Finite-State Languages

We are ready to begin studying the intimate relationship between abstract machines and languages generated by formal grammars. In this chapter we treat this relationship at the simplest level, that of finite-state accepters and regular languages. Not only will this study add considerably to our understanding of these machines and languages, but it will unify the concepts treated in previous chapters.

Our immediate goal is to demonstrate that these three statements are equivalent:

1. The language $L$ is recognized by some finite-state accepter.
2. The language $L$ is generated by some regular grammar.
3. The language $L$ is described by some regular expression.

5.1 Finite-State Accepters

We noted in Chapter 1 that a relation between abstract machines and languages can be established by means of accepters, machines that classify each input string as being either accepted or rejected. A finite-state machine with state assigned outputs and output alphabet $\{0, 1\}$ can be viewed as such an accepter, the states of the machine being divided into two classes:

1. Accepting states: those with output 1.
2. Rejecting states: those with output 0.
The language recognized by a finite-state accepter consists precisely of those strings that take the machine from its initial state to an accepting state. Instead of specifying the accepting states by means of an output function, it is easier simply to identify the subset of the machine's states that are accepting states. In a state diagram, nodes for accepting states will be drawn as double circles.

5.1.1 Nondeterministic Accepters

Our first step is to relate finite-state accepters to regular grammars. A grammar contains a set of productions that may be applied in any consistent order to derive a terminal string, and is permissive in the sense that there are points during a derivation at which a choice must be made among several applicable productions. In contrast, the finite-state machines studied in Chapter 4 are imperative in the sense that each transition is uniquely determined by the preceding state and present input symbol: no alternative behavior is allowed.

To simplify the treatment of the two systems, it is convenient to introduce a generalization of the finite-state accepter that is permissive in nature. The generalization permits any number of states to be successors of a given state for a given input symbol.

**Example 5.1:** The machine shown in Figure 5.1 is a permissive machine. If it is in state A, two transitions are possible for input symbol 1:

\[ A \xrightarrow{1} B \quad \text{and} \quad A \xrightarrow{1} C \]

Also, for some combinations of state and input symbol, no transition is specified in the figure; specifically, state A has no 0-successor and state C has no 1-successor.

![Figure 5.1. Nondeterministic accepter.](image)

Because the successor of a state is not always unique, the transitions of such a generalized accepter can no longer be specified by a function \( f: Q \times S \rightarrow Q \). Instead we must specify which triples from the set \( Q \times S \times Q \)
are transitions of the machine. Moreover, we do not require that the initial state of the accepter be unique.

**Definition 5.1:** A finite-state accepter (FSA) is a five-tuple

\[ M = (Q, S, P, I, F) \]

in which

- \( Q \) is a finite set of states
- \( S \) is a finite input alphabet
- \( I \subseteq Q \) are the initial states
- \( F \subseteq Q \) are the final states
- \( P \subseteq Q \times S \times Q \) is the transition relation of \( M \): whenever \((q, s, q')\) is an element of \( P \), then

\[ q \xrightarrow{s} q' \]

is a transition of \( M \).

Because the behavior of an FSA is not necessarily deterministic, we sometimes call an FSA a nondeterministic finite-state accepter. The definitions of \( s \)-successor, \( \omega \)-successor, and admissible state sequence presented in Chapter 4 apply without qualification to such accepters.

If the transition relation \( P \) of a finite-state accepter is a function, then each state in the accepter will have a unique successor for each input symbol. In such a case, the behavior of the machine for an input sequence depends only on the given sequence and on the state in which the machine is started. If there is only one choice of initial state, the machine will behave deterministically.

**Definition 5.2:** Let \( M = (Q, S, P, I, F) \) be a finite-state accepter. If the transitions \( P \) of \( M \) constitute a function \( P : Q \times S \rightarrow Q \), and if \( M \) has exactly one initial state, then \( M \) is a deterministic finite-state accepter.

Like the finite-state machines of Chapter 4, deterministic FSA's associate a unique state sequence with each string of input symbols. In a nondeterministic accepter, however, a state may have more than one successor for a given input symbol, and there is not necessarily a unique admissible state sequence for each input string. For these machines, we can no longer think of input strings as "causing" state transitions. Instead, we must now regard input strings as specifying paths through a state diagram. For some strings there may be several paths; for others there may be none.
Example 5.2: The accepter $M$ of Figure 5.1 has three admissible state sequences for the input string 101:

\[
\begin{align*}
A & \xrightarrow{1} B \xrightarrow{0} B \xrightarrow{1} C \\
A & \xrightarrow{1} C \xrightarrow{0} A \xrightarrow{1} B \\
A & \xrightarrow{1} C \xrightarrow{0} A \xrightarrow{1} C
\end{align*}
\]

Thus we have both

\[
A \xrightarrow{101} C \quad \text{and} \quad A \xrightarrow{101} B
\]

Since for a given FSA $M$ we may have both

\[
q \xrightarrow{\omega} q' \quad \text{and} \quad q \xrightarrow{\omega} q'', \quad q' \neq q''
\]

it is no longer appropriate to say that $\omega$ takes $M$ from state $q$ to state $q'$. Rather, we shall say that $\omega$ may lead $M$ from state $q$ to state $q'$ (and also that $\omega$ may lead $M$ from state $q$ to $q''$). The following statements are equivalent:

1. Accepter $M$ has an admissible state sequence from state $q$ to state $q'$ for input string $\omega$.
2. State $q'$ is an $\omega$-successor of state $q$:

\[
q \xrightarrow{\omega} q'
\]

3. There is a directed path from $q$ to $q'$ in the state diagram of $M$ with transitions labeled by the symbols of $\omega$.

A finite-state accepter accepts a string $\omega$ just if there is an admissible state sequence for $\omega$ from some initial state of the accepter to some final state.

Definition 5.3: Let $M = (Q, S, P, I, F)$ be an FSA. Then $M$ accepts a string $\omega \in S^*$ if and only if

\[
q \xrightarrow{\omega} q'
\]

for some $q \in I$ and some $q' \in F$. The language recognized by $M$ is the set

\[
L(M) = \{ \omega \in S^* | M \text{ accepts } \omega \}
\]

Since the final states of an accepter are those which indicate acceptance of an input string, the terms final state and accepting state are used interchangeably.

Example 5.3: Consider the behavior of the accepter in Figure 5.1 for the input string 1011. The allowed sequences of transitions for 1011 and its prefixes can be represented by a tree diagram (Figure 5.2).
Certain sequences terminate before the final input symbol because state C has no 1-successor. Since C is the accepting state, the diagram shows that strings 1, 101, and 1011 are accepted, and that the string 10 is rejected.

5.1.2 Conversion to Deterministic Accepters

How might we relate the behavior of a nondeterministic automaton to the operation of a physical device? We could imagine that the machine is in a definite state at each time instant and, when confronted with alternative transitions for some input symbol, makes an arbitrary choice of its next state. (A tree diagram such as Figure 5.2 represents the paths the machine could possibly follow. To determine whether a given string is accepted, we must imagine operating the machine a sufficient number of times that each route through the state diagram is attempted.) Alternatively, we might imagine that whenever several transitions are possible, the machine splits into identical copies that simultaneously pursue alternative paths. In the case of finite-state automata, there is yet a third useful point of view. We can consider the machine to be in some combination of states at each time instant, according to our uncertainty of the true state of the nondeterministic machine. This idea leads directly to a procedure for obtaining a deterministic machine equivalent to any given FSA.

Example 5.4: The information in Figure 5.2 concerning the behavior of the accepter M (redrawn in Figure 5.3) for the input sequence 1011 can be represented by a sequence of transitions between sets of states:

\[
\{A\} \rightarrow \{B, C\} \rightarrow \{A, B\} \rightarrow \{B, C\} \rightarrow \{C\}
\]

The appearance of a state set in this sequence means that there are paths in the state diagram by which the corresponding string of input
symbols leads from state A to each state in the set. For instance, the appearance of \(\{A, B\}\) in the sequence signifies that

\[ A \xrightarrow{10} A \quad \text{and} \quad A \xrightarrow{10} B \]

are both possible.

The subset of states to which an input string can lead a machine from an initial state is called the \textit{reachable set} for that string. Using the concept of reachable sets, we may describe the behavior of \(M\) for an arbitrary input string.

\textbf{Example 5.4 (continued):} Initially, \(M\) must be in state A. The 1-successor of A is either B or C, and there are no 0-successors of A. Thus Figure 5.3b describes the reachable sets of \(M\) for all strings of length 0 or 1. If \(M\) is in state B or C and a 0 input occurs, the successor state must be either A or B. If a 1 input occurs, the successor
state can only be C. This is shown in Figure 5.3c. Continuing this procedure, we obtain the tree diagram in Figure 5.3d, in which each set of states appearing at the end of a path has already appeared elsewhere in the diagram.

