T} = \{bob, alice, duck, big, smiles, quacks, loudly\}$. Here bob, $alice$, and $duck$ are nouns, big is an adjective, $smiles$ and $quacks$ are verbs, and $loudly$ is an adverb. In our fragment of English a sentence consists of a noun phrase followed by a verb phrase, which we denote by the rule S $\rightarrow$ NP VP. This and the other rules $\mathcal{R}$ of the grammar are shown below. They include rules to map non-terminals to terminals, such as N $\rightarrow$ bob

S	$\rightarrow$	NP VP	N	$\rightarrow$	bob	V	$\rightarrow$	$smiles$
NP	$\rightarrow$	N	N	$\rightarrow$	$alice$	V	$\rightarrow$	$quacks$
NP	$\rightarrow$	AJ N	N	$\rightarrow$	$duck$	AV	$\rightarrow$	$loudly$
VP	$\rightarrow$	V	AJ	$\rightarrow$	big			
VP	$\rightarrow$	V AV						

With these rules the following strings (sentences) can be generated: $bob\ smiles$; $big\ duck\ quacks\ loudly$; and $alice\ quacks$. The first two sentences are acceptable English sentences, but the third is not if we interpret $alice$ as a person. This example illustrates the need for rules that limit the rewriting of non-terminals to an appropriate context of surrounding symbols.

Grammars for formal languages generalize these ideas. Grammars are used to interpret programming languages. A language is translated and given meaning through a series of steps the first of which is **lexical analysis**. In lexical analysis symbols such as a, l, i, c, e are grouped into tokens such as $alice$, or some other string denoting $alice$. This task is typically done with a finite-state machine. The second step in translation is **parsing**, a process in which a tokenized string is associated with a series of **derivations** or applications of the rules of a grammar. For example, $big\ duck\ quacks\ loudly$, can be produced by the following sequence of derivations: S $\rightarrow$ NP VP; NP $\rightarrow$ AJ N; AJ $\rightarrow$ big; N $\rightarrow$ $duck$; VP $\rightarrow$ V AV; V $\rightarrow$ $quacks$; AV $\rightarrow$ $loudly$.

In his exploration of models for natural language, Noam Chomsky introduced four language types of decreasing expressibility, now called the **Chomsky hierarchy**, in which each language is described by the type of grammar generating it. These languages serve as a basis for the classification of programming languages. The four types are the phrase-structure languages, the context-sensitive languages, the context-free languages, and the regular languages.

There is an exact correspondence between each of these types of languages and particular machine architectures in the sense that for each language type T there is a machine architecture A recognizing languages of type T and for each architecture A there is a type T such that all languages recognized by A are of type T. The correspondence between language and architecture is shown in the following table, which also lists the section or problem where the result is established. Here the **linear bounded automaton** is a Turing machine in which the number of tape cells that are used is linear in the length of the input string.

Level	*Language Type*	*Machine Type*	*Proof Location*
0	phrase-structure	Turing machine	Section 5.4
1	context-sensitive	linear bounded automaton	Problem 4.36
2	context-free	nondet. pushdown automaton	Section 4.12
3	regular	finite-state machine	Section 4.10

We now give formal definitions of each of the grammar types under consideration.

4.9.1 Phrase-Structure Languages

In Section 5.4 we show that the phrase-structure grammars defined below are exactly the languages that can be recognized by Turing machines.

DEFINITION 4.9.1 *A* **phrase-structure grammar** G *is a four-tuple* $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ *where* $\mathcal{N}$ *and* $\mathcal{T}$ *are disjoint alphabets of* **non-terminals** *and* **terminals**, *respectively. Let* $V = \mathcal{N} \cup \mathcal{T}$. *The* **rules** $\mathcal{R}$ *form a finite subset of* $V^+ \times V^*$ *(denoted* $\mathcal{R} \subseteq V^+ \times V^*$*) where for every rule* $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}$, $\boldsymbol{a}$ *contains at least one non-terminal symbol. The symbol* $\text{S} \in \mathcal{N}$ *is the* **start symbol**.

If $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}$ *we write* $\boldsymbol{a} \to \boldsymbol{b}$. *If* $\boldsymbol{u} \in V^+$ *and* $\boldsymbol{a}$ *is a contiguous substring of* $\boldsymbol{u}$, *then* $\boldsymbol{u}$ *can be replaced by the string* $\boldsymbol{v}$ *by substituting* $\boldsymbol{b}$ *for* $\boldsymbol{a}$. *If this holds, we write* $\boldsymbol{u} \Rightarrow_G \boldsymbol{v}$ *and call it an* **immediate derivation**. *Extending this notation, if through a sequence of immediate derivations (called a* **derivation***)* $\boldsymbol{u} \Rightarrow_G \boldsymbol{x_1}, \boldsymbol{x_1} \Rightarrow_G \boldsymbol{x_2}, \cdots, \boldsymbol{x_n} \Rightarrow_G \boldsymbol{v}$ *we can transform* $\boldsymbol{u}$ *to* $\boldsymbol{v}$, *we write* $\boldsymbol{u} \overset{*}{\Rightarrow}_G \boldsymbol{v}$ *and say that* $\boldsymbol{v}$ **derives** *from* $\boldsymbol{u}$. *If the rules* $\mathcal{R}$ *contain* $(\boldsymbol{a}, \boldsymbol{a})$ *for all* $\boldsymbol{a} \in \mathcal{N}^+$, *the relation* $\overset{*}{\Rightarrow}_G$ *is called the* **transitive closure** *of the relation* $\Rightarrow_G$ *and* $\boldsymbol{u} \overset{*}{\Rightarrow}_G \boldsymbol{u}$ *for all* $\boldsymbol{u} \in V^*$ *containing at least one non-terminal symbol.*

The **language** $L(G)$ *defined by the grammar* G *is the set of all terminal strings that can be derived from the start symbol* S*; that is,*

$$L(G) = \{\boldsymbol{u} \in \mathcal{T}^* \mid \text{S} \overset{*}{\Rightarrow}_G \boldsymbol{u}\}$$

When the context is clear we drop the subscript G in $\Rightarrow_G$ and $\overset{*}{\Rightarrow}_G$. These definitions are best understood from an example. In all our examples we use letters in SMALL CAPS to denote non-terminals and letters in *italics* to denote terminals, except that ϵ, the empty letter, may also be a terminal.

EXAMPLE 4.9.1 *Consider the grammar* $G_1 = (\mathcal{N}_1, \mathcal{T}_1, \mathcal{R}_1, \text{S})$, *where* $\mathcal{N}_1 = \{\text{S}, \text{B}, \text{C}\}$, $\mathcal{T}_1 = \{a, b, c\}$ *and* $\mathcal{R}_1$ *consists of the following rules:*

$a)$	S	$\rightarrow$	aSBC	$d)$	aB	$\rightarrow$	ab	$g)$	cC	$\rightarrow$	cc
$b)$	S	$\rightarrow$	aBC	$e)$	bB	$\rightarrow$	bb				
$c)$	CB	$\rightarrow$	BC	$f)$	bC	$\rightarrow$	bc				

Clearly the string aaBCBC can be rewritten as aaBBCC using rule (c), that is, aaBCBC $\Rightarrow$ aaBBCC. One application of (d), one of (e), one of (f), and one of (g) reduces it to the string $aabbcc$. Since one application of (a) and one of (b) produces the string aaBBCC, it follows that the language $L(G_1)$ contains $aabbcc$.

Similarly, two applications of (a) and one of (b) produce aaaBCBCBC, after which three applications of (c) produce the string aaaBBBCCC. One application of (d) and two of (e) produce $aaabbb$CCC, after which one application of (f) and two of (g) produces $aaabbbccc$. In general, one can show that $L(G_1) = \{a^n b^n c^n \mid n \geq 1\}$. (See Problem 4.38.)

4.9.2 Context-Sensitive Languages

The context-sensitive languages are exactly the languages accepted by linear bounded automata, nondeterministic Turing machines whose tape heads visit a number of cells that is a constant multiple of the length of an input string. (See Problem 4.36.)

DEFINITION 4.9.2 *A* **context-sensitive grammar** G *is a phrase structure grammar* $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ *in which each rule* $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}$ *satisfies the condition that* $\boldsymbol{b}$ *has no fewer characters than does* $\boldsymbol{a}$, *namely,* $|\boldsymbol{a}| \leq |\boldsymbol{b}|$. *The languages defined by context-sensitive grammars are called* **context-sensitive languages** *(CSL).*

Each rule of a context-sensitive grammar maps a string to one that is no shorter. Since the left-hand side of a rule may have more than one character, it may make replacements based on the context in which a non-terminal is found. Examples of context-sensitive languages are given in Problems 4.38 and 4.39.

4.9.3 Context-Free Languages

As shown in Section 4.12, the context-free languages are exactly the languages accepted by pushdown automata.

DEFINITION 4.9.3 *A* **context-free grammar** $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ *is a phrase structure grammar in which each rule in* $\mathcal{R} \subseteq \mathcal{N} \times V^*$ *has a single non-terminal on the left-hand side. The languages defined by context-free grammars are called* **context-free languages** *(CFL).*

Each rule of a context-free grammar maps a non-terminal to a string over V^* without regard to the context in which the non-terminal is found because the left-hand side of each rule consists of a single non-terminal.

EXAMPLE 4.9.2 *Let* $\mathcal{N}_2 = \{\text{S}, \text{A}\}$, $\mathcal{T}_2 = \{\epsilon, a, b\}$, *and* $\mathcal{R}_2 = \{\text{S} \rightarrow a\text{S}b, \text{S} \rightarrow \epsilon\}$. *Then the grammar* $G_2 = (\mathcal{N}_2, \mathcal{T}_2, \mathcal{R}_2, \text{S})$ *is context-free and generates the language* $L(G_2) = \{a^n b^n \mid n \geq 0\}$. *To see this, let the rule* $\text{S} \rightarrow a\text{S}b$ *be applied* k *times to produce the string* $a^k \text{S} b^k$. *A final application of the last rule establishes the result.*

EXAMPLE 4.9.3 *Consider the grammar G_3 with the following rules and the implied terminal and non-terminal alphabets:*

$a)$ S $\rightarrow$ cMcNc $\quad$ $d)$ N $\rightarrow$ bNb

$b)$ M $\rightarrow$ aMa $\quad$ $e)$ N $\rightarrow$ c

$c)$ M $\rightarrow$ c

G_3 is context-free and generates the language $L(G_3) = \{ca^n ca^n cb^m cb^m c \,|\, n, m \geq 0\}$, as is easily shown.

Context-free languages capture important aspects of many programming languages. As a consequence, the parsing of context-free languages is an important step in the parsing of programming languages. This topic is discussed in Section 4.11.

4.9.4 Regular Languages

DEFINITION 4.9.4 *A* **regular grammar** *G is a context-free grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$, where the right-hand side is either a terminal or a terminal followed by a non-terminal. That is, its rules are of the form* A $\rightarrow$ a *or* A $\rightarrow$ bC*. The languages defined by regular grammars are called* **regular languages**.

Some authors define a regular grammar to be one whose rules are of the form A $\rightarrow$ a or A $\rightarrow$ $b_1 b_2 \cdots b_k$C. It is straightforward to show that any language generated by such a grammar can be generated by a grammar of the type defined above.

The following grammar is regular.