The set of states that may be reached by an s-transition from some other set of states is determined solely by the state diagram of the machine. Therefore, recurrences of the same reachable set do not have to be distinguished in the diagram, and we may merge identical nodes in Figure 5.3d to obtain Figure 5.4a. This figure resembles the

Figure 5.4. Construction of an equivalent deterministic accepter.
state diagram of a deterministic accepter, except that no transitions are specified for a 0 input in reachable set \{A\} or a 1 input in reachable set \{C\}. These transitions are missing because the original machine \( M \) specifies no states as 0-successors of \( A \) or as 1-successors of \( C \). Strings that require the use of these missing transitions do not lead to any state of the nondeterministic accepter: the reachable set for these strings is the empty set \( \emptyset \). If we include \( \emptyset \) as a reachable set (Figure 5.4b), the diagram becomes the state diagram of a deterministic finite-state accepter (Figure 5.4c). The states of the deterministic machine are in one-to-one correspondence with the reachable sets of \( M \).

If \( \emptyset \) is the reachable set for some string \( \omega \), then \( \emptyset \) is the reachable set for any string having \( \omega \) as a prefix. Thus all transitions from the \( \emptyset \) state return to the \( \emptyset \) state, and the \( \emptyset \) state is a trap state of the accepter.

Example 5.4 suggests that one can always convert a nondeterministic finite-state accepter into a deterministic accepter that recognizes the same language. Let

\[
M_n = (Q_n, S_n, P_n, I_n, F_n)
\]

be any finite-state accepter, and suppose that we wish to construct a deterministic accepter

\[
M_d = (Q_d, S_d, P_d, I_d, F_d)
\]

such that \( L(M_d) = L(M_n) \). Presenting a string \( \omega \) to \( M_n \) may lead it to any one of several states. Let the set of possible states be

\[
X_{[\omega]} = \{q' \in Q_n | q \xrightarrow{\omega} q' \text{ for some } q \in I_n\}
\]

The set \( X_{[\omega]} \) contains the reachable states of \( M_n \) for the input string \( \omega \). If no symbol has been applied to \( M_n \), the machine must be in one of its initial states; hence

\[
X_{[\epsilon]} = I_n
\]

For each string \( \varphi \cdot s \), we can express \( X_{[\varphi \cdot s]} \) in terms of \( X_{[\varphi]} \): a state \( q' \) will be reachable for \( \varphi \cdot s \) if and only if

\[
q'' \xrightarrow{\varphi} q'
\]

is a transition of \( M_n \) for some \( q'' \in X_{[\varphi]} \). That is,

(1) \[
X_{[\varphi \cdot s]} = \{q' \in Q_n | q'' \xrightarrow{\varphi} q' \text{ for some } q'' \in X_{[\varphi]}\}
\]

Thus \( X_{[\varphi \cdot s]} \) is uniquely determined by the input symbol \( s \) and the set \( X_{[\varphi]} \). The number of distinct reachable sets \( X_{[\omega]} \) will be finite because each is a subset of the finite set \( Q_n \). By providing \( M_d \) with a state corresponding to each reachable set and with transitions consistent with equation (1), we
obtain a deterministic accepter for $L(M_n)$. Since $X_{\text{fin}}$ contains a final state of $M_n$ if and only if $\omega \in L(M_n)$, the final states of $M_d$ correspond to reachable sets that contain final states of $M_n$. The details of the construction are as follows:

1. The elements of $Q_d$ are the subsets of $Q_n$ reachable for some input string:

$$Q_d = \{X_{\omega} | \omega \in S^*\}$$

2. The initial state of $M_d$ is $X_{\text{fin}} = I_n$.

3. The accepting states of $M_d$ are the reachable sets that contain accepting states of $M_n$:

$$F_d = \{X \in Q_d | X \cap F_n \neq \emptyset\}$$

4. State $X'$ is the $s$-successor of state $X$ in $M_d$ just if $X'$ consists precisely of the $s$-successors in $M_n$ of the members of $X$:

$$X \xrightarrow{\omega} X' \quad \text{in } M_d$$

if and only if

$$X' = \{q' | q \xrightarrow{\omega} q' \text{ in } M_n \text{ for some } q \in X\}$$

**Theorem 5.1:** For each finite-state accepter $M_n$ one can construct a deterministic finite-state accepter $M_d$ such that $L(M_d) = L(M_n)$.

**Proof:** Let $M_d$ be the FSA constructed from $M_n$ by the procedure given above. Clearly, $M_d$ is deterministic. We must show that $L(M_d) = L(M_n)$.

A state $X_{\omega}$ of $M_d$ is accepting if and only if it contains an accepting state of $M_n$. Since any state in $X_{\omega}$ is reachable for $\omega$, we have

$$X_{\text{fin}} \in F_d \quad \text{if and only if } \omega \in L(M_n)$$

Thus it is sufficient to show that each string $\omega$ leads $M_d$ to the state $X_{\omega}$:

$$X_{\text{fin}} \xrightarrow{\omega} X_{\omega}, \quad \text{each } \omega \in S^*$$

We use an induction on the length of $\omega$.

**Basis:** Certainly $X_{\text{fin}} \xrightarrow{\varepsilon} X_{\text{fin}}$.

**Induction:** Let $\omega = \varphi \cdot s$, and suppose that

$$X_{\text{fin}} \xrightarrow{\varphi} X_{\varphi}$$

A state $q'$ is reachable for $\omega$ just if

$$q \xrightarrow{\varphi} q'' \xrightarrow{s} q'$$
for some $q \in X_{(x)}$ and some state $q''$ reachable for $\varphi$. That is,

$$q' \in X_{(y)} \quad \text{if and only if} \quad q'' \xrightarrow{\varepsilon} q' \quad \text{for some} \quad q'' \in X_{(y)}$$

It follows from construction rule 4 that $M_\varphi$ has the transition

$$X_{(y)} \xrightarrow{\varepsilon} X_{(y)}$$

Using (2), we conclude that

$$X_{(1)} \xrightarrow{\omega} X_{(y)}$$

Thus, permitting nondeterministic behavior does not increase the language-recognizing ability of finite-state machines.

### 5.1.3 Applications to Machine Design

Nondeterministic accepters are frequently useful in the design of finite-state machines. Two examples will illustrate.

#### Example 5.5: We shall design a deterministic FSA that accepts any string consisting entirely of 0's except for a single occurrence of either the substring 101 or a substring of 1's. [The set of strings satisfying this property is represented by the expression $0^*(101 \cup 11*)0^*$.] The construction is shown in Figure 5.5. Accepters $M_1$ and $M_3$ recognize the sets 101 and 11*, respectively. So that an arbitrary string of 0's may precede or follow either of these sets, we add self-loops to the initial states and to the accepting states to obtain accepters $M_3$ and $M_4$. In $M_4$, a new state $G$ must be included so that a final string of 0's can be accepted only after the last of a string of 1's has been presented. To form an accepter for the union of $L(M_3)$ and $L(M_4)$, we regard the state diagrams of $M_3$ and $M_4$ as jointly constituting a single nondeterministic accepter $M_5$. The corresponding deterministic machine $M_5$ is found by starting with the set $\{A, E\}$ of initial states, and determining what additional state sets must be included to account for all paths in $M_5$. States $BF$, $CG$, $D$, $F$, and $G$ are all accepting states in $M_5$, because they correspond to reachable sets that contain accepting states of $M_5$.

The construction of an equivalent deterministic accepter can also be carried out using state tables:

#### Example 5.6: Suppose that we wish to construct a deterministic accepter for the set of all binary strings containing, at any position, the substring 0110. One might find it difficult to design the required accepter directly, because prefixes of 0110 may appear in an input
string with less than four symbols of separation, as in the string 010110. A deterministic accepter must therefore keep track of all possible interpretations of previously presented symbols as initial symbols of the required substring.

Designing a nondeterministic accepter for the language and converting it to a deterministic accepter is an organized way of solving the problem. In Figure 5.6a, the self-loops at states A and E
permit arbitrary sequences of input symbols to precede and follow the string 0110. Figure 5.6b is a state table for this accepter. Non-determinism appears in the table as blank entries and as entries specifying multiple successor states. A state table for an equivalent deterministic accepter is constructed by associating rows of the table with reachable sets of the given machine. The first row is labeled by the initial states (just state A in this example); then the successor set for each combination of row and input symbol is added to the table as a new row. For instance, in the original machine the 0-successors of state A are the states A and B, and thus a row labeled AB is added to the table for the new machine, as shown in Figure 5.6c. The completed table is shown as Figure 5.6d; the final states of the machine are those marked with a 1 in the output column. Note that states ABE, ACE, ADE, and AE are all equivalent: our procedure for constructing a deterministic accepter does not necessarily yield a reduced machine.

Figure 5.6. Construction for Example 5.6.
5.2 Finite-State Accepters and Regular Grammars

The relation between finite-state accepters and regular grammars is established as a correspondence between the sets of strings denoted by the nonterminal letters of a grammar and certain sets of strings associated with states of an accepter. We treat the relationship of right-linear grammars to finite-state accepters explicitly, because these grammars play the more important role in later chapters. An entirely analogous development may be formulated in terms of left-linear grammars.

Given a grammar $G$, we write $L(G, A)$ to mean the set of terminal strings derivable in $G$ from the nonterminal letter $A$:

$$L(G, A) = \{ \omega \in T^* | A \xrightarrow{*} \omega \text{ in } G \}$$

For any right-linear grammar we shall see how these sets may be identified with certain sets of strings, called end sets, associated with the states of a finite-state accepter $M$.

Definition 5.4: Let $M = (Q, S, P, I, F)$ be an rfa. The end set $E(q)$ of a state $q$ of $M$ is the collection of input strings that can lead from $q$ to an accepting state of $M$:

$$E(q) = \{ \omega \in S^* | q \xrightarrow{\omega} q' \text{ for some } q' \in F \}$$

Example 5.7: Consider the finite-state accepter $M$ shown in Figure 5.7. The end set $E(A)$ is the set of strings that lead $M$ from state $A$ to the accepting state $D$. A string in $E(A)$ must consist of a 1 followed by some string that leads $M$ from state $B$ to state $D$:

(1) $E(A) = 1 \cdot E(B)$

Similarly, a string in $E(C)$ must be a 0 followed by some string in $E(B)$:

(2) $E(C) = 0 \cdot E(B)$

Finally, a string in $E(B)$ can be either a single 1 leading directly to the accepting state $D$, or a 0 followed by some string in $E(C)$:

(3) $E(B) = 0 \cdot E(C) \cup 1$
Relations (1), (2), and (3) comprise a system of set equations that must be satisfied by the end sets of the accepter \( M \).

Note that the end set of state \( D \) is \( \lambda \), because \( D \) is accepting and no transitions exit state \( D \). Thus relation (3) could be written alternatively as

\[
(3') \quad \mathcal{E}(B) = 0 \cdot \mathcal{E}(C) \cup 1 \cdot \mathcal{E}(D)
\]

\[
(3'') \quad \mathcal{E}(D) = \lambda
\]

The language recognized by \( M \) is easily expressed in terms of end sets: it consists of precisely those strings in the end set of \( M \)'s initial state. That is, \( L(M) = \mathcal{E}(A) \).

Example 5.7 has shown how a system of set equations may be derived from a finite-state accepter. Before generalizing this example, we introduce some fundamental properties of end sets. It should be clear that the end set \( \mathcal{E}(q) \) of an accepting state always contains the empty string. Conversely, if \( q \) is not an accepting state, then \( \lambda \) is not in \( \mathcal{E}(q) \), since at least one input symbol is required to reach another state of \( M \). Now suppose that \( \omega \in \mathcal{E}(q) \) for nonempty \( \omega \). There must be an admissible state sequence for \( \omega \) consisting of at least one transition. If \( s \) is the first letter of \( \omega \), this state sequence has the form

\[
q \xrightarrow{s} q' \xrightarrow{r} q'',
\]

where \( \omega = s \cdot \varphi \) and \( q'' \in F \). It follows that there is a state \( q' \) of \( M \) such that

\[
q \xrightarrow{s} q'
\]

is a state transition of \( M \), and the string \( \varphi \) is in the end set \( E(q') \). The converse is also true. Finally, a string \( \omega \) is accepted by \( M \) if and only if \( M \) has an admissible state sequence

\[
q \xrightarrow{\omega} q'
\]

where \( q \) and \( q' \) are initial and accepting states, respectively. It follows that the language recognized by \( M \) is the union of the end sets of all initial states of \( M \).

**Proposition 5.1:** Let \( M = (Q, S, P, I, F) \) be an FSA, and let \( E(q) \) be the end set of state \( q \). Then

1. \( \lambda \in E(q) \) if and only if \( q \in F \).
2. If \( \omega = s \cdot \varphi \), then \( \omega \in E(q) \) if and only if \( \varphi \in E(q') \) and \( q \xrightarrow{s} q' \) is a transition of \( M \) for some \( q' \in Q \).
3. \( L(M) = \bigcup_{q \in I} E(q) \).

**Proposition 5.2:** The end sets of a finite-state accepter \( M = (Q, S, P, I, F) \) satisfy a system of right-linear set equations: for each
$q \in Q,$

$$E(q) = \bigcup_{q' \in Q} V(q, q')E(q') \cup W(q)$$

where

$$V(q, q') = \{s \in S | M \text{ has } q \xrightarrow{s} q'\}$$

and

$$W(q) = \begin{cases} \lambda & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

Each coefficient set $V(q, q')$ contains the input symbols that can take $M$ from $q$ to $q'$.

Linear set equations are analogous to linear algebraic equations: the set operations union and concatenation are analogous respectively to the arithmetic operations addition and multiplication. (Keep in mind, however, that concatenation is not a commutative operation.) The equations are linear because the end sets, which are the unknowns, make at most one appearance in each term. They are right linear because the unknown sets are the right constituent of each term making up the right-hand side. We shall see in Section 5.3 how the system of set equations obtained from a finite-state accepter may be solved to yield regular expressions for the end sets in terms of the coefficient sets of the equations.

In setting up the equation system for an accepter, it helps to note that the end set of a nonaccepting trap state is empty; terms containing such end sets may be deleted from set equations without affecting their solution. Also, if $q$ is an inaccessible state, then the set equation for $E(q)$ may be omitted from the equation system without affecting the solution for $L(M)$.

If $M$ is a finite-state accepter having a transition

$$q \xrightarrow{s} q'$$

then, according to Proposition 5.1, the string $s\varphi$ is in $E(q)$ whenever $\varphi$ is in $E(q')$. Thus we may assert that

$$E(q) \supseteq s \cdot E(q')$$

If $G$ is a right-linear grammar that has a production

$$A \rightarrow sB$$

then the string $s\varphi$ is denoted by $A$ whenever $\varphi$ is denoted by $B$, and we may assert that

$$L(G, A) \supseteq s \cdot L(G, B)$$

The similarity of relations (1) and (2) suggests that, by relating rules of a right-linear grammar to the transitions of an accepter, we may be able to identify the end sets of the accepter with the sets of terminal strings denoted by nonterminal symbols of the grammar.
Example 5.8: Consider the grammar

\[ G: \quad \Sigma \rightarrow A \quad A \rightarrow 1B \quad B \rightarrow 1 \]
\[ B \rightarrow 0C \quad C \rightarrow 0B \]

From the \( \Sigma \) rule of \( G \) we find that

\[ L(\Sigma) = L(A) \]

where, for convenience, we are writing \( L(G, X) \) as simply \( L(X) \). From the single \( A \) rule we have

(1) \[ L(A) = 1 \cdot L(B) \]

and the \( C \) rule yields

(2) \[ L(C) = 0 \cdot L(B) \]

The two \( B \) rules imply that

(3) \[ L(B) = 0 \cdot L(C) \cup 1 \]

Relations (1), (2), and (3) make up a system of right-linear set equations identical to the set equations formulated for the accepter \( M \) in Example 5.7, except the unknowns here are \( L(A), L(B), \) and \( L(C) \) instead of \( E(A), E(B), \) and \( E(C) \). Since any solution of either system is also a solution of the other, and since \( L(M) = E(A) \) and \( L(G) = L(A) \), the language generated by \( G \) is exactly the language recognized by \( M \).

The equality of the languages defined by \( M \) and \( G \) stems from the relationship of the structure of \( M \) to the productions in \( G \):

\[
\begin{array}{ll}
\text{In } M & \text{In } G \\
A \xrightarrow{1} B & A \rightarrow 1B \\
B \xrightarrow{0} C & B \rightarrow 0C \\
C \xrightarrow{0} B & C \rightarrow 0B \\
B \xrightarrow{1} D, D \in F & B \rightarrow 1 \\
A \in I & \Sigma \rightarrow A \\
\end{array}
\]

Because of this relationship, there is a one-to-one correspondence between the admissible state sequences of \( M \) and the derivations of \( G \). For each \( k \geq 0 \), the derivation

\[ \Sigma \Rightarrow A \rightarrow 1B \rightarrow 10C \rightarrow 100B \rightarrow \ldots \rightarrow 1(00)^k B \Rightarrow 1(00)^k 1 \]

corresponds to the accepting state sequence

\[ A \xrightarrow{1} B \xrightarrow{0} C \xrightarrow{0} B \xrightarrow{0} \ldots \xrightarrow{0} B \xrightarrow{1} D \]

for the string \( 1(00)^k 1 \).
Example 5.8 illustrates the principle used to construct a right-linear grammar for the language recognized by an arbitrary finite-state accepter \( M \). The grammar has a nonterminal letter for each state of the accepter:

\[
N = \{ N(q) \mid q \in Q \}
\]

The productions of the grammar are obtained from the transitions of \( M \) so that

\[
N(q) \xrightarrow{*} \omega \quad \text{if and only if} \quad \omega \in E(q), \, \omega \neq \lambda
\]