EXAMPLE 4.9.4 *Consider the grammar $G_4 = (\mathcal{N}_4, \mathcal{T}_4, \mathcal{R}_4, \text{S})$ where $\mathcal{N}_4 = \{\text{S}, \text{A}, \text{B}\}$, $\mathcal{T}_4 = \{0,1\}$ and $\mathcal{R}_4$ consists of the rules given below.*

$a)$ S $\rightarrow$ 0A $\quad$ $d)$ B $\rightarrow$ 0A

$b)$ S $\rightarrow$ 0 $\quad$ $e)$ B $\rightarrow$ 0

$c)$ A $\rightarrow$ 1B

It is straightforward to see that the rules a) S $\rightarrow$ 0, b) S $\rightarrow$ 01B, c) B $\rightarrow$ 0, and d) B $\rightarrow$ 01B generate the same strings as the rules given above. Thus, the language G_4 contains the strings $0, 010, 01010, 0101010, \ldots$, that is, strings of the form $(01)^k 0$ for $k \geq 0$. Consequently $L(G_4) = (01)^* 0$. A formal proof of this result is left to the reader. (See Problem 4.44.)

4.10 Regular Language Recognition

As explained in Section 4.1, a deterministic finite-state machine (DFSM) M is a five-tuple $M = (\Sigma, Q, \delta, s, F)$, where Σ is the input alphabet, Q is the set of states, $\delta : Q \times \Sigma \mapsto Q$ is the next-state function, s is the initial state, and F is the set of final states. A nondeterministic FSM (NFSM) is similarly defined except that δ is a next-set function $\delta : Q \times \Sigma \mapsto 2^Q$. In other words, in an NFSM there may be more than one next state for a given state and input. In Section 4.2 we showed that the languages recognized by these two machine types are the same.

We now show that the languages $L(G)$ and $L(G) \cup \{\epsilon\}$ defined by regular grammars G are exactly those recognized by FSMs.

THEOREM 4.10.1 *The languages $L(G)$ and $L(G) \cup \{\epsilon\}$ generated by regular grammars G and recognized by finite-state machines are the same.*

Proof Given a regular grammar G, we construct a corresponding NFSM M that accepts exactly the strings generated by G. Similarly, given a DFSM M we construct a regular grammar G that generates the strings recognized by M.

From a regular grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ with rules $\mathcal{R}$ of the form $\text{A} \rightarrow a$ and $\text{A} \rightarrow b\text{C}$ we create a grammar G' generating the same language by replacing a rule $\text{A} \rightarrow a$ with rules $\text{A} \rightarrow a\text{B}$ and $\text{B} \rightarrow \epsilon$ where B is a new non-terminal unique to $\text{A} \rightarrow a$. Thus, every derivation $\text{S} \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$, $\boldsymbol{w} \in \mathcal{T}^*$, now corresponds to a derivation $\text{S} \stackrel{*}{\Rightarrow}_{G'} \boldsymbol{w}\text{B}$ where $\text{B} \rightarrow \epsilon$. Hence, the strings generated by G and G' are the same.

Now construct an NFSM $M_{G'}$ whose states correspond to the non-terminals of this new regular grammar and whose input alphabet is its set of terminals. Let the start state of $M_{G'}$ be labeled S. Let there be a transition from state A to state B on input a if there is a rule $\text{A} \rightarrow a\text{B}$ in G'. Let a state B be a final state if there is a rule of the form $\text{B} \rightarrow \epsilon$ in G'. Clearly, every derivation of a string $\boldsymbol{w}$ in $L(G')$ corresponds to a path in M that begins in the start state and ends on a final state. Hence, $\boldsymbol{w}$ is accepted by $M_{G'}$. On the other hand, if a string $\boldsymbol{w}$ is accepted by $M_{G'}$, given the one-to-one correspondence between edges and rules, there is a derivation of $\boldsymbol{w}$ from S in G'. Thus, the strings generated by G and the strings accepted by $M_{G'}$ are the same.

Now assume we are given a DFSM M that accepts a language L_M. Create a grammar G_M whose non-terminals are the states of M and whose start symbol is the start state of M. G_M has a rule of the form $q_1 \rightarrow aq_2$ if M makes a transition from state q_1 to q_2 on input a. If state q is a final state of M, add the rule $q \rightarrow \epsilon$. If a string is accepted by M, that is, it causes M to move to a final state, then G_M generates the same string. Since G_M generates only strings of this kind, the language accepted by M is is $L(G_M)$. Now convert G_M to a regular grammar $\widetilde{G}_M$ by replacing each pair of rules $q_1 \rightarrow aq_2$, $q_2 \rightarrow \epsilon$ by the pair $q_1 \rightarrow aq_2$, $q_1 \rightarrow a$, deleting all rules $q \rightarrow \epsilon$ corresponding to unreachable final states q, and deleting the rule $\text{S} \rightarrow \epsilon$ if $\epsilon \in L_M$. Then, $L_M - \{\epsilon\} = L(G_M) - \{\epsilon\} = L(\widetilde{G}_M)$. ■

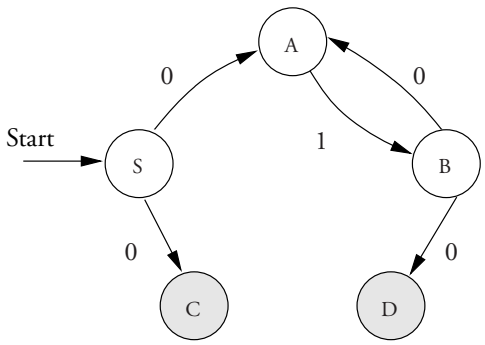


Figure 4.27 A nondeterministic FSM that accepts a language generated by a regular language in which all rules are of the form $\text{A} \rightarrow b\text{C}$ or $\text{A} \rightarrow \epsilon$. A state is associated with each non-terminal, the start symbol S is associated with the start state, and final states are associated with non-terminals A such that $\text{A} \rightarrow \epsilon$. This particular NFSM accepts the language $L(G_4)$ of Example 4.9.4.

A simple example illustrates the construction of an NFSM from a regular grammar. Consider the grammar G_4 of Example 4.9.4. A new grammar G_4' is constructed with the following rules: a) S $\rightarrow$ 0A, b) S $\rightarrow$ 0C, c) C $\rightarrow \epsilon$, d) A $\rightarrow$ 1B, e) B $\rightarrow$ 0A, f) B $\rightarrow$ 0D, and g) D $\rightarrow \epsilon$. Figure 4.27 (page 185) shows an NFSM that accepts the language generated by this grammar. A DFSM recognizing the same language can be obtained by invoking the construction of Theorem 4.2.1.

4.11 Parsing Context-Free Languages

Parsing is the process of deducing those rules of a grammar G (a **derivation**) that generates a terminal string $\boldsymbol{w}$. The first rule must have the start symbol S on the left-hand side. In this section we give a brief introduction to the parsing of context-free languages, a topic central to the parsing of programming languages. The reader is referred to a textbook on compilers for more detail on this subject. (See, for example, [11] and [98].) The concepts of Boolean matrix multiplication and transitive closure are used in this section, topics that are covered in Chapter 6.

Generally a string $\boldsymbol{w}$ has many derivations. This is illustrated by the context-free grammar G_3 defined in Example 4.9.3 and described below.

EXAMPLE 4.11.1 $G_3 = (\mathcal{N}_3, \mathcal{T}_3, \mathcal{R}_3, \text{S})$, *where* $\mathcal{N}_3 = \{\text{S}, \text{M}, \text{N}\}$, $\mathcal{T}_3 = \{\text{A}, \text{B}, \text{C}\}$ *and* $\mathcal{R}_3$ *consists of the rules below:*

$$
\begin{array}{llll}
a) & \text{S} \rightarrow c\text{MN}c & d) & \text{N} \rightarrow b\text{N}b \\
b) & \text{M} \rightarrow a\text{M}a & e) & \text{N} \rightarrow c \\
c) & \text{M} \rightarrow c &
\end{array}
$$

The string $caacaabcbc$ can be derived by applying rules (a), (b) twice, (c), (d) and (e) to produce the following derivation:

$$
\begin{aligned}
\text{S} &\Rightarrow c\text{MN}c \Rightarrow ca\text{M}a\text{N}c \Rightarrow ca^2\text{M}a^2\text{N}c \\
&\Rightarrow ca^2ca^2\text{N}c \Rightarrow ca^2ca^2b\text{N}bc \Rightarrow ca^2ca^2bcbc
\end{aligned}
\tag{4.2}
$$

The same string can be obtained by applying the rules in the following order: (a), (d), (e), (b) twice, and (c). Both derivations are described by the **parse tree** of Fig. 4.28. In this tree each instance of a non-terminal is rewritten using one of the rules of the grammar. The order of the descendants of a non-terminal vertex in the parse tree is the order of the corresponding symbols in the string obtained by replacing this non-terminal. The string ca^2ca^2bcbc, the **yield** of this parse tree, is the terminal string obtained by visiting the leaves of this tree in a left-to-right order. The **height** of the parse tree is the number of edges on the longest path (having the most edges) from the root (associated with the start symbol) to a terminal symbol. A **parser** for a language $L(G)$ is a program or machine that examines a string and produces a derivation of the string if it is in the language and an error message if not.

Because every string generated by a context-free grammar has a derivation, it has a corresponding parse tree. Given a derivation, it is straightforward to convert it to a **leftmost derivation**, a derivation in which the leftmost remaining non-terminal is expanded first. (A **rightmost derivation** is a derivation in which the rightmost remaining non-terminal is expanded first.) Such a derivation can be obtained from the parse tree by deleting all vertices

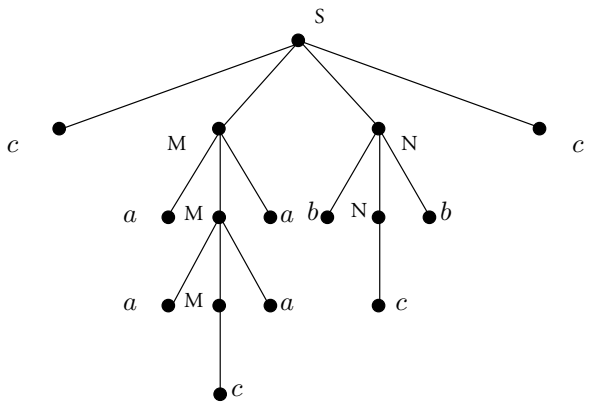


Figure 4.28 A parse tree for the grammar G_3.

associated with terminals and then traversing the remaining vertices in a depth-first manner (visit the first descendant of a vertex before visiting its siblings), assuming that descendants of a vertex are ordered from left to right. When a vertex is visited, apply the rule associated with that vertex in the tree. The derivation given in (4.2) is leftmost.

Not only can some strings in a context-free language have multiple derivations, but in some languages they have multiple parse trees. Languages containing strings with more than one parse tree are said to be **ambiguous languages**. Otherwise languages are **non-ambiguous**.

Given a string that is believed to be generated by a grammar, a **compiler** attempts to parse the string after first scanning the input to identify letters. If the attempt fails, an error message is produced. Given a string generated by a context-free grammar, can we guarantee that we can always find a derivation or parse tree for that string or determine that none exists? The answer is yes, as we now show.

To demonstrate that every CFL can be parsed, it is convenient first to convert the grammar for such a language to Chomsky normal form.

DEFINITION 4.11.1 *A context-free grammar G is in* **Chomsky normal form** *if every rule is of the form* $\text{A} \rightarrow \text{BC}$ *or* $\text{A} \rightarrow u$, $u \in \mathcal{T}$ *except if* $\epsilon \in L(G)$, *in which case* $\text{S} \rightarrow \epsilon$ *is also in the grammar.*

We now give a procedure to convert an arbitrary context-free grammar to Chomsky normal form.