The rules of the construction are stated in Table 5.1 and are explained in Example 5.9. According to the definition given in Chapter 3, a grammar \( G \) constructed according to Table 5.1 is not strictly right linear because of

<table>
<thead>
<tr>
<th>Rule</th>
<th>If ( M ) has</th>
<th>then ( G ) has</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( I \cap F \neq \emptyset )</td>
<td>( \Sigma \rightarrow \lambda )</td>
<td>( \lambda \in L(M) )</td>
</tr>
<tr>
<td>2</td>
<td>( q \in I )</td>
<td>( \Sigma \rightarrow N(q) )</td>
<td>( E(q) \subseteq L(M) )</td>
</tr>
<tr>
<td>3</td>
<td>( q \xrightarrow{s} q', q' \in F )</td>
<td>( N(q) \rightarrow s )</td>
<td>( s \in E(q) )</td>
</tr>
<tr>
<td>4</td>
<td>( q \xrightarrow{s} q' )</td>
<td>( N(q) \rightarrow sN(q') )</td>
<td>( E(q) \equiv sE(q') )</td>
</tr>
</tbody>
</table>

productions of the form \( \Sigma \rightarrow N(q) \) resulting from rule 2. However, if we add to \( G \) the production \( \Sigma \rightarrow \omega \) for each production \( N(q) \rightarrow \omega \) resulting from rules 3 and 4, we can remove the production \( \Sigma \rightarrow N(q) \) without changing the language generated by the grammar. Thus any grammar generated according to Table 5.1 can be transformed, if we so desire, into one that is strictly right linear and that generates the same language.

**Example 5.9:** We shall construct a right-linear grammar \( G \) from the finite-state accepter \( M \) shown in Figure 5.8. Let the end sets \( E(A) \), \( E(B) \), and \( E(C) \) of the accepter be written simply as \( A \), \( B \), and \( C \). These sets are related by the right-linear set equations

\[
A = 1A \cup 1C
\]

\[
B = 0A \cup 1B \cup 1C \cup \lambda
\]

\[
C = 1B \cup 0C \cup \lambda
\]

and \( L(M) = A \cup B \).
The nonterminal symbols of G are

\[ N = \{A, B, C\} \]

in correspondence with the end sets of M. The productions of G are obtained using the rules of Table 5.1.

First, since state B is both an initial and final state, G has the production \( \Sigma \rightarrow \lambda \) as required by construction rule 1.

Second, \( L(M) \) is the union of sets A and B. By construction rule 2, G has the productions

\[ \Sigma \rightarrow A \quad \text{and} \quad \Sigma \rightarrow B \]

Third, G has \( N(q) \rightarrow s \) whenever \( s \in E(q) \), by construction rule 3. As noted in Table 5.1,

\[ s \in E(q) \quad \text{if and only if} \quad M \text{ has } q \rightarrow q', q' \in F \]

In the case of state A, M has

\[ A \rightarrow C, \quad C \in F \]

and therefore \( 1 \in A \). Accordingly, G has the production

\[ A \rightarrow 1 \]

Similarly, the transitions

\[ B \rightarrow B \quad C \rightarrow C \]

\[ B \rightarrow C \quad C \rightarrow B \]

where B and C are final states, require that G have the productions

\[ B \rightarrow 1 \quad C \rightarrow 0 \quad C \rightarrow 1 \]

Next we consider the set equation

\[ A = 1A \cup 1C \]
A nonempty string $\omega$ is in $A$ if and only if $\omega = 1\varphi$ and $\varphi$ is in either $A$ or $C$. In $G$ the corresponding requirement is met by including the productions

$$A \rightarrow 1A \quad A \rightarrow 1C$$

as required by construction rule 4. The remaining two set equations require the productions

$$B \rightarrow 0A \quad C \rightarrow 1B$$
$$B \rightarrow 1B \quad C \rightarrow 0C$$
$$B \rightarrow 1C$$

by the same reasoning.

Summarizing, the grammar $G$ is

$$G: \quad \text{rule 1: } \Sigma \rightarrow \lambda \quad \text{rule 4: } A \rightarrow 1A$$
$$\text{rule 2: } \Sigma \rightarrow A \quad A \rightarrow 1C$$
$$\Sigma \rightarrow B \quad B \rightarrow 0A$$
$$\text{rule 3: } A \rightarrow 1 \quad B \rightarrow 1B$$
$$B \rightarrow 1 \quad B \rightarrow 1C$$
$$C \rightarrow 0 \quad C \rightarrow 1B$$
$$C \rightarrow 1 \quad C \rightarrow 0C$$

The relation between state sequences in $M$ and derivations in $G$ is illustrated in Figure 5.9. Two accepting state sequences for the

Figure 5.9. Relation between derivations of $G$ and state sequences of $M$.  

string 1101 are shown, together with the corresponding derivations in \( G \). The nonterminals of \( G \) appear in the derivations in the same order as the corresponding states appear in the state sequences. Figure 5.9 shows that \( G \) is an ambiguous grammar.

Example 5.9 illustrates the one-to-one correspondence between the state sequences of an accepter and the leftmost derivations in the corresponding grammar. In the following proof, this is established as a general property of the construction, and is the basis for our treatment of ambiguity in Section 5.4.

**Theorem 5.2:** For any finite-state accepter \( M \), one can construct a right-linear grammar \( G \) such that \( L(G) = L(M) \).

**Proof:** Let \( M = (Q, S, P, I, F) \) be given, and let \( G = (N, S, P, \Sigma) \) be the right-linear grammar constructed according to Table 5.1. We must show that \( \omega \in L(G) \) if and only if \( \omega \in L(M) \).

We have \( \lambda \in L(M) \) if and only if \( I \cap F \neq \emptyset \). But, by construction, \( G \) has the production \( \Sigma \rightarrow \lambda \) if and only if \( I \cap F \neq \emptyset \); hence \( \lambda \in L(G) \) if and only if \( \lambda \in L(M) \).

The construction rules establish one-to-one correspondences between transitions in \( M \) and productions in \( G \):

- \( q \xrightarrow{t} q' \) in \( M \) maps to \( N(q) \rightarrow sN(q') \) in \( G \).
- \( q \xrightarrow{t} q', q' \in F \) in \( M \) maps to \( N(q) \rightarrow s \) in \( G \).

It follows by a simple induction that, for each nonempty string \( \omega = s_1s_2 \ldots s_k \), there is a one-to-one correspondence between state sequences

\[
q_0 \xrightarrow{s_1} q_1 \xrightarrow{s_2} \ldots \xrightarrow{s_k} q_k,
\]

and derivations

\[
N(q_0) \rightarrow s_1N(q_1) \rightarrow s_1s_2N(q_2) \rightarrow \ldots \rightarrow s_1s_2 \ldots s_{k-1}N(q_{k-1}) \rightarrow s_1s_2 \ldots s_k
\]

in \( G \). Therefore,

\[
(1) \quad \omega \in E(q_0) \quad \text{if and only if} \quad N(q_0) \xrightarrow{*} \omega
\]

and this holds for each \( q_0 \) in \( Q \).

The construction also establishes a one-to-one correspondence of each end set \( E(q), q \in I \), with a production \( \Sigma \rightarrow N(q) \) in \( G \). Hence

\[
L(G) = \bigcup_{q \in I} L(G, N(q))
\]
and since

\[ L(M) = \bigcup_{q \in I} E(q) \]

it follows from (1) that \( L(M) = L(G) \).

**Corollary 5.2.1:** The state sequences in \( M \) from \( q \in I \) to \( q' \in F \) are in one-to-one correspondence with the derivations of terminal strings from \( \Sigma \) in \( G \).

**Corollary 5.2.2:** For each \( q \), the end set \( E(q) \) of \( M \) and the nonterminal \( N(q) \) in \( G \) satisfy the relation \( L(G, N(q)) = E(q) \).

The construction rules given in Table 5.1 may be reversed to yield an accepter that recognizes the language generated by any right-linear grammar. The rules of the construction are given in Table 5.2 and explained in Example 5.10. For simplicity, we have assumed that each \( \Sigma \) rule of \( G \) is of the form \( \Sigma \rightarrow \lambda \) or \( \Sigma \rightarrow A \).

<table>
<thead>
<tr>
<th>Rule</th>
<th>If ( G ) has</th>
<th>then ( M ) has</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A \rightarrow sB )</td>
<td>( q_A \overset{s}{\rightarrow} q_B )</td>
<td>( L(G, A) \supseteq sL(G, B) )</td>
</tr>
<tr>
<td>2</td>
<td>( A \rightarrow s )</td>
<td>( q_A \overset{s}{\rightarrow} q_F )</td>
<td>( s \in L(G, A) )</td>
</tr>
<tr>
<td>3</td>
<td>( \Sigma \rightarrow A )</td>
<td>( q_A \in I )</td>
<td>( L(G, A) \subseteq L(G) )</td>
</tr>
<tr>
<td>4</td>
<td>( \Sigma \rightarrow \lambda )</td>
<td>( q_F \in I )</td>
<td>( \lambda \in L(G) )</td>
</tr>
</tbody>
</table>

**Example 5.10:** Consider the language

\[ L = a^*b^* \cup (ab)^* \]

Using the nonterminal letter \( A \) to denote strings in \( a^*b^* \) and the nonterminal letter \( C \) to denote strings in \( (ab)^* \), the following grammar generates \( L \):

\[
\begin{align*}
G: & \quad \Sigma \rightarrow \lambda \quad A \rightarrow aA \quad C \rightarrow aD \\
& \quad \Sigma \rightarrow A \quad A \rightarrow aB \\
& \quad \Sigma \rightarrow C \quad B \rightarrow bB \quad D \rightarrow bC \quad (ab)^* \rightarrow \lambda \\
& \quad A \rightarrow a \\
& \quad A \rightarrow b \\
& \quad B \rightarrow b
\end{align*}
\]
The corresponding accepter $M$ (Figure 5.10) will have states $A$, $B$, $C$, $D$, and $q_F$, and transitions arranged so that the strings denoted by each nonterminal of $G$ constitute the end set of the corresponding state in $M$. From the productions $A \rightarrow aA$ and $A \rightarrow aB$

![Diagram](image_url)

we see that the end set $A$ must include the sets $aA$ and $aB$. This is accomplished by including in $M$ the transitions

$$A \xrightarrow{a} A \quad \text{and} \quad A \xrightarrow{a} B$$

as required by rule 1 of Table 5.2. The remaining transitions of $M$ are accounted for in a similar way.

Next, the productions

$$A \rightarrow aA \quad A \rightarrow b$$

require that the symbols $a$ and $b$ both be in the end set $A$. This is accomplished by including transitions

$$A \xrightarrow{a} q_F \quad A \xrightarrow{b} q_F$$

in $M$, where $q_F$ is the accepting state, as specified by rule 2 of Table 5.2. The remaining transitions into $q_F$ are accounted for in a similar way.

The productions

$$\Sigma \rightarrow A \quad \Sigma \rightarrow C$$

require that $L(M)$ contain both end sets $A$ and $C$. Therefore, states $A$ and $C$ are made initial states of $M$, as required by rule 3 of Table
5.2. Finally, the production $\Sigma \rightarrow \lambda$ requires that $q_0$ be an initial state, as specified by rule 4 of Table 5.2.

Example 5.10 illustrates how very naturally a grammar corresponds to a nondeterministic state diagram, and why we have allowed accepters to be nondeterministic. Should we desire, an accepter constructed from a grammar according to Table 5.2 can be converted to a deterministic accepter by the construction of Section 5.1.2. This is often a far simpler procedure than constructing a deterministic accepter directly from a given grammar.

**Theorem 5.3:** For any right-linear grammar $G$, one can construct a finite-state accepter $M$ such that $L(M) = L(G)$.

**Proof:** Let $M$ be the accepter constructed according to Table 5.2. The argument that $L(M) = L(G)$ is essentially the same as given in the proof of Theorem 5.2.

Theorems 5.2 and 5.3 assert the equivalence, in terms of the class of languages that they define, of finite-state accepters and right-linear grammars. In Chapter 3 we defined regular grammars to include left-linear grammars as well. That left-linear grammars are equivalent to right-linear grammars, and thus that finite-state accepters are equivalent to regular grammars, is demonstrated quite simply.

Suppose that $G$ is a right-linear grammar. Let $G'$ be the left-linear grammar specified by the following construction:

- If $G$ has $\Sigma \rightarrow sA$, then $G'$ has $A \rightarrow s$
- If $G$ has $A \rightarrow sB$, then $G'$ has $B \rightarrow As$
- If $G$ has $A \rightarrow s$, then $G'$ has $\Sigma \rightarrow As$
- If $G$ has $\Sigma \rightarrow s$, then $G'$ has $\Sigma \rightarrow s$
- If $G$ has $\Sigma \rightarrow \lambda$, then $G'$ has $\Sigma \rightarrow \lambda$

This one-to-one correspondence between the productions of $G$ and the productions of $G'$ establishes a one-to-one correspondence between derivations in $G$ and $G'$. A derivation

$$\Sigma \rightarrow s_1A_1 \rightarrow s_1s_2A_2 \rightarrow \ldots \rightarrow s_1s_2\ldots s_kA_k \rightarrow s_1s_2\ldots s_{k+1}$$

in $G$ corresponds to the derivation

$$\Sigma \rightarrow A_k s_{k+1} \rightarrow A_{k-1}s_k s_{k+1} \rightarrow \ldots \rightarrow A_1s_2\ldots s_k s_{k+1} \rightarrow s_1s_2\ldots s_{k+1}$$

in $G'$. Hence $L(G') = L(G)$. 
Example 5.11: Consider the right-linear grammar

\begin{align*}
G: & \quad \Sigma \rightarrow \lambda \quad (1) \quad A \rightarrow aA \quad (7) \\
& \quad \Sigma \rightarrow aA \quad (2) \quad A \rightarrow aB \quad (8) \\
& \quad \Sigma \rightarrow aB \quad (3) \quad B \rightarrow bB \quad (9) \\
& \quad \Sigma \rightarrow bB \quad (4) \quad A \rightarrow a \quad (10) \\
& \quad \Sigma \rightarrow a \quad (5) \quad B \rightarrow b \quad (11) \\
& \quad \Sigma \rightarrow b \quad (6)
\end{align*}

for which \( L(G) = a^*b^* \). Following the construction given, a left-linear grammar equivalent to \( G \) is

\begin{align*}
G': & \quad \Sigma \rightarrow \lambda \quad (1') \quad A \rightarrow Aa \quad (7') \\
& \quad A \rightarrow a \quad (2') \quad B \rightarrow Aa \quad (8') \\
& \quad B \rightarrow a \quad (3') \quad B \rightarrow Bb \quad (9') \\
& \quad B \rightarrow b \quad (4') \quad \Sigma \rightarrow Aa \quad (10') \\
& \quad \Sigma \rightarrow a \quad (5') \quad \Sigma \rightarrow Bb \quad (11') \\
& \quad \Sigma \rightarrow b \quad (6')
\end{align*}

The derivations of the sentence \( aabbb \) according to \( G \) and \( G' \) are

\begin{align*}
G: & \quad \Sigma \rightarrow aA \Rightarrow aAB \Rightarrow aabB \Rightarrow aabbb \Rightarrow aabbb \\
G': & \quad \Sigma \rightarrow Bb \Rightarrow Bbb \Rightarrow Bb \Rightarrow Bb \Rightarrow Aabbb \Rightarrow aabbb
\end{align*}

Left-linear grammars can also be obtained directly from finite-state accepters by a construction similar to that used for right-linear grammars. One defines the begin set \( B(q) \) of state \( q \) to be the set of strings leading to state \( q \) from some initial state

\[ B(q) = \{ \omega | q' \xrightarrow{\omega} q, q' \in I \} \]

The left-linear grammar \( G \) has nonterminals \( \{N(q) | q \in Q\} \) and productions chosen so that \( L(G, N(q)) = B(q) \). The construction rules, set equations, and other aspects of this approach are analogous to those developed above. They are explored in detail in the Problems.

The principal results developed so far may be summarized by asserting the equivalence of the following statements:

1. The language \( L \) is recognized by some finite-state accepter.
2. The language \( L \) is generated by some right-linear grammar.
3. The language \( L \) is generated by some left-linear grammar.

In addition, we can state some new facts about regular grammars:

1. From a given right-linear grammar, one can always obtain a left-linear grammar that generates the same language, and vice versa.
2. One can always determine whether two regular grammars generate the same language: they generate the same language if and only if their corresponding deterministic accepters are equivalent.

3. One can always determine whether a regular grammar \( G \) generates any strings at all [that is, whether \( L(G) \) is empty]: \( L(G) \) is empty if and only if there is no path from any initial state to any final state in the state diagram of the corresponding accepter.

### 5.3 Regular Expressions and Finite-State Accepters

The state diagram of a finite-state accepter and the productions of a regular grammar provide only indirect descriptions of the structure of regular languages. We would like to be able to express a regular language explicitly in terms of simple sets of strings, and we show in this section that this can always be done. Indeed, the set operations union, concatenation, and closure are sufficient to express any regular language in terms of singleton alphabetic symbols.

#### 5.3.1 Regular Expressions

In Chapter 2 we showed informally how the set operations union, concatenation, and closure could be used to form expressions for sets of strings. For example, we might describe a language \( L \) by the expression \( (0 \cup 1)^*11 \). (That is, \( L \) is the collection of strings in \( \{0, 1\}^* \) that end in 11.) We distinguish between the string of symbols making up the expression itself and the set of strings the expression describes. The former is a *regular expression*; the latter is a *regular set*. Two distinct regular expressions may describe the same regular set. For example, the expressions

\[1(01)^* \text{ and } (10)^*1\]

each describe the set containing all strings of alternating 1's and 0's that start and end with a 1.