THEOREM 4.11.1 *Every context-free language can be generated by a grammar in Chomsky normal form.*

Proof Let $L = L(G)$ where G is a context-free grammar. We construct a context-free grammar G' that is in Chomsky normal form. The process described in this proof is illustrated by the example that follows.

Initially G' is identical with G. We begin by eliminating all ϵ-**rules** of the form $\text{B} \rightarrow \epsilon$. except for $\text{S} \rightarrow \epsilon$ if $\epsilon \in L(G)$. If either $\text{B} \rightarrow \epsilon$ or $\text{B} \Rightarrow \epsilon$, for every rule that has B on the right-hand side, such as $\text{A} \rightarrow \alpha\text{B}\beta\text{B}\gamma$, $\alpha, \beta, \gamma \in (V - \{\text{B}\})^*$ ($V = \mathcal{N} \cup \mathcal{T}$), we add a rule for each possible replacement of B by ϵ; for example, we add $\text{A} \rightarrow \alpha\beta\text{B}\gamma$, $\text{A} \rightarrow \alpha\text{B}\beta\gamma$,

and $\mathrm{A} \rightarrow \alpha\beta\gamma$. Clearly the strings generated by the new rules are the same as are generated by the old rules.

Let $\mathrm{A} \rightarrow w_1 \cdots w_i \cdots w_k$ for some $k \geq 1$ be a rule in G' where $w_i \in V$. We replace this rule with the new rules $\mathrm{A} \rightarrow Z_1 Z_2 \cdots Z_k$, and $Z_i \rightarrow w_i$ for $1 \leq i \leq k$. Here Z_i is a new non-terminal. Clearly, the new version of G' generates the same language as does G.

With these changes the rules of G' consist of rules either of the form $\mathrm{A} \rightarrow u$, $u \in \mathcal{T}$ (a single terminal) or $\mathrm{A} \rightarrow \boldsymbol{w}$, $\boldsymbol{w} \in \mathcal{N}^+$ (a string of at least one non-terminal). There are two cases of $\boldsymbol{w} \in \mathcal{N}^+$ to consider, a) $|\boldsymbol{w}| = 1$ and b) $|\boldsymbol{w}| \geq 2$. We begin by eliminating all rules of the first kind, that is of the form $\mathrm{A} \rightarrow \mathrm{B}$.

Rules of the form $\mathrm{A} \rightarrow \mathrm{B}$ can be cascaded to form rules of the type $\mathrm{C} \stackrel{*}{\Rightarrow} \mathrm{D}$. The number of distinct derivations of this kind is at most $|\mathcal{N}|!$ because if any derivation contains two instances of a non-terminal, the derivation can be shortened. Thus, we need only consider derivations in which each non-terminal occurs at most once. For each such pair C, D with a relation of this kind, add the rule $\mathrm{C} \rightarrow \mathrm{D}$ to G'. If $\mathrm{C} \rightarrow \mathrm{D}$ and $\mathrm{D} \rightarrow \boldsymbol{w}$ for $|\boldsymbol{w}| \geq 2$ or $\boldsymbol{w} = u \in \mathcal{T}$, add $\mathrm{C} \rightarrow \boldsymbol{w}$ to the set of rules. After adding all such rules, delete all rules of the form $\mathrm{A} \rightarrow \mathrm{B}$. By construction this new set of rules generates the same language as the original set of rules but eliminates all rules of the first kind.

We now replace rules of the type $\mathrm{A} \rightarrow \mathrm{A}_1\mathrm{A}_2 \cdots \mathrm{A}_k$, $k \geq 3$. Introduce $k-2$ new non-terminals $\mathrm{N}_1, \mathrm{N}_2, \cdots, \mathrm{N}_{k-2}$ peculiar to this rule and replace the rule with the following rules: $\mathrm{A} \rightarrow \mathrm{A}_1\mathrm{N}_1$, $\mathrm{N}_1 \rightarrow \mathrm{A}_2\mathrm{N}_2$, $\cdots$, $\mathrm{N}_{k-3} \rightarrow \mathrm{A}_{k-2}\mathrm{N}_{k-2}$, $\mathrm{N}_{k-2} \rightarrow \mathrm{A}_{k-1}\mathrm{A}_k$. Clearly, the new grammar generates the same language as the original grammar and is in the Chomsky normal form. ■

EXAMPLE 4.11.2 *Let* $G_5 = (\mathcal{N}_5, \mathcal{T}_5, \mathcal{R}_5, \mathrm{E})$ *(with start symbol* E*) be the grammar with* $\mathcal{N}_5 = \{\mathrm{E}, \mathrm{T}, \mathrm{F}\}$, $\mathcal{T}_5 = \{a, b, +, *, (,)\}$, *and* $\mathcal{R}_5$ *consisting of the rules given below:*

$a)$ $\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T}$ $\quad$ $d)$ $\mathrm{T} \rightarrow \mathrm{F}$ $\quad$ $f)$ $\mathrm{F} \rightarrow a$

$b)$ $\mathrm{E} \rightarrow \mathrm{T}$ $\quad$ $e)$ $\mathrm{F} \rightarrow (\mathrm{E})$ $\quad$ $g)$ $\mathrm{F} \rightarrow b$

$c)$ $\mathrm{T} \rightarrow \mathrm{T} * \mathrm{F}$

Here E, T, *and* F *denote expressions, terms, and factors. It is straightforward to show that* $\mathrm{E} \stackrel{*}{\Rightarrow} (a * b + a) * (a + b)$ *and* $\mathrm{E} \stackrel{*}{\Rightarrow} a * b + a$ *are two possible derivations.*

We convert this grammar to the Chomsky normal form using the method described in the proof of Theorem 4.11.1. Since $\mathcal{R}$ contains no ϵ-rules, we do not need the rule $\mathrm{E} \rightarrow \epsilon$, nor do we need to eliminate ϵ-rules.

First we convert rules of the form $\mathrm{A} \rightarrow \boldsymbol{w}$ so that each entry in $\boldsymbol{w}$ is a non-terminal. To do this we introduce the non-terminals **(**, **)**, **+**, and ***** and the rules below. Here we use a boldface font to distinguish between the non-terminal and terminal equivalents of these four mathematical symbols. Since we are adding to the original set of rules, we number them consecutively with the original rules.

$h)$ $\mathbf{(} \rightarrow ($ $\quad$ $j)$ $\mathbf{+} \rightarrow +$

$i)$ $\mathbf{)} \rightarrow)$ $\quad$ $k)$ $\mathbf{*} \rightarrow *$

Next we add rules of the form $\mathrm{C} \rightarrow \mathrm{D}$ for all chains of single non-terminals such that $\mathrm{C} \stackrel{*}{\Rightarrow} \mathrm{D}$. Since by inspection $\mathrm{E} \stackrel{*}{\Rightarrow} \mathrm{F}$, we add the rule $\mathrm{E} \rightarrow \mathrm{F}$. For every rule of the form $\mathrm{A} \rightarrow \mathrm{B}$ for which $\mathrm{B} \rightarrow \boldsymbol{w}$, we add the rule $\mathrm{A} \rightarrow \boldsymbol{w}$. We then delete all rules of the form $\mathrm{A} \rightarrow \mathrm{B}$. These

changes cause the rules of G' to become the following. (Below we use a different numbering scheme because all these rules replace rules (a) through (k).)

1)	E	$\rightarrow$	E+T	7)	T	$\rightarrow$	(E)	13)	(	$\rightarrow$	(
2)	E	$\rightarrow$	T∗F	8)	T	$\rightarrow$	a	14)	)	$\rightarrow$	)
3)	E	$\rightarrow$	(E)	9)	T	$\rightarrow$	b	15)	+	$\rightarrow$	+
4)	E	$\rightarrow$	a	10)	F	$\rightarrow$	(E)	16)	∗	$\rightarrow$	∗
5)	E	$\rightarrow$	b	11)	F	$\rightarrow$	a				
6)	T	$\rightarrow$	T∗F	12)	F	$\rightarrow$	b				

We now reduce the number of non-terminals on the right-hand side of each rule to two through the addition of new non-terminals. The result is shown in Example 4.11.3 below, where we have added the non-terminals A, B, C, D, G, and H.

EXAMPLE 4.11.3 *Let $G_6 = (\mathcal{N}_6, \mathcal{T}_6, \mathcal{R}_6, \text{E})$ (with start symbol E) be the grammar with $\mathcal{N}_6 = \{\text{A, B, C, D, E, F, G, H, T}, +, *, (,)\}$, $\mathcal{T}_6 = \{a, b, +, *, (,)\}$, and $\mathcal{R}_6$ consisting of the rules given below.*

(A)	E	$\rightarrow$	EA	(I)	T	$\rightarrow$	TD	(Q)	H	$\rightarrow$	E)
(B)	A	$\rightarrow$	+T	(J)	D	$\rightarrow$	∗F	(R)	F	$\rightarrow$	a
(C)	E	$\rightarrow$	TB	(K)	T	$\rightarrow$	(G	(S)	F	$\rightarrow$	b
(D)	B	$\rightarrow$	∗F	(L)	G	$\rightarrow$	E)	(T)	(	$\rightarrow$	(
(E)	E	$\rightarrow$	(C	(M)	T	$\rightarrow$	a	(U)	)	$\rightarrow$	)
(F)	C	$\rightarrow$	E)	(N)	T	$\rightarrow$	b	(V)	+	$\rightarrow$	+
(G)	E	$\rightarrow$	a	(P)	F	$\rightarrow$	(H	(W)	∗	$\rightarrow$	∗
(H)	E	$\rightarrow$	b								

The new grammar clearly generates the same language as does the original grammar, but it is in Chomsky normal form. It has 22 rules, 13 non-terminals, and six terminals whereas the original grammar had seven rules, three non-terminals, and six terminals.

We now use the Chomsky normal form to show that for every CFL there is a polynomial-time algorithm that tests for membership of a string in the language. This algorithm can be practical for some languages.

THEOREM 4.11.2 *Given a context-free grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$, an $O(n^3|\mathcal{N}|^2)$-step algorithm exists to determine whether or not a string $\boldsymbol{w} \in \mathcal{T}^*$ of length n is in $L(G)$ and to construct a parse tree for it if it exists.*

Proof If G is not in Chomsky normal form, convert it to this form. Given a string $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$, the goal is to determine whether or not $\text{S} \stackrel{*}{\Rightarrow} \boldsymbol{w}$. Let $\emptyset$ denote the empty set. The approach taken is to construct an $(n+1) \times (n+1)$ **set matrix** S whose entries are sets of non-terminals of G with the property that the i, j entry, $a_{i,j}$, is the set of non-terminals C such that $\text{C} \stackrel{*}{\Rightarrow} w_i \cdots w_{j-1}$. Thus, the string $\boldsymbol{w}$ is in $L(G)$ if $\text{S} \in a_{1,n+1}$, since S generates the entire string $\boldsymbol{w}$. Clearly, $a_{i,j} = \emptyset$ for $j \leq i$. We illustrate this construction with the example following this proof.