For the purposes of this section we must be precise about what strings of symbols constitute regular expressions:

**Definition 5.5:** Let \( V \) be a finite alphabet. A *regular expression* on \( V \) is any finite string of symbols from the set

\[\{a | a \in V\} \cup \{\cup, *, (, ), \lambda, \emptyset\}\]

that may be formed according to the following rules:

1. \( \lambda \) is a regular expression.
2. \( \emptyset \) is a regular expression.
3. If \( a \in V \), then \( a \) is a regular expression.
If $\alpha$ and $\beta$ are regular expressions, then the following are regular expressions:

4. $(\alpha \beta)$.
5. $(\alpha \cup \beta)$.
6. $(\alpha^*)$.

A regular expression describes a set according to our usual interpretations of the set operations. The parentheses are generally omitted when it will not cause confusion.

Two regular expressions are equivalent if and only if they describe the same set of strings. The reader should be familiar with some simple equivalences for regular expressions. Beyond the usual properties of set union and concatenation, the most important equivalences concern properties of the closure (Kleene star) operation. These are given below, where $\alpha$, $\beta$, and $\gamma$ stand for arbitrary regular expressions:

1. $(\alpha^*)^* = \alpha^*$.
2. $\alpha \alpha^* = \alpha^* \alpha$.
3. $\alpha \alpha^* \cup \lambda = \alpha^*$.
4. $\alpha (\beta \cup \gamma) = \alpha \beta \cup \alpha \gamma$.
5. $\alpha (\beta \alpha)^* = (\alpha \beta)^* \alpha$.
6. $(\alpha \cup \beta)^* = (\alpha^* \cup \beta^*)^*$.
7. $(\alpha \cup \beta)^* = (\alpha^* \beta^*)^*$.
8. $(\alpha \cup \beta)^* = \alpha^* (\beta \alpha^*)^*$.

The validity of each identity follows directly from the properties of union, concatenation, and closure. Verification is left to the reader.

In general, the distributive law does not hold for the closure operation. For example, the statement $(\alpha \cup \beta)^* = \alpha^* \cup \beta^*$ is false because the right-hand side denotes no string in which both $\alpha$ and $\beta$ appear.

Given a regular expression $\gamma$ that describes a set $X$, it is easy to construct a regular expression $\gamma^R$ that describes the set $X^R$ containing the reverse of each string in $X$:

1. If $\gamma$ is $\lambda$, then $\gamma^R$ is $\gamma$.
2. If $\gamma$ is $\alpha \beta$, then $\gamma^R$ is $(\beta^R \alpha^R)$.
3. If $\gamma$ is $\alpha \cup \beta$, then $\gamma^R$ is $(\alpha^R \cup \beta^R)$.
4. If $\gamma$ is $\alpha^*$, then $\gamma^R$ is $\alpha^{R*}$.

Thus, to construct an expression for the reverse of a regular set, it is only necessary to reverse the order of subexpressions joined by the concatenation
operation. For example, if \( \alpha \) is

\[
01(0^*11 \cup 1^*00)01 = A
\]

then \( \alpha^* \) is

\[
10(110^* \cup 001^*)10 = A^*
\]

Thus the reverse of every regular set is a regular set.

By reversing the order of subexpressions joined by concatenation in the equivalences given above, we obtain additional equivalences for regular expressions. For instance, from

\[
(\alpha \cup \beta)^* = \alpha^*(\beta \alpha^*)^*
\]

we conclude that

\[
(\alpha \cup \beta)^* = (\alpha^* \beta)^* \alpha^*
\]

is also valid.

5.3.2 Reduction and Solution of Set Equation Systems

For each finite-state accepter \( M = (Q, S, P, I, F) \), we may construct a system of right-linear set equations according to Proposition 5.2. If \( M \) has \( n \) states \( q_1, \ldots, q_n \), the set equations are

\[
E_k = \bigcup_{j=1}^n V_{kj} E_j \cup W_k, \quad k = 1, 2, \ldots, n
\]

where

- \( E_k \) is the end set for state \( q_k \)
- \( V_{kj} = \{ s \in S \mid q_k \xrightarrow{s} q_j \text{ in } M \} \)
- \( W_k = \{ \lambda \text{ if } q_k \in F \}
\)

For some choices of \( j \) and \( k \), \( V_{kj} \) may be empty, in which case the term \( V_{kj} E_j \) vanishes.

From Proposition 5.2 we know that the end sets of \( M \) are a solution of the right-linear system. We have yet to establish, however, that these sets are the only such solution. The construction of regular expressions for these solutions and the proof of their uniqueness are the subjects of the following paragraphs. Although in the remainder of this section we deal exclusively with right-linear set equations, our results extend easily to left-linear systems.

The solution of these equation systems resembles the reduction procedure for linear algebraic equations. We manipulate one equation to obtain an expression for an unknown set (say \( E_i \)) in terms of the remaining unknown sets. We substitute this expression for every appearance of \( E_i \) in the remaining equations, thereby obtaining a new system having one less equation and one less unknown than the original system. This reduction step is repeated until we obtain an expression for the last unknown (say \( E_n \)) in terms of the constant sets \( \{V_{ij}\} \) and \( \{W_k\} \) of the equation system.
To be more specific, we shall exhibit one step in the reduction procedure for the right-linear equation system. Choosing to eliminate $E_1$, we rewrite the system with the $E_i$ equation written separately:

$$E_i = V_{11}E_i \cup \left( \bigcup_{j=2}^{n} V_{1j}E_j \right) \cup W_1$$

$$E_k = V_{k1}E_i \cup \left( \bigcup_{j=2}^{n} V_{kj}E_j \right) \cup W_k, \quad k = 2, \ldots, n$$

First we find a solution for the $E_1$ equation. The solution is not obvious, because $E_1$ appears on both sides of the equation. According to the $E_1$ equation, a string $\omega$ is in $E_1$ if and only if

1. $\omega \in \bigcup_{j=2}^{n} V_{1j}E_j \cup W_1$

or

2. $\omega = s\varphi$, where $\varphi \in E_1$ and $s \in V_{11}$

Figure 5.11a shows how these two conditions may be represented by a pseudo state graph in which sets are allowed as labels of transitions. The structure of this graph is shown abstractly in Figure 5.11b, where

$$Q = \bigcup_{j=2}^{n} V_{1j}E_j \cup W_1$$

$$P = V_{11}$$

$$X = E_1$$

Thus the equation has the form

$$X = PX \cup Q$$
where $X$ denotes an unknown end set. From the figure it is evident that

$$X = P^*Q$$

is a solution. This fact has been called Arden's rule, and the uniqueness of this solution will be shown shortly in the proof of Theorem 5.4. Applying Arden's rule to the $E_1$ equation, we obtain

$$E_1 = V_{i1}^* \left( \bigcup_{j=2}^n V_{1j}E_j \cup W_1 \right)$$

Substituting this expression for $E_1$ in each of the remaining $n - 1$ equations, we obtain

$$E_k = \bigcup_{j=2}^n (V_{kj}V_{i1}V_{1j} \cup V_{kj})E_j \cup V_{kj}V_{i1}^*W_1 \cup W_k, \quad k = 2, \ldots, n$$

By assigning

$$V_{kj} = V_{k1}V_{i1}^*V_{1j} \cup V_{kj}$$
$$W_k = V_{k1}V_{i1}^*W_1 \cup W_k$$

we obtain $n - 1$ right-linear equations in $n - 1$ unknowns

$$E_k = \bigcup_{j=2}^n V_{kj}E_j \cup W_k, \quad k = 2, \ldots, n$$

from which $E_1$ has been successfully eliminated.

The reduction step has an instructive interpretation in terms of pseudo state graphs. Figure 5.12a represents the original right-linear equation system where state $q_i$ is representative of states with transitions into state $q_1$, and state $q_j$ is representative of states to which there are transitions from state $q_i$. (States $q_i$ and $q_j$ need not be distinct.) A single final state is included with a transition labeled $W_k$ entering from each node $q_k$ of the graph. Once we have properly accounted for all paths leading to the final state or to state $q_j$ by way of a transition to state $q_1$, state $q_1$ may be eliminated from the graph. Figures 5.12b and c show the transformation of the state graph into a new state graph (without state $q_1$) that represents the reduced system.

The complete solution of the equation system is found through repeated application of the reduction step described above. Let $E_k$ stand for the right linear system of $n - k$ equations obtained by eliminating unknowns $E_1, \ldots, E_k$. Each reduction step transforms system $E_k$ into $E_{k+1}$ through the elimination of $E_{k+1}$. The system $E_{n-1}$ consists of one equation in the remaining unknown set $E_n$, and is solved by Arden's rule. The resulting expression for $E_n$ involves only the operations of union, concatenation, and closure applied to the constant sets (the $V$'s and $W$'s), and is therefore a regular expression. For $k = n - 1, \ldots, 1$, we substitute expressions for $E_{k+1}, \ldots, E_n$ into the $E_k$ equation of system $E_{k-1}$ to obtain a regular expression for the unknown set $E_k$. In this way we obtain the complete solution of the equation system.
Figure 5.12. Transformation of a state graph by an elimination step.

Example 5.12: The right-linear equations for the accepter in Figure 5.13a are

\[ A = 1A \cup 0B \cup \lambda \]
\[ B = 1B \cup 1C \]
\[ C = 1A \cup 0B \cup 1C \]
Since an expression for $L(M) = A$ is desired, we choose to eliminate $C$ first, and then $B$. Solving the $C$ equation by Arden’s rule, we find

$$C = 1^*(1A \cup 0B)$$

Substituting into the $A$ and $B$ equations and regrouping terms yields two equations in two unknowns:

$$A = 1A \cup 0B \cup \lambda$$

$$B = (1 \cup 11^*0)B \cup 11^*1A$$

This system corresponds to the graph in Figure 5.13b in which state $C$ has been eliminated. Applying Arden’s rule to the new $B$ equation, we find

$$B = (1 \cup 11^*0)^*11^*1A$$
Using this in the A equation and regrouping terms, we have

\[ A = (1 \cup 0(1 \cup 11^*0)^*11^*1)A \cup \lambda \]

This equation is represented by the state graph in Figure 5.13c. A final application of Arden's rule yields

\[ L(M) = A = (1 \cup 0(1 \cup 11^*0)^*11^*1)^*\lambda. \]

In applying the reduction procedure, the unknown sets of the equation system may be eliminated in any order, and an intelligent choice will often save effort and lead to more comprehensible regular expressions. Although the expressions obtained will depend on the order of reduction, we shall see that they must always describe the same sets.

### 5.3.3 Uniqueness of Solutions

We wish to show that the reduction procedure developed above yields regular expressions that describe a unique solution to the right-linear equation system of a finite-state accepter. We first show that Arden's rule gives a unique solution of \( X = PX \cup Q \) whenever \( P \) does not contain the empty string; then we apply this result to the reduction procedure.

We have noted that \( X = P*Q \) is a solution of the set equation \( X = PX \cup Q \), where \( P \) and \( Q \) are any subsets of \( V^* \), and this may be verified by substituting \( P*Q \) for \( X \) on the right side:

\[
\begin{align*}
X &= PX \cup Q \\
    &= PP*Q \cup Q \\
    &= (PP* \cup \lambda)Q \\
    &= P*Q
\end{align*}
\]

We have not shown that \( P*Q \) is the only such solution, however, and in general it is not. In particular, if \( P \) contains the empty string, then \( V^* \) is a solution of the equation regardless of the choice of \( P \) and \( Q \). In Theorem 5.4 we establish the uniqueness of the solution \( P*Q \) whenever \( \lambda \) is not in \( P \).

**Theorem 5.4:** Let \( P \) and \( Q \) be arbitrary sets of strings on a finite alphabet \( V \). Then

\[ X = P*Q \text{ is a solution of } X = PX \cup Q \]

and is a unique solution whenever \( \lambda \notin P \).

**Proof:** We have already shown that \( P*Q \) is a solution of the equation. We now show that if \( \lambda \notin P \) only one set can satisfy the equation.
Suppose that $X_1$ and $X_2$ are distinct solutions to $X = PX \cup Q$, where $P$ does not contain $\lambda$:

$$X_1 = PX_1 \cup Q \quad X_1 \neq X_2, \lambda \notin P$$
$$X_2 = PX_2 \cup Q$$

Then $X_0 = X_1 \cup X_2$ is also a solution:

$$P(X_1 \cup X_2) \cup Q = PX_1 \cup PX_2 \cup Q$$
$$= (PX_1 \cup Q) \cup (PX_2 \cup Q)$$
$$= X_1 \cup X_2$$

Since $X_1$ and $X_2$ are distinct, one of $X_1$ and $X_2$ must be properly contained in $X_0$. Suppose that $X_1 \subset X_0$, and let $A$ be the difference

$$A = X_0 - X_1$$

Then

$$A \neq \emptyset \quad X_0 = X_1 \cup A \quad A \cap X_1 = \emptyset$$

Since $X_0$ is a solution, we have

$$(X_1 \cup A) = P(X_1 \cup A) \cup Q$$
$$= PA \cup (PX_1 \cup Q)$$
$$= PA \cup X_1$$

Intersecting both sides of the last equation with $A$ gives

$$(A \cap X_1) \cup (A \cap A) = (A \cap PA) \cup (A \cap X_1)$$

Since $A \cap X_1 = \emptyset$, this becomes

(1) \hspace{1cm} A = A \cap PA

Since $A \neq \emptyset$, there exists a shortest string $\omega_0$ in $A$. By equation (1), $\omega_0 \in A \cap PA$, and therefore $\omega_0 \in PA$. It follows that

$$\omega_0 = \alpha \beta, \quad \text{where} \quad \alpha \in P \quad \text{and} \quad \beta \in A$$

Since $\lambda \notin P$ by assumption, $|\alpha| \geq 1$ and therefore $|\beta| < |\omega_0|$. Since $\beta \in A$, this contradicts the statement that no string in $A$ is shorter than $\omega_0$. Thus the assumption that $X_1 \neq X_2$ leads to a contradiction, and we conclude that all sets that satisfy $X = PX \cup Q$ must be identical. Since $P^*Q$ is a solution, it must be unique.

We now apply Theorem 5.4 to show that the reduction procedure developed in the previous paragraphs produces a unique solution to the system of set equations derived from any finite-state accepter, and that this solution consists of the end sets of the accepter.
Proposition 5.3: Let $M = (Q, S, P, I, F)$ be an FSA with states $Q = \{q_1, \ldots, q_n\}$ and let

\[ V_{ij} = \{s \in S | q_i \xrightarrow{a} q_j \text{ in } M\} \]

\[ W_k = \begin{cases} \lambda & \text{if } q_k \in F \\ \varnothing & \text{otherwise} \end{cases} \]

Then the right-linear system

\[ E_k = \bigcup_{j=1}^{n} V_{kj}E_j \cup W_k, \quad k = 1, 2, \ldots, n \]

has a unique solution such that

\[ L(M) = \bigcup_{q \in I} E_k \]

where $E_1, \ldots, E_n$ are the end sets of $M$.

Proof:

Existence: The end sets are defined by

\[ E_k = \{\omega \in S^* | q_k \xrightarrow{\omega} q \text{ for some } q \in F\} \]

From Proposition 5.2, the end sets $E_1, \ldots, E_n$ of $M$ are a solution of the equation system.

Uniqueness: Suppose that $E_1, \ldots, E_n$ are arbitrary subsets of $S^*$ that form a solution of the equation system. We shall use Theorem 5.4 to show that each set $E_k$ is unique.

None of the sets $V_{ij}$ in the equation system contains $\lambda$. In particular, $\lambda \notin V_{11}$ in the $E_1$ equation. Therefore, by Theorem 5.4,

\[ E_1 = V_{11}^* \left( \bigcup_{j=2}^{n} V_{j1}E_j \cup W_1 \right) \]

must be a unique solution of the $E_1$ equation. Substituting this expression for $E_1$ into the other equations of the system produces a new system that must be satisfied by $E_2, \ldots, E_n$. Again, the coefficients of the sets $E_2, \ldots, E_n$ in the system do not contain $\lambda$. Repeating this procedure results, after $n$ steps, in an expression for $E_n$ in terms of constant sets. From Theorem 5.4, the solutions for the unknowns at each step are unique, and thus the fixed set denoted by this expression is the only possible value of $E_n$ in any solution to the equation system.

Since the numbering of the states of $M$ is immaterial, this uniqueness argument applies separately to each set $E_k$, $k = 1, \ldots, n$. Therefore, the complete solution of the equation system is unique.

Applying the elimination procedure to a right-linear system results in a regular expression for each end set. Since $L(M)$ is the union of some of the
end sets, a regular expression for \( L(M) \) is easily obtained. We have therefore established the following theorem, which is half of Kleene's important result about finite-state machines.

**Theorem 5.5**: Each finite-state accepter recognizes a language that can be described by some regular expression.

The other half of Kleene's result is the converse of Theorem 5.5: each regular expression describes a set recognized by some finite-state accepter. This is the subject of the next section.

**5.3.4 Accepters for Regular Sets**

We wish to show how a finite-state accepter can be constructed for the set of strings described by an arbitrary regular expression \( \alpha \). The synthesis is carried out recursively by combining accepters for subexpressions of \( \alpha \). We start with primitive accepters for

1. The empty set \( \emptyset \).
2. The set \( \lambda \).
3. The set \( a \), for each symbol \( a \) in the appropriate alphabet \( V \).