We show by induction that set matrix S is the transitive closure (denoted B^+) of the $(n+1) \times (n+1)$ set matrix B whose i, j entry $b_{i,j} = \emptyset$ for $j \neq i+1$ when $1 \leq i \leq n$

and $b_{i,i+1}$ is defined as follows:

$$b_{i,i+1} = \{\text{A} \mid (\text{A} \rightarrow w_i) \text{ in } \mathcal{R} \text{ where } w_i \in \mathcal{T}\}$$

$$B = \begin{bmatrix} \emptyset & b_{1,2} & \emptyset & \dots & \emptyset \\ \emptyset & \emptyset & b_{2,3} & \dots & \emptyset \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \emptyset & \emptyset & \emptyset & \dots & b_{n,n+1} \\ \emptyset & \emptyset & \emptyset & \dots & \emptyset \end{bmatrix}$$

Thus, the entry $b_{i,i+1}$ is the set of non-terminals that generate the ith terminal symbol w_i of $\boldsymbol{w}$ in one step. The value of each entry in the matrix B is the empty set except for the entries $b_{i,i+1}$ for $1 \leq i \leq n$, $n = |\boldsymbol{w}|$.

We extend the concept of matrix multiplication (see Chapter 6) to the product of two set matrices. Doing this requires a new definition for the product of two sets (entries in the matrix) as well as for the addition of two sets. The **product $S_1 \cdot S_2$ of sets of nonterminals** S_1 and S_2 is defined as:

$$S_1 \cdot S_2 = \{\text{A} \mid \text{there exists B} \in S_1 \text{ and C} \in S_2 \text{ such that } (\text{A} \rightarrow \text{BC}) \in \mathcal{R}\}$$

Thus, $S_1 \cdot S_2$ is the set of non-terminals for which there is a rule in $\mathcal{R}$ of the form $\text{A} \rightarrow \text{BC}$ where $\text{B} \in S_1$ and $\text{C} \in S_2$. The **sum** of two sets is their union.

The i, j entry of the product $C = D \times E$ of two $m \times m$ matrices D and E, each containing sets of non-terminals, is defined below in terms of the product and union of sets:

$$c_{i,j} = \bigcup_{k=1}^{m} d_{i,k} \cdot e_{k,j}$$

We also define the **transitive closure** C^+ of an $m \times m$ matrix C as follows:

$$C^+ = C^{(1)} \cup C^{(2)} \cup C^{(3)} \cup \cdots C^{(m)}$$

where

$$C^{(s)} = \bigcup_{r=1}^{s-1} C^{(r)} \times C^{(s-r)} \text{ and } C^{(1)} = C$$

By the definition of the matrix product, the entry $b_{i,j}^{(2)}$ of the matrix $B^{(2)}$ is $\emptyset$ if $j \neq i+2$ and otherwise is the set of non-terminals A that produce $w_i w_{i+1}$ through a derivation tree of depth 2; that is, there are rules such that $\text{A} \rightarrow \text{BC}$, $\text{B} \rightarrow w_i$, and $\text{C} \rightarrow w_{i+1}$, which implies that $\text{A} \stackrel{*}{\Rightarrow} w_i w_{i+1}$.

Similarly, it follows that both $B^{(1)}B^{(2)}$ and $B^{(2)}B^{(1)}$ are $\emptyset$ in all positions except $i, i+3$ for $1 \leq i \leq n-2$. The entry in position $i, i+3$ of $B^{(3)} = B^{(1)}B^{(2)} \bigcup B^{(2)}B^{(1)}$ contains the set of non-terminals A that produce $w_i w_{i+1} w_{i+2}$ through a derivation tree of depth 3; that is, $\text{A} \rightarrow \text{BC}$ and either B produces $w_i w_{i+1}$ through a derivation of depth 2 ($\text{B} \stackrel{*}{\Rightarrow} w_i w_{i+1}$) and C produces w_{i+2} in one step ($\text{C} \rightarrow w_{i+2}$) or B produces w_i in one step ($\text{B} \rightarrow w_i$) and C produces $w_{i+1} w_{i+2}$ through a derivation of depth 2 ($\text{C} \stackrel{*}{\Rightarrow} w_{i+1} w_{i+2}$).

Finally, the only entry in $B^{(n)}$ that is not $\emptyset$ is the $1, n+1$ entry and it contains the set of non-terminals, if any, that generate $\boldsymbol{w}$. If S is in this set, $\boldsymbol{w}$ is in $L(G)$.

The transitive closure $S = B^+$ involves $\sum_{r=1}^{n} r = (n+1)n/2$ products of set matrices. The product of two $(n+1) \times (n+1)$ set matrices of the type considered here involves at most n products of sets. Thus, at most $O(n^3)$ products of sets is needed to form S. In turn, a product of two sets, $S_1 \cdot S_2$, can be formed with $O(q^2)$ operations, where $q = |\mathcal{N}|$ is the number of non-terminals. It suffices to compare each pair of entries, one from S_1 and the other from S_2, through a table to determine if they form the right-hand side of a rule.

As the matrices are being constructed, if a pair of non-terminals is discovered that is the right-hand side of a rule, that is, $\text{A} \rightarrow \text{BC}$, then a link can be made from the entry A in the product matrix to the entries B and C. From the entry S in $a_{1,n+1}$, if it exists, links can be followed to generate a parse tree for the input string. ■

The procedure described in this proof can be extended to show that membership in an arbitrary CFL can be determined in time $O(M(n))$, where $M(n)$ is the number of operations to multiply two $n \times n$ matrices [341]. This is the fastest known general algorithm for this problem when the grammar is part of the input. For some CFLs, faster algorithms are known that are based on the use of the deterministic pushdown automaton. For fixed grammars membership algorithms often run in $O(n)$ steps. The reader is referred to books on compilers for such results. The procedure of the proof is illustrated by the following example.

EXAMPLE 4.11.4 *Consider the grammar G_6 of Example 4.11.3. We show how the five-character string $a*b+a$ in $L(G_6)$ can be parsed. We construct the 6×6 matrices $B^{(1)}$, $B^{(2)}$, $B^{(3)}$, $B^{(4)}$, $B^{(5)}$, as shown below. Since $B^{(5)}$ contains* E *in the $1, n+1$ position, $a*b+a$ is in the language. Furthermore, we can follow links between non-terminals (not shown) to demonstrate that this string has the parse tree shown in Fig. 4.29. The matrix $B^{(4)}$ is not shown because each of its entries is $\emptyset$.*

$$B^{(1)} = \begin{bmatrix} \emptyset & \{\text{E,F,T}\} & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \{*\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\text{E,F,T}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \{+\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{\text{E,F,T}\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

$$B^{(2)} = \begin{bmatrix} \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\text{B}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{\text{A}\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix} \qquad B^{(3)} = \begin{bmatrix} \emptyset & \emptyset & \emptyset & \{\text{E}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{\text{E}\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

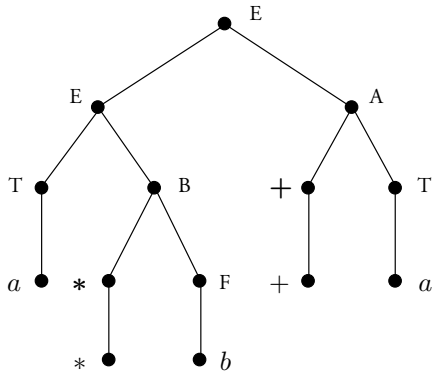


Figure 4.29 The parse tree for the string $a * b + a$ in the language $L(G_6)$.

$$B^{(5)} = \begin{bmatrix} \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{\mathrm{E}\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

4.12 CFL Acceptance with Pushdown Automata*

While it is now clear that an algorithm exists to parse every context-free language, it is useful to show that there is a class of automata that accepts exactly the context-free languages. These are the nondeterministic pushdown automata (PDA) described in Section 4.8.

We now establish the principal results of this section, namely, that the context-free languages are accepted by PDAs and that the languages accepted by PDAs are context-free. We begin with the first result.

THEOREM 4.12.1 *For each context-free grammar G there is a PDA M that accepts $L(G)$. That is, $L(M) = L(G)$.*

Proof Before beginning this proof, we extend the definition of a PDA to allow it to push strings onto the stack instead of just symbols. That is, we extend the stack alphabet Γ to include a small set of strings. When a string such as $abcd$ is pushed, a is pushed before b, b before c, etc. This does not increase the power of the PDA, because for each string we can add unique states that M enters after pushing each symbol except the last. With the pushing of the last symbol M enters the successor state specified in the transition being executed.

Let $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathrm{S})$ be a context-free grammar. We construct a PDA $M = (\Sigma, \Gamma, Q, \Delta, s, F)$, where $\Sigma = \mathcal{T}$, $\Gamma = \mathcal{N} \cup \mathcal{T} \cup \{\gamma\}$ (γ is the blank stack symbol), $Q = \{s, p, f\}$, $F = \{f\}$, and Δ consists of transitions of the types shown below. Here $\forall$ denotes "for all" and $\forall(\mathrm{A} \mapsto \boldsymbol{w}) \in \mathcal{R}$ means for all transitions in $\mathcal{R}$.

a) $(s, \epsilon, \epsilon; p, \text{S})$

b) $(p, a, a; p, \epsilon)$ $\quad \forall a \in \mathcal{T}$

c) $(p, \epsilon, \text{A}; p, \boldsymbol{v})$ $\quad \forall(\text{A} \mapsto \boldsymbol{v}) \in \mathcal{R}$

d) $(p, \epsilon, \gamma; f, \epsilon)$

Let $\boldsymbol{w}$ be placed left-adjusted on the input tape of M. Since $\boldsymbol{w}$ is generated by G, it has a leftmost derivation. (Consider for example that given in (4.2) on page 186.) The PDA begins by pushing the start symbol S onto the stack and entering state p (Rule (a)). From this point on the PDA simulates a leftmost derivation of the string $\boldsymbol{w}$ placed initially on its tape. (See the example that follows this proof.) M either matches a terminal of G on the top of the stack with one under the tape head (Rule (b)) or it replaces a non-terminal on the top of the stack with a rule of $\mathcal{R}$ by pushing the right-hand side of the rule onto the stack (Rule (c)). Finally, when the stack is empty, M can choose to enter the final state f and accept $\boldsymbol{w}$. It follows that any string that can be generated by G can also be accepted by M and vice versa. ■

The leftmost derivation of the string $caacaabcbc$ by the grammar G_3 of Example 4.11.1 is shown in (4.2). The PDA M of the above proof can simulate this derivation, as we show. With the notation T : ... and S : ... (shown below before the computation begins) we denote the contents of the tape and stack at a point in time at which the underlined symbols are those under the tape head and at the top of the stack, respectively. We ignore the blank tape and stack symbols unless they are the ones underlined.

T : $\underline{c}aacaabcbc$ $\qquad$ S : $\underline{\gamma}$

After the first step taken by M, the tape and stack configurations are:

T : $\underline{c}aacaabcbc$ $\qquad$ S : $\underline{\text{S}}$

From this point on M simulates a derivation by G_3. Consulting (4.2), we see that the rule $\text{S} \rightarrow c\text{MN}c$ is the first to be applied. M simulates this with the transition $(p, \epsilon, \text{S}; p, c\text{MN}c)$, which causes S to be popped from the stack and $c\text{MN}c$ to be pushed onto it without advancing the tape head. The resulting configurations are shown below:

T : $\underline{c}aacaabcbc$ $\qquad$ S : $\underline{c}\text{MN}c$

Next the transition $(p, c, c; p, \epsilon)$ is applied to pop one item from the stack, exposing the non-terminal M and advancing the tape head to give the following configurations:

T : $c\underline{a}acaabcbc$ $\qquad$ S : $\underline{\text{M}}\text{N}c$

The subsequent rules, in order, are the following:

1) $\text{M} \rightarrow a\text{M}a$ $\quad$ 2) $\text{M} \rightarrow a\text{M}a$ $\quad$ 3) $\text{M} \rightarrow c$ $\quad$ 4) $\text{N} \rightarrow b\text{N}b$ $\quad$ 5) $\text{N} \rightarrow c$

The corresponding transitions of the PDA are shown in Fig. 4.30.

We now show that the language accepted by a PDA can be generated by a context-free grammar.

$$
\begin{array}{llll}
T: & c\underline{a}acaabcbc & S: & \underline{a}\mathrm{M}a\mathrm{N}c \\
T: & ca\underline{a}caabcbc & S: & \underline{\mathrm{M}}a\mathrm{N}c \\
T: & ca\underline{a}caabcbc & S: & \underline{a}\mathrm{M}aa\mathrm{N}c \\
T: & caa\underline{c}aabcbc & S: & \underline{\mathrm{M}}aa\mathrm{N}c \\
T: & caa\underline{c}aabcbc & S: & \underline{c}aa\mathrm{N}c \\
T: & caac\underline{a}abcbc & S: & \underline{a}a\mathrm{N}c \\
T: & caaca\underline{a}bcbc & S: & \underline{a}\mathrm{N}c \\
T: & caacaa\underline{b}cbc & S: & \underline{\mathrm{N}}c \\
T: & caacaa\underline{b}cbc & S: & \underline{b}\mathrm{N}bc \\
T: & caacaab\underline{c}bc & S: & \underline{\mathrm{N}}bc \\
T: & caacaab\underline{c}bc & S: & \underline{c}bc \\
T: & caacaabc\underline{b}c & S: & \underline{b}c \\
T: & caacaabcb\underline{c} & S: & \underline{c} \\
T: & caacaabcbc\underline{\beta} & S: & \underline{\gamma}
\end{array}
$$

Figure 4.30 PDA transitions corresponding to the leftmost derivation of the string $caacaabcbc$ in the grammar G_3 of Example 4.11.1.

THEOREM 4.12.2 *For each PDA M there is a context-free grammar G that generates the language $L(M)$ accepted by M. That is, $L(G) = L(M)$.*

Proof It is convenient to assume that when the PDA M accepts a string it does so with an empty stack. If M is not of this type, we can design a PDA M' accepting the same language that does meet this condition. The states of M' consist of the states of M plus three additional states, a new initial state s', a cleanup state k, and a new final state f'. Its tape symbols are identical to those of M. Its stack symbols consist of those of M plus one new symbol κ. In its initial state M' pushes κ onto the stack without reading a tape symbol and enters state s, which was the initial state of M. It then operates as M (it has the same transitions) until entering a final state of M, upon which it enters the cleanup state k. In this state it pops the stack until it finds the symbol κ, at which time it enters its final state f'. Clearly, M' accepts the same language as M but leaves its stack empty.

We describe a context-free grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathrm{S})$ with the property that $L(G) = L(M)$. The non-terminals of G consist of S and the triples $< p, y, q >$ defined below denoting goals:

$$< p, y, q > \in \mathcal{N} \text{ where } \mathcal{N} \subset Q \times (\Gamma \cup \{\epsilon\}) \times Q$$

The meaning of $< p, y, q >$ is that M moves from state p to state q in a series of steps during which its only effect on the stack is to pop y. The triple $< p, \epsilon, q >$ denotes the goal of moving from state p to state q leaving the stack in its original condition. Since M starts with an empty stack in state s with a string $\boldsymbol{w}$ on its tape and ends in a final state f with its stack empty, the non-terminal $< s, \epsilon, f >$, $f \in F$, denotes the goal of M moving from state s to a final state f on input $\boldsymbol{w}$, and leaving the stack in its original state.

The rules of G, which represent goal refinement, are described by the following conditions. Each condition specifies a family of rules for a context-free grammar G. Each rule either replaces one non-terminal with another, replaces a non-terminal with the empty string, or rewrites a non-terminal with a terminal or empty string followed by one or two non-terminals. The result of applying a sequence of rules is a string of terminals in the language $L(G)$. Below we show that $L(G) = L(M)$.

1)	S	$\rightarrow$	$< s, \epsilon, f >$	$\forall f \in F$
2)	$< p, \epsilon, p >$	$\rightarrow$	ϵ	$\forall p \in Q$
3)	$< p, y, r >$	$\rightarrow$	$x < q, z, r >$	$\forall r \in Q$ and $\forall (p, x, y; q, z) \in \Delta$, where $y \neq \epsilon$
4)	$< p, u, r >$	$\rightarrow$	$x < q, z, t >< t, u, r >$	$\forall r, t \in Q$, $\forall (p, x, \epsilon; q, z) \in \Delta$, and $\forall u \in \Gamma \cup \{\epsilon\}$

Condition (1) specifies rules that map the start symbol of G onto the goal non-terminal symbol $< s, \epsilon, f >$ for each final state f. These rules insure that the start symbol of G is rewritten as the goal of moving from the initial state of M to a final state, leaving the stack in its original condition.

Condition (2) specifies rules that map non-terminals $< p, \epsilon, p >$ onto the empty string. Thus, all goals of moving from a state to itself leaving the stack in its original condition can be ignored. In other words, no input is needed to take M from state p back to itself leaving the stack unchanged.

Condition (3) specifies rules stating that for all $r \in Q$ and $(p, x, y; q, z)$, $y \neq \epsilon$, that are transitions of M, a goal $< p, y, r >$ to move from state p to state r while removing y from the stack can be accomplished by reading tape symbol x, replacing the top stack symbol y with z, and then realizing the goal $< q, z, r >$ of moving from state q to state r while removing z from the stack.

Condition (4) specifies rules stating that for all $r, t \in Q$ and $(p, x, \epsilon; q, z)$ that are transitions of M, the goal $< p, u, r >$ of moving from state p to state r while popping u for arbitrary stack symbol u can be achieved by reading input x and pushing z on top of u and then realizing the goal $< q, z, t >$ of moving from q to some state t while popping z followed by the goal $< t, u, r >$ of moving from t to r while popping u.

We now show that any string accepted by M can be generated by G and any string generated by G can be accepted by M. It follows that $L(M) = L(G)$. Instead of showing this directly, we establish a more general result.

CLAIM: For all $r, t \in Q$ and $u \in \Gamma \cup \{\epsilon\}$, $< r, u, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$ if and only if the PDA M can move from state r to state t while reading $\boldsymbol{w}$ and popping u from the stack.

The theorem follows from the claim because $< s, \epsilon, f > \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$ if and only if the PDA M can move from initial state s to a final state f while reading $\boldsymbol{w}$ and leaving the stack empty, that is, if and only if M accepts $\boldsymbol{w}$.

We first establish the "if" portion of the claim, namely, if for $r, t \in Q$ and $u \in \Gamma \cup \{\epsilon\}$ the PDA M can move from r to t while reading $\boldsymbol{w}$ and popping u from the stack, then $< r, u, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$. The proof is by induction on the number of steps taken by M. If no

step is taken (basis for induction), $r = t$, nothing is popped and the string ϵ is read by M. Since the grammar G contains the rule $< r, \epsilon, r > \rightarrow \epsilon$, the basis is established.

Suppose that the "if" portion of the claim is true for k or fewer steps (inductive hypothesis). We show that it is true for $k + 1$ steps (induction step). If the PDA M can move from r to t in $k + 1$ steps while reading $\boldsymbol{w} = x\boldsymbol{v}$ and removing u from the stack, then on its first step it must execute a transition $(r, x, y; q, z)$, $q \in Q$, $z \in \Gamma \cup \{\epsilon\}$, for $x \in \Sigma$ with either $y = u$ if $u \neq \epsilon$ or $y = \epsilon$. In the first case, M enters state q, pops u, and pushes z. M subsequently pops z as it reads $\boldsymbol{v}$ and moves to state t in k steps. It follows from the inductive hypothesis that $< q, z, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{v}$. Since $y \neq \epsilon$, a rule of type (3) applies, that is, $< r, y, t > \rightarrow x < q, z, t >$. It follows that $< r, y, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$, the desired conclusion.

In the second case $y = \epsilon$ and M makes the transition $(r, x, \epsilon; q, z)$ by moving from r to t and pushing z while reading x. To pop u, which must have been at the top of the stack, M must first pop z and then pop u. Let it pop z as it moves from q to some intermediate state t' while reading a first portion $\boldsymbol{v_1}$ of the input word $\boldsymbol{v}$. Let it pop u as it moves from t' to t while reading a second portion $\boldsymbol{v_2}$ of the input word $\boldsymbol{v}$. Here $\boldsymbol{v_1 v_2} = \boldsymbol{v}$. Since the move from q to t' and from t' to t each involves at most k steps, it follows that the goals $< q, z, t' >$ and $< t', u, r >$ satisfy $< q, z, t' > \stackrel{*}{\Rightarrow}_G \boldsymbol{v_1}$ and $< t', u, r > \stackrel{*}{\Rightarrow}_G \boldsymbol{v_2}$. Because M's first transition meets condition (4), there is a rule $< r, u, t > \rightarrow x < q, z, t' >< t', u, r >$. Combining these derivations yields the desired conclusion.

Now we establish the "only if" part of the claim, namely, if for all $r, t \in Q$ and $u \in \Gamma \cup \{\epsilon\}$, $< r, u, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$, then the PDA M can move from state r to state t while reading $\boldsymbol{w}$ and removing u from the stack. Again the proof is by induction, this time on the number of derivation steps. If there is a single derivation step (basis for induction), it must be of the type stated in condition (2), namely $< p, \epsilon, p > \rightarrow \epsilon$. Since M can move from state p to p without reading the tape or pushing data onto its stack, the basis is established.

Suppose that the "only if" portion of the claim is true for k or fewer derivation steps (inductive hypothesis). We show that it is true for $k + 1$ steps (induction step). That is, if $< r, u, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$ in $k + 1$ steps, then we show that M can move from r to t while reading $\boldsymbol{w}$ and popping u from the stack. We can assume that the first derivation step is of type (3) or (4) because if it is of type (2), the derivation can be shortened and the result follows from the inductive hypothesis. If the first derivation is of type (3), namely, of the form $< r, u, t > \rightarrow x < q, z, t >$, then by the inductive hypothesis, M can execute $(r, x, u; q, z)$, $u \neq \epsilon$, that is, read x, pop u, push z, and enter state q. Since $< r, u, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{w}$, where $\boldsymbol{w} = x\boldsymbol{v}$, it follows that $< q, z, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{v}$. Again by the inductive hypothesis M can move from q to t while reading $\boldsymbol{v}$ and popping z. Combining these results, we have the desired conclusion.

If the first derivation is of type (4), namely, $< r, u, t > \rightarrow x < q, z, t' >< t', u, t >$, then the two non-terminals $< q, z, t' >$ and $< t', u, t >$ must expand to substrings $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$, respectively, of $\boldsymbol{v}$ where $\boldsymbol{w} = x\boldsymbol{v_1 v_2} = x\boldsymbol{v}$. That is, $< q, z, t' > \stackrel{*}{\Rightarrow}_G \boldsymbol{v_1}$ and $< t', u, t > \stackrel{*}{\Rightarrow}_G \boldsymbol{v_1}$. By the inductive hypothesis, M can move from q to t' while reading $\boldsymbol{v_1}$ and popping z and it can also move from t' to t while reading $\boldsymbol{v_2}$ and popping u. Thus, M can move from r to t while reading $\boldsymbol{w}$ and popping u, which is the desired conclusion. ■

4.13 Properties of Context-Free Languages

In this section we derive properties of context-free languages. We begin by establishing a pumping lemma that demonstrates that every CFL has a certain periodicity property. This property, together with other properties concerning the closure of the class of CFLs under the operations of concatenation, union and intersection, is used to show that the class is not closed under complementation and intersection.