Then, given that accepters \( M_1 \) and \( M_2 \) recognize arbitrary regular sets \( R_1 \) and \( R_2 \), we show how \( M_1 \) and \( M_2 \) can be combined to create an accepter for each of the sets

\[
R_1 \cup R_2 \quad R_1 \cdot R_2 \quad R_1^*
\]

For the construction, it will be convenient to assume that each accepter has exactly one initial state with no entering transitions, and exactly one accepting state with no exiting transitions. Example 5.13 demonstrates the difficulties that arise in the construction if this is not the case.

**Example 5.13**: In Figure 5.14a, accepter \( M_1 \) recognizes the set \( R_1 = 01^* \), and accepter \( M_2 \) recognizes the set \( R_2 = 0^*10 \). To obtain an accepter for \( R_1 \cup R_2 \), it would be convenient to merge the two initial states and to merge the two final states, as in Figure 5.14b. It is obvious from the diagram that this new machine recognizes \( 0^*10^* \cup 0^*10^* \), rather than \( 0^*10^* \cup 0^*10^* \) as intended. Also, if we attempt to find an accepter for \( R_1 \cdot R_2 \) by merging the final state of \( M_1 \) with the initial state of \( M_2 \), we obtain instead the accepter for \( 0(0 \cup 1)^*10 \) shown in Figure 5.14c.

In either case, the combined machine recognizes a larger set of strings than desired.
The difficulties exposed by Example 5.13 are due to the transition leaving the final state of $M_1$ and the transition entering the initial state of $M_2$. These transitions create undesired paths when the state diagrams are combined. If we prohibit such transitions in the accepters used in the construction, complications arise in the case of machines that accept the empty string. A better approach is to use accepters in which $\lambda$-transitions are permitted. We digress to study accepters with $\lambda$-transitions, and return to the construction of finite-state accepters from regular expressions in Section 5.3.6.

5.3.5 Accepters with Lambda Transitions

A lambda transition

\[ q \xrightarrow{\lambda} q' \]
in an accepter \( M \) signifies that \( M \) may leave state \( q \) and enter state \( q' \) without the presentation of an input symbol. In an accepter with \( \lambda \)-transitions, an admissible state sequence for a string may include arbitrarily many such transitions. For example, the machine in Figure 5.15 accepts the string \( acb \) via the state sequence

\[
A \xrightarrow{\lambda} B \xrightarrow{a} C \xrightarrow{c} C \xrightarrow{\lambda} B \xrightarrow{b} C \xrightarrow{\lambda} D
\]

![Figure 5.15. Acceptor with \( \lambda \)-transitions.]

We define a class of accepters in which \( \lambda \)-transitions are used to "isolate" single initial and final states.

**Definition 5.6:** A \( \lambda \)-accepter is a finite-state accepter of the form \( M = (Q, S, P, q_I, q_F) \) with \( \lambda \)-transitions such that

1. \( M \) has exactly one initial state, \( q_I \), with no transitions entering and only \( \lambda \)-transitions exiting.
2. \( M \) has exactly one final state, \( q_F \), with no transitions exiting and only \( \lambda \)-transitions entering.

Note that conditions 1 and 2 imply \( q_I \neq q_F \).

We must show that the class of \( \lambda \)-accepters is equivalent to the class of finite-state accepters. To this end, we shall show how to convert any finite-state accepter to a \( \lambda \)-accepter, and vice versa.

The conversion of a finite-state accepter to a \( \lambda \)-accepter, illustrated in Figure 5.16, is straightforward:

1. Let \( M = (Q, S, P, I, F) \) be an arbitrary FSA. We wish to construct a \( \lambda \)-accepter \( M' = (Q', S, P', q_I, q_F) \) such that \( L(M') = L(M) \). Let

\[
Q' = Q \cup \{q_F, I\}
\]

2. The transitions of \( M' \) include all the transitions of \( M \) and the new \( \lambda \)-transitions

\[
q_I \xrightarrow{\lambda} q \quad \text{for each } q \in I
\]

\[
q' \xrightarrow{\lambda} q_F \quad \text{for each } q' \in F
\]
The reader should convince himself that the construction does not alter the language recognized by the accepter.

We must now show that any $\lambda$-accepter $M$ can be changed into a finite-state accepter $M'$ such that $L(M') = L(M)$. We shall do this in two steps.

First, suppose that $M$ has a loop of $\lambda$-transitions

$$q_0 \xrightarrow{\lambda} q_1 \xrightarrow{\lambda} \ldots \xrightarrow{\lambda} q_n \xrightarrow{\lambda} q_0$$

in which $q_0, q_1, \ldots, q_n$ are all distinct states. (Note that by Definition 5.6 no such loop will ever contain the initial state or the final state of the accepter.) Then the end sets of $q_0, q_1, \ldots, q_n$ are identical, because if a state sequence for a string leads from any state of the loop to the final state, then a state sequence for that string leads from every state of the loop to the final state. Thus the states in any $\lambda$-loop may be merged into a single state, and this is the first step in the construction of an FSA from a $\lambda$-accepter:

**Step 1:** Suppose that the $\lambda$-accepter $M = (Q, S, P, q_i, q_f)$ contains a loop of $\lambda$-transitions. Let $Q_\lambda \subseteq Q$ be the states in the loop. Construct the $\lambda$-accepter $M' = (Q', S, P', q_i, q_f)$ without the loop as follows:

1. $Q' = (Q - Q_\lambda) \cup \{q_i\}$.
2. If $M$ has the transition

   $$q_1 \xrightarrow{s} q_2, \quad s \in (S \cup \{\lambda\})$$

   then $M'$ has a transition

   $$q'_1 \xrightarrow{s} q'_2$$
where
\[ q'_1 = \begin{cases} q_t & \text{if } q_1 \in Q_t \\ q_1 & \text{otherwise} \end{cases} \quad q'_2 = \begin{cases} q_t & \text{if } q_2 \in Q_t \\ q_2 & \text{otherwise} \end{cases} \]

Repeat the construction until no \( \lambda \)-loops remain.

We note that, if \( M \) is an accepter without \( \lambda \)-loops, a state sequence consisting only of \( \lambda \)-transitions cannot have repeated appearances of states. For each state \( q \), the paths into \( q \) consisting only of \( \lambda \)-transitions constitute a loop-free \( \lambda \)-subgraph of \( M \) as suggested in Figure 5.17a.

![Diagram of \( \lambda \)-subgraph of state \( q \)]

(b)

![Diagram of removal of \( \lambda \)-transitions]

\textbf{Figure 5.17. Removal of \( \lambda \)-transitions.}
A transition

\[ q \xrightarrow{\lambda} q' \]

in \( M \) implies

\[ q'' \xrightarrow{\lambda} q' \]

for any state \( q'' \) in the \( \lambda \)-subgraph of \( q \). Adding the transition

\[ q'' \xrightarrow{\lambda} q' \]

to \( M \) provides a similar path in the accepter that does not contain \( \lambda \)-transitions. This motivates step 2 of the construction:

\[ \text{Figure 5.18. Machines for Example 5.14.} \]
Sec. 5.3  Regular Expressions and Finite-State Accepters

Step 2: Let $M = (Q, S, P, q_0, F)$ be a $\lambda$-accepter with no $\lambda$-loops. Construct an FSA $M' = (Q', S, P', q'_0, F')$ as follows:

1. $Q' = Q - \{q_F\}$.
2. $F' = \{q \in Q | q \xrightarrow{\lambda} q_F \text{ in } M\}$.
3. The transitions of $M'$ are
   a. Each transition $q \xrightarrow{s} q'$ in $M$ where $s \in S$.
   b. The transition $q'' \xrightarrow{\lambda} q \xrightarrow{s} q'$ whenever $M$ has a state sequence $q'' \xrightarrow{\lambda} q \xrightarrow{s} q'$.

We leave it to the reader to verify that the final $\lambda$-free machine $M'$ recognizes the same language as the $\lambda$-accepter $M$.

The result of applying step 2 to the subgraph of Figure 5.17a is illustrated by Figure 5.17b.

Example 5.14: Figure 5.18a shows a $\lambda$-accepter with a $\lambda$-loop. Figure 5.18b shows the machine after step 1 of the construction, in which states A, B, and C have been merged to eliminate the loop. The machine obtained after step 2 of the construction is shown in Figure 5.18c.

5.3.6 Constructing an Acceptor from a Regular Expression

The construction of $\lambda$-accepters for $R_1 \cup R_2$, $R_1 \cdot R_2$, and $R^\dagger$ from $\lambda$-accepters for $R_1$ and $R_2$ is straightforward, as shown in Figure 5.19. A $\lambda$-accepter for $R_1 \cup R_2$ is obtained by merging the initial states $q_{1i}$ and $q_{2i}$ and the accepting states $q_{1f}$ and $q_{2f}$. A $\lambda$-accepter for $R_1 \cdot R_2$ is obtained