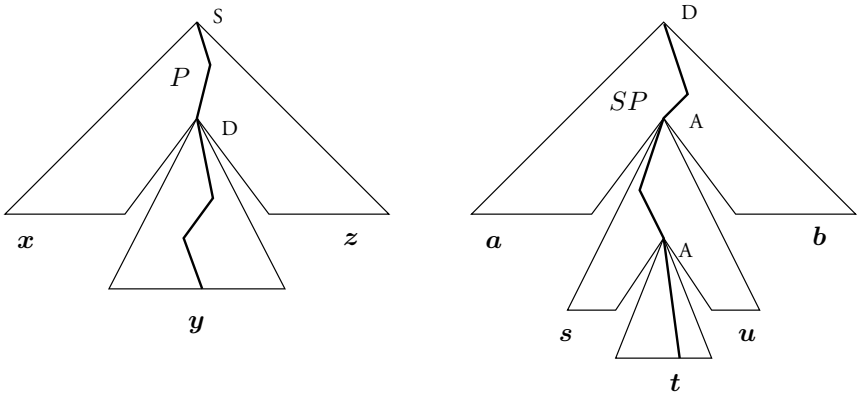
4.13.1 CFL Pumping Lemma

The pumping lemma for regular languages established in Section 4.5 showed that if a regular language contains an infinite number of strings, then it must have strings of a particular form. This lemma was used to show that some languages are not regular. We establish a similar result for context-free languages.

LEMMA 4.13.1 *Let $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ be a context-free grammar in Chomsky normal form with m non-terminals. Then, if $\boldsymbol{w} \in L(G)$ and $|\boldsymbol{w}| \geq 2^{m-1} + 1$, there are strings $\boldsymbol{r}$, $\boldsymbol{s}$, $\boldsymbol{t}$, $\boldsymbol{u}$, and $\boldsymbol{v}$ with $\boldsymbol{w} = \boldsymbol{rstuv}$ such that $|\boldsymbol{su}| \geq 1$ and $|\boldsymbol{stu}| \leq 2^m$ and for all integers $n \geq 0$, $\text{S} \stackrel{*}{\Rightarrow}_G \boldsymbol{rs}^n\boldsymbol{tu}^n\boldsymbol{v} \in L(G)$.*

Proof Since each production is of the form $\text{A} \rightarrow \text{BC}$ or $\text{A} \rightarrow a$, a subtree of a parse tree of height h has a yield (number of leaves) of at most 2^{h-1}. To see this, observe that each rule that generates a leaf is of the form $\text{A} \rightarrow a$. Thus, the yield is the number of leaves in a binary tree of height $h-1$, which is at most 2^{h-1}.

Let $K = 2^{m-1} + 1$. If there is a string $\boldsymbol{w}$ in L of length K or greater, its parse tree has height greater than m. Thus, a longest path P in such a tree (see Fig. 4.31(a)) has more



(a) (b)

Figure 4.31 $L(G)$ is generated by a grammar G in Chomsky normal form with m non-terminals. (a) Each $\boldsymbol{w} \in L(G)$ with $|\boldsymbol{w}| \geq 2^{m-1} + 1$ has a parse tree with a longest path P containing at least $m+1$ non-terminals. (b) SP, the portion of P containing the last $m+1$ non-terminals on P, has a non-terminal A that is repeated. The derivation $\text{A} \rightarrow \boldsymbol{s}\text{A}\boldsymbol{u}$ can be deleted or repeated to generate new strings in $L(G)$.

than m non-terminals on it. Consider the subpath SP of P containing the last $m+1$ non-terminals of P. Let D be the first non-terminal on SP and let the yield of its parse tree be $\boldsymbol{y}$. It follows that $|\boldsymbol{y}| \leq 2^m$. Thus, the yield of the full parse tree, $\boldsymbol{w}$, can be written as $\boldsymbol{w} = \boldsymbol{xyz}$ for strings $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ in $\mathcal{T}^*$.

By the pigeonhole principle stated in Section 4.5, some non-terminal is repeated on SP. Let A be such a non-terminal. Consider the first and second time that A appears on SP. (See Fig. 4.31(b).) Repeat all the rules of the grammar G that produced the string $\boldsymbol{y}$ except for the rule corresponding to the first instance of A on SP and all those rules that depend on it. It follows that $\mathrm{D} \stackrel{*}{\Rightarrow} \boldsymbol{a}\mathrm{A}\boldsymbol{b}$ where $\boldsymbol{a}$ and $\boldsymbol{b}$ are in $\mathcal{T}^*$. Similarly, apply all the rules to the derivation beginning with the first instance of A on P up to but not including the rules beginning with the second instance of A. It follows that $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{s}\mathrm{A}\boldsymbol{u}$, where $\boldsymbol{s}$ and $\boldsymbol{u}$ are in $\mathcal{T}^*$ and at least one is not ϵ since no rules of the form $\mathrm{A} \rightarrow \mathrm{B}$ are in G. Finally, apply the rules starting with the second instance of A on P. Let $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{t}$ be the yield of this set of rules. Since $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{s}\mathrm{A}\boldsymbol{u}$ and $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{t}$, it follows that L also contains $\boldsymbol{xatbz}$. L also contains $\boldsymbol{xas}^n\boldsymbol{tu}^n\boldsymbol{bz}$ for $n \geq 1$ because $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{s}\mathrm{A}\boldsymbol{u}$ can be applied n times after $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{s}\mathrm{A}\boldsymbol{u}$ and before $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{t}$. Now let $\boldsymbol{r} = \boldsymbol{xa}$ and $\boldsymbol{v} = \boldsymbol{bz}$. ■

We use this lemma to show the existence of a language that is not context-free.

LEMMA 4.13.2 *The language* $L = \{a^n b^n c^n \mid n \geq 0\}$ *over the alphabet* $\Sigma = \{a, b, c\}$ *is not context-free.*

Proof We assume that L is context-free generated by a grammar with m non-terminals and show this implies L contains strings not in the language. Let $n_0 = 2^{m-1} + 1$.

Since L is infinite, the pumping lemma can be applied. Let $\boldsymbol{rstuv} = a^n b^n c^n$ for $n = n_0$. From the pumping lemma $\boldsymbol{rs}^2\boldsymbol{tu}^2\boldsymbol{v}$ is also in L. Clearly if $\boldsymbol{s}$ or $\boldsymbol{u}$ is not empty (and at least one is), then they contain either one, two, or three of the symbols in Σ. If one of them, say $\boldsymbol{s}$, contains two symbols, then $\boldsymbol{s^2}$ contains a b before an a or a c before a b, contradicting the definition of the language. The same is true if one of them contains three symbols. Thus, they contain exactly one symbol. But this implies that the number of a's, b's, and c's in $\boldsymbol{rs^2tu^2v}$ is not the same, whether or not $\boldsymbol{s}$ and $\boldsymbol{u}$ contain the same or different symbols. ■

4.13.2 CFL Closure Properties

In Section 4.6 we examined the closure properties of regular languages. We demonstrated that they are closed under concatenation, union, Kleene closure, complementation, and intersection. In this section we show that the context-free languages are closed under concatenation, union, and Kleene closure but not complementation or intersection. A class of languages is closed under an operation if the result of performing the operation on one or more languages in the class produces another language in the class.

The concatenation, union, and Kleene closure of languages are defined in Section 4.3. The concatenation of languages L_1 and L_2, denoted $L_1 \cdot L_2$, is the language $\{\boldsymbol{uv} \mid \boldsymbol{u} \in L_1 \text{ and } \boldsymbol{v} \in L_2\}$. The union of languages L_1 and L_2, denoted $L_1 \cup L_2$, is the set of strings that are in L_1 or L_2 or both. The Kleene closure of a language L, denoted L^* and called the Kleene star, is the language $\bigcup_{i=0}^{\infty} L^i$ where $L^0 = \{\epsilon\}$ and $L^i = L \cdot L^{i-1}$.

THEOREM 4.13.1 *The context-free languages are closed under concatenation, union, and Kleene closure.*

Proof Consider two arbitrary CFLs $L(H_1)$ and $L(H_2)$ generated by grammars $H_1 = (\mathcal{N}_1, \mathcal{T}_1, \mathcal{R}_1, \mathrm{S}_1)$ and $H_2 = (\mathcal{N}_2, \mathcal{T}_2, \mathcal{R}_2, \mathrm{S}_2)$. Without loss of generality assume that their non-terminal alphabets (and rules) are disjoint. (If not, prefix every non-terminal in the second grammar with a symbol not used in the first. This does not change the language generated.)

Since each string in $L(H_1) \cdot L(H_2)$ consists of a string of $L(H_1)$ followed by a string of $L(H_2)$, it is generated by the context-free grammar $H_3 = (\mathcal{N}_3, \mathcal{T}_3, \mathcal{R}_3, \mathrm{S}_3)$ in which $\mathcal{N}_3 = \mathcal{N}_1 \cup \mathcal{N}_2 \cup \{\mathrm{S}_3\}$, $\mathcal{T}_3 = \mathcal{T}_1 \cup \mathcal{T}_2$, and $\mathcal{R}_3 = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{\mathrm{S}_3 \rightarrow \mathrm{S}_1\mathrm{S}_2\}$. The new rule $\mathrm{S}_3 \rightarrow \mathrm{S}_1\mathrm{S}_2$ generates a string of $L(H_1)$ followed by a string of $L(H_2)$. Thus, $L(H_1) \cdot L(H_2)$ is context-free.

The union of languages $L(H_1)$ and $L(H_2)$ is generated by the context-free grammar $H_4 = (\mathcal{N}_4, \mathcal{T}_4, \mathcal{R}_4, \mathrm{S}_4)$ in which $\mathcal{N}_4 = \mathcal{N}_1 \cup \mathcal{N}_2 \cup \{\mathrm{S}_4\}$, $\mathcal{T}_4 = \mathcal{T}_1 \cup \mathcal{T}_2$, and $\mathcal{R}_4 = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{\mathrm{S}_4 \rightarrow \mathrm{S}_1, \mathrm{S}_4 \rightarrow \mathrm{S}_2\}$. To see this, observe that after applying $\mathrm{S}_4 \rightarrow \mathrm{S}_1$ all subsequent rules are drawn from H_1. (The sets of non-terminals are disjoint.) A similar statement applies to the application of $\mathrm{S}_4 \rightarrow \mathrm{S}_2$. Since H_4 is context-free, $L(H_4) = L(H_1) \cup L(H_2)$ is context-free.

The Kleene closure of $L(H_1)$, namely $L(H_1)^*$, is generated by the context-free grammar $H_5 = (\mathcal{N}_1, \mathcal{T}_1, \mathcal{R}_5, \mathrm{S}_1)$ in which $\mathcal{R}_5 = \mathcal{R}_1 \cup \{\mathrm{S}_1 \rightarrow \epsilon, \mathrm{S}_1 \rightarrow \mathrm{S}_1\mathrm{S}_1\}$. To see this, observe that $L(H_5)$ includes ϵ, every string in $L(H_1)$, and, through $i-1$ applications of $\mathrm{S}_1 \rightarrow \mathrm{S}_1\mathrm{S}_1$, every string in $L(H_1)^i$. Thus, $L(H_1)^*$ is generated by H_5 and is context-free. ■

We now use this result and Lemma 4.13.2 to show that the set of context-free languages is not closed under complementation and intersection, operations defined in Section 4.6. The complement of a language L over an alphabet Σ, denoted $\overline{L}$, is the set of strings in Σ^* that are not in L. The intersection of two languages L_1 and L_2, denoted $L_1 \cap L_2$, is the set of strings that are in both languages.

THEOREM 4.13.2 *The set of context-free languages is not closed under complementation or intersection.*

Proof The intersection of two languages L_1 and L_2 can be defined in terms of the complement and union operations as follows:

$$L_1 \cap L_2 = \Sigma^* - (\Sigma^* - L_1) \cup (\Sigma^* - L_2)$$

Thus, since the union of two CFLs is a CFL, if the complement of a CFL is also a CFL, from this identity, the intersection of two CFLs is also a CFL. We now show that the intersection of two CFLs is not always a CFL.

The language $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ is generated by the grammar $H_1 = (\mathcal{N}_1, \mathcal{T}_1, \mathcal{R}_1, \mathrm{S}_1)$, where $\mathcal{N}_1 = \{\mathrm{S}, \mathrm{A}, \mathrm{B}\}$, $\mathcal{T}_1 = \{a, b, c\}$, and the rules $\mathcal{R}_1$ are:

$a)$ $\mathrm{S} \rightarrow \mathrm{AB}$ $\quad$ $d)$ $\mathrm{B} \rightarrow \mathrm{B}c$

$b)$ $\mathrm{A} \rightarrow a\mathrm{A}b$ $\quad$ $e)$ $\mathrm{B} \rightarrow \epsilon$

$c)$ $\mathrm{A} \rightarrow \epsilon$

The language $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$ is generated by the grammar $H_2 = (\mathcal{N}_2, \mathcal{T}_2, \mathcal{R}_2, \mathrm{S}_2)$, where $\mathcal{N}_2 = \{\mathrm{S}, \mathrm{A}, \mathrm{B}\}$, $\mathcal{T}_2 = \{a, b, c\}$ and the rules $\mathcal{R}_2$ are:

$a)$ S $\rightarrow$ AB
$b)$ A $\rightarrow$ aA
$c)$ A $\rightarrow$ ϵ
$d)$ B $\rightarrow$ bBc
$e)$ B $\rightarrow$ ϵ

Thus, the languages L_1 and L_2 are context-free. However, their intersection is $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$, which was shown in Lemma 4.13.2 not to be context-free. Thus, the set of CFLs is not closed under intersection, nor is it closed under complementation. ■

Problems

FSM MODELS

4.1 Let $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ be the FSM model described in Definition 3.1.1. It differs from the FSM model of Section 4.1 in that its output alphabet Ψ has been explicitly identified. Let this machine recognize the language $L(M)$ consisting of input strings $\boldsymbol{w}$ that cause the last output produced by M to be the first letter in Ψ. Show that every language recognized under this definition is a language recognized according to the "final-state definition" in Definition 4.1.1 and vice versa.

4.2 The **Mealy machine** is a seven-tuple $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ identical in its definition with the Moore machine of Definition 3.1.1 except that its output function $\lambda : Q \times \Sigma \mapsto \Psi$ depends on both the current state and input letter, whereas the output function $\lambda : Q \mapsto \Psi$ of the Moore FSM depends only on the current state. Show that the two machines recognize the same languages and compute the same functions with the exception of ϵ.

4.3 Suppose that an FSM is allowed to make state ϵ-transitions, that is, state transitions on the empty string. Show that the new machine model is no more powerful than the Moore machine model.

Hint: Show how ϵ-transitions can be removed, perhaps by making the resultant FSM nondeterministic.

EQUIVALENCE OF DFSMS AND NFSMS

4.4 Functions computed by FSMs are described in Definition 3.1.1. Can a consistent definition of function computation by NFSMs be given? If not, why not?

4.5 Construct a deterministic FSM equivalent to the nondeterministic FSM shown in Fig. 4.32.

REGULAR EXPRESSIONS

4.6 Show that the regular expression $0(0^*10^*)^+$ defines strings starting with 0 and containing at least one 1.

4.7 Show that the regular expressions 0^*, $0(0^*10^*)^+$, and $1(0+1)^*$ partition the set of all strings over 0 and 1.

4.8 Give regular expressions generating the following languages over $\Sigma = \{0, 1\}$:

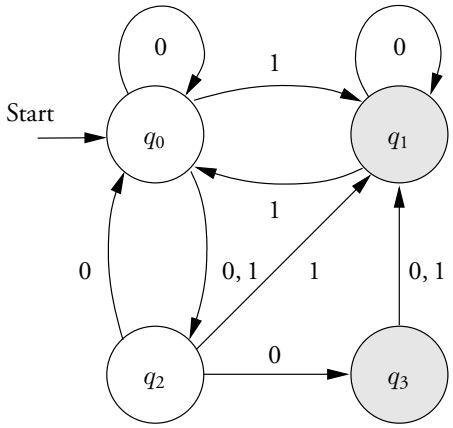


Figure 4.32 A nondeterministic FSM.

a) $L = \{w \mid w$ has length at least 3 and its third symbol is a $0\}$
b) $L = \{w \mid w$ begins with a 1 and ends with a $0\}$
c) $L = \{w \mid w$ contains at least three 1s$\}$

4.9 Give regular expressions generating the following languages over $\Sigma = \{0, 1\}$:
a) $L = \{w \mid w$ is any string except 11 and $111\}$
b) $L = \{w \mid$ every odd position of w is a $1\}$

4.10 Give regular expressions for the languages over the alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} describing positive integers that are:
a) even
b) odd
c) a multiple of 5
d) a multiple of 4

4.11 Give proofs for the rules stated in Theorem 4.3.1.

4.12 Show that $\epsilon + 01 + (010)(10 + 010)^*(\epsilon + 1 + 01)$ and $(01 + 010)^*$ describe the same language.

REGULAR EXPRESSIONS AND FSMS

4.13 a) Find a simple nondeterministic finite-state machine accepting the language $(01 \cup 001 \cup 010)^*$ over $\Sigma = \{0, 1\}$.
b) Convert the nondeterministic finite state machine of part (a) to a deterministic finite-state machine by the method of Section 4.2.

4.14 a) Let $\Sigma = \{0, 1, 2\}$, and let L be the language over Σ that contains each string $\boldsymbol{w}$ ending with some symbol that does not occur anywhere else in $\boldsymbol{w}$. For example, 011012, 20021, 11120, 0002, 10, and 1 are all strings in L. Construct a nondeterministic finite-state machine that accepts L.

b) Convert the nondeterministic finite-state machine of part (a) to a deterministic finite-state machine by the method of Section 4.2.

4.15 Describe an algorithm to convert a regular expression to an NFSM using the proof of Theorem 4.4.1.

4.16 Design DFSMs that recognize the following languages:

a) a^*bca^*

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

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

4.17 Design an FSM that recognizes decimal strings (over the alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} representing the integers whose value is 0 modulo 3.

Hint: Use the fact that $(10)^k = 1 \bmod 3$ (where 10 is "ten") to show that $(a_k(10)^k + a_{k-1}(10)^{k-1} + \cdots + a_1(10)^1 + a_0) \bmod 3 = (a_k + a_{k-1} + \cdots + a_1 + a_0) \bmod 3$.

4.18 Use the above FSM design to generate a regular expression describing those integers whose value is 0 modulo 3.

4.19 Describe an algorithm that constructs an NFSM from a regular expression r and accepts a string $\boldsymbol{w}$ if $\boldsymbol{w}$ contains a string denoted by r that begins anywhere in $\boldsymbol{w}$.

THE PUMPING LEMMA

4.20 Show that the following languages are not regular:

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

b) $L = \{0^n 1^{2n} 0^n \mid n \geq 1\}$

c) $L = \{a^n b^n c^n \mid n \geq 0\}$

4.21 Strengthen the pumping lemma for regular languages by demonstrating that if L is a regular language over the alphabet Σ recognized by a DFSM with m states and it contains a string $\boldsymbol{w}$ of length m or more, then any substring $\boldsymbol{z}$ of $\boldsymbol{w}$ ($\boldsymbol{w} = \boldsymbol{uzv}$) of length m can be written as $\boldsymbol{z} = \boldsymbol{rst}$, where $|\boldsymbol{s}| \geq 1$ such that for all integers $n \geq 0$, $\boldsymbol{urs}^n\boldsymbol{tv} \in L$. Explain why this pumping lemma is stronger than the one stated in Lemma 4.5.1.

4.22 Show that the language $L = \{a^i b^j \mid i > j\}$ is not regular.

4.23 Show that the following language is not regular:

a) $\{u^n z v^m z w^{n+m} \mid n, m \geq 1\}$

PROPERTIES OF REGULAR LANGUAGES

4.24 Use Lemma 4.5.1 and the closure property of regular languages under intersection to show that the following languages are not regular:

a) $\{\boldsymbol{w}\boldsymbol{w}^R \mid \boldsymbol{w} \in \{0, 1\}^*\}$

b) $\{\boldsymbol{w}\overline{\boldsymbol{w}} \mid$ where $\overline{\boldsymbol{w}}$ denotes $\boldsymbol{w}$ in which 0's and 1's are interchanged$\}$

c) $\{\boldsymbol{w} \mid \boldsymbol{w}$ has equal number of 0's and 1's$\}$

4.25 Prove or disprove each of the following statements:

a) Every subset of a regular language is regular

b) Every regular language has a proper subset that is also a regular language

c) If L is regular, then so is $\{\boldsymbol{xy} \mid \boldsymbol{x} \in L \text{ and } \boldsymbol{y} \notin L\}$

d) If L is a regular language, then so is $\{\boldsymbol{w} : \boldsymbol{w} \in L \text{ and } \boldsymbol{w}^R \in L\}$

e) $\{\boldsymbol{w} \mid \boldsymbol{w} = \boldsymbol{w}^R\}$ is regular

STATE MINIMIZATION

4.26 Find a minimal-state FSM equivalent to that shown in Fig. 4.33.

4.27 Show that the languages recognized by M and $M_{\equiv}$ are the same, where $\equiv$ is the equivalence relation on M defined by states that are indistinguishable by input strings of any length.

4.28 Show that the equivalence relation R_L is right-invariant.

4.29 Show that the equivalence relation R_M is right-invariant.

4.30 Show that the right-invariance equivalence relation (defined in Definition 4.7.2) for the language $L = \{a^n b^n \mid n \geq 0\}$ has an unbounded number of equivalence classes.

4.31 Show that the DFSM in Fig. 4.20 is the machine M_L associated with the language $L = (10^*1 + 0)^*$.

PUSHDOWN AUTOMATA

4.32 Construct a pushdown automaton that accepts the following language: $L = \{\boldsymbol{w} \mid \boldsymbol{w}$ is a string over the alphabet $\Sigma = \{(,)\}$ of balanced parentheses$\}$.

4.33 Construct a pushdown automaton that accepts the following language: $L = \{\boldsymbol{w} \mid \boldsymbol{w}$ contains more 1's than 0's$\}$.

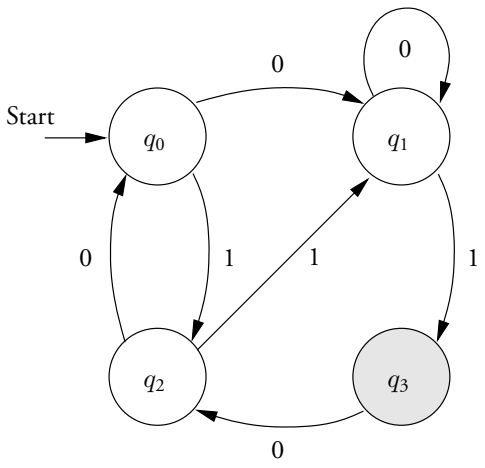


Figure 4.33 A four-state finite-state machine.

PHRASE STRUCTURE LANGUAGES

4.34 Give phrase-structure grammars for the following languages:

a) $\{\boldsymbol{ww} \mid \boldsymbol{w} \in \{a, b\}^*\}$

b) $\{0^{2^i} \mid i \geq 1\}$

4.35 Show that the following language can be described by a phrase-structure grammar:

$$\{a^i \mid i \text{ is not prime}\}$$

CONTEXT-SENSITIVE LANGUAGES

4.36 Show that every context-sensitive language can be accepted by a **linear bounded automaton** (LBA), a nondeterministic Turing machine in which the tape head visits a number of cells that is a constant multiple of the number of characters in the input string $\boldsymbol{w}$.

Hint: Consider a construction similar to that used in the proof of Theorem 5.4.2. Instead of using a second tape, use a second track on the tape of the TM.

4.37 Show that every language accepted by a linear bounded automaton can be generated by a context-sensitive language.

Hint: Consider a construction similar to that used in the proof of Theorem 5.4.1 but instead of deleting characters at the end of TM configuration, encode the end markers [and] by enlarging the tape alphabet of the LBA to permit the first and last characters to be either marked or unmarked.

4.38 Show that the grammar G_1 in Example 4.9.1 is context-sensitive and generates the language $L(G_1) = \{a^n b^n c^n \mid n \geq 1\}$.

4.39 Show that the language $\{0^{2^i} \mid i \geq 1\}$ is context-sensitive.

4.40 Show that the context-sensitive languages are closed under union, intersection, and concatenation.

CONTEXT-FREE LANGUAGES

4.41 Show that language generated by the context-free grammar G_3 of Example 4.9.3 is $L(G_3) = \{ca^n ca^n cb^m cb^m c \mid n, m \geq 0\}$.

4.42 Construct context-free grammars for each of the following languages:

a) $\{\boldsymbol{ww}^R \mid \boldsymbol{w} \in \{a, b\}^*\}$

b) $\{\boldsymbol{w} \mid \boldsymbol{w} \in \{a, b\}^*, \boldsymbol{w} = \boldsymbol{w}^R\}$

c) $L = \{\boldsymbol{w} \mid \boldsymbol{w}$ has twice as many 0's as 1's$\}$

4.43 Give a context-free grammars for each of the following languages:

a) $\{\boldsymbol{w} \in \{a, b\}^* \mid \boldsymbol{w}$ has twice as many a's as b's$\}$

b) $\{a^r b^s \mid r \leq s \leq 2r\}$

REGULAR LANGUAGES

4.44 Show that the regular language G_4 described in Example 4.9.4 is $L(G_4) = (01)^*0$.

4.45 Show that grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$, where $\mathcal{N} = \{\text{A}, \text{B}, \text{S}\}$, $\mathcal{T} = \{a, b\}$ and the rules $\mathcal{R}$ are given below, is regular.

$a)$ S $\rightarrow$ abA $\quad$ $d)$ S $\rightarrow$ ϵ $\quad$ $f)$ B $\rightarrow$ aS

$b)$ S $\rightarrow$ baB $\quad$ $e)$ A $\rightarrow$ bS $\quad$ $g)$ A $\rightarrow$ b

$c)$ S $\rightarrow$ B

Give a derivation for the string $abbbaa$.

4.46 Provide a regular grammar generating strings over $\{0, 1\}$ not containing 00.

4.47 Give a regular grammar for each of the following languages and show that there is a FSM that accepts it. In all cases $\Sigma = \{0, 1\}$.

a) $L = \{\boldsymbol{w} \mid$ the length of $\boldsymbol{w}$ is odd$\}$

b) $L = \{\boldsymbol{w} \mid \boldsymbol{w}$ contains at least three 1s$\}$

REGULAR LANGUAGE RECOGNITION

4.48 Construct a finite-state machine that recognizes the language generated by the grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$, where $\mathcal{N}$ = {S, X, Y}, $\mathcal{T} = \{x, y\}$, and $\mathcal{R}$ contains the following rules: S $\rightarrow$ xX, S $\rightarrow$ yY, X $\rightarrow$ yY, Y $\rightarrow$ xX, X $\rightarrow$ ϵ, and Y $\rightarrow$ ϵ.

4.49 Describe finite-state machines that recognize the following languages:

a) $\{\boldsymbol{w} \in \{a, b\}^* \mid \boldsymbol{w}$ has an odd number of a's$\}$

b) $\{\boldsymbol{w} \in \{a, b\}^* \mid \boldsymbol{w}$ has ab and ba as substrings$\}$

4.50 Show that, if L is a regular language, then the language obtained by reversing the letters in each string in L is also regular.

4.51 Show that, if L is a regular language, then the language consisting of strings in L whose reversals are also in L is regular.

PARSING CONTEXT-FREE LANGUAGES

4.52 Use the algorithm of Theorem 4.11.2 to construct a parse tree for the string $(a * b + a) * (a + b)$ generated by the grammar G_5 of Example 4.11.2, and give a leftmost and a rightmost derivation for the string.

4.53 Let $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ be the context-free grammar with $\mathcal{N} = \text{S}$ and $\mathcal{T} = \{(,), 0\}$ with rules $\mathcal{R} = \{\text{S} \rightarrow 0, \text{S} \rightarrow \text{SS}, \text{S} \rightarrow (\text{S})\}$. Use the algorithm of Theorem 4.11.2 to generate a parse tree for the string $(0)((0))$.

CFL ACCEPTANCE WITH PUSHDOWN AUTOMATA

4.54 Construct PDAs that accept each of the following languages:

a) $\{a^n b^n \mid n \geq 0\}$

b) $\{\boldsymbol{w}\boldsymbol{w}^R \mid \boldsymbol{w} \in \{a, b\}^*\}$

c) $\{\boldsymbol{w} \mid \boldsymbol{w} \in \{a, b\}^*, \boldsymbol{w} = \boldsymbol{w}^R\}$

4.55 Construct PDAs that accept each of the following languages:

a) $\{\boldsymbol{w} \in \{a, b\}^* \mid \boldsymbol{w} \text{ has twice as many } a\text{'s as } b\text{'s}\}$

b) $\{a^r b^s \mid r \leq s \leq 2r\}$

4.56 Use the algorithm of Theorem 4.12.2 to construct a context-free grammar that accepts the language accepted by the PDA in Example 4.8.2.

4.57 Construct a context-free grammar for the language $\{\boldsymbol{w}c\boldsymbol{w}^R \mid \boldsymbol{w} \in \{a, b\}^*\}$.

Hint: Use the algorithm of Theorem 4.12.2 to construct a context-free grammar that accepts the language accepted by the PDA in Example 4.8.1.

PROPERTIES OF CONTEXT-FREE LANGUAGES

4.58 Show that the intersection of a context-free language and a regular language is context-free.

Hint: From machines accepting the two language types, construct a machine accepting their intersection.

4.59 Suppose that L is a context-free language and R is a regular one. Is $L - R$ necessarily context-free? What about $R - L$? Justify your answers.

4.60 Show that, if L is context-free, then so is $L^R = \{\boldsymbol{w}^R \mid \boldsymbol{w} \in L\}$.

4.61 Let $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ be context-free. A non-terminal A is **self-embedding** if and only if $\text{A} \overset{*}{\Rightarrow}_G s\text{A}u$ for some $s, u \in \mathcal{T}$.

a) Give a procedure to determine whether $\text{A} \in \mathcal{N}$ is self-embedding.

b) Show that, if G does not have a self-embedding non-terminal, then it is regular.

CFL PUMPING LEMMA

4.62 Show that the following languages are not context-free:

a) $\{0^{2^i} \mid i \geq 1\}$

b) $\{b^{n^2} \mid n \geq 1\}$

c) $\{0^n \mid n \text{ is a prime}\}$

4.63 Show that the following languages are not context-free:

a) $\{0^n 1^n 0^n 1^n \mid n \geq 0\}$

b) $\{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$

c) $\{\boldsymbol{w}\boldsymbol{w} \mid \boldsymbol{w} \in \{0, 1\}^*\}$

4.64 Show that the language $\{\boldsymbol{w}\boldsymbol{w} \mid \boldsymbol{w} \in \{a, b\}^*\}$ is not context-free.

CFL CLOSURE PROPERTIES

4.65 Let M_1 and M_2 be pushdown automata accepting the languages $L(M_1)$ and $L(M_2)$. Describe PDAs accepting their union $L(M_1) \cup L(M_2)$, concatenation $L(M_1) \cdot L(M_2)$, and Kleene closure $L(M_1)^*$, thereby giving an alternate proof of Theorem 4.13.1.

4.66 Use closure under concatenation of context-free languages to show that the language $\{\boldsymbol{w}\boldsymbol{w}^R\boldsymbol{v}^R\boldsymbol{v} \mid \boldsymbol{w}, \boldsymbol{v} \in \{a, b\}^*\}$ is context-free.

Chapter Notes

The concept of the finite-state machine is often attributed to McCulloch and Pitts [210]. The models studied today are due to Moore [222] and Mealy [214]. The equivalence of deterministic and non-deterministic FSMs (Theorem 4.4.1) was established by Rabin and Scott [265].

Kleene established the equivalence of regular expressions and finite-state machines. The proof used in Theorems 4.4.1 and 4.4.2 is due to McNaughton and Yamada [211]. The pumping lemma (Lemma 4.5.1) is due to to Bar-Hillel, Perles, and Shamir [28]. The closure properties of regular expressions are due to McNaughton and Yamada [211].

State minimization was studied by Huffman [143] and Moore [222]. The Myhill-Nerode Theorem was independently obtained by Myhill [226] and Nerode [228]. Hopcroft [138] has given an efficient algorithm for state miminization.

Chomsky [68,69] defined four classes of formal language, the regular, context-free, context-sensitive, and phrase-structure languages. He and Miller [71] demonstrated the equivalence of languages generated by regular grammars and those recognized by finite-state machines. Chomsky introduced the normal form that carries his name [69]. Oettinger [232] introduced the pushdown automaton and Schutzenberger [304], Chomsky [70], and Evey [96] independently demonstrated the equivalence of context-free languages and pushdown automata.

Two efficient algorithms for parsing context-free languages were developed by Earley [93] and Cocke (unpublished) and independently by Kasami [161] and Younger [370]. These are cubic-time algorithms. Our formulation of the parsing algorithm of Section 4.11 is based on Valiant's derivation [341] of the Cocke-Kasami-Younger recognition matrix, where he also presents the fastest known general algorithm to parse context-free languages. The CFL pumping lemma and the closure properties of CFLs are due to Bar-Hillel, Perles, and Shamir [28].

Myhill [227] introduced the deterministic linear-bounded automata and Landweber [188] showed that languages accepted by linear-bounded automata are context-sensitive. Kuroda [183] generalized the linear-bounded automata to be nondeterministic and established the equivalence of such machines and the context-sensitive languages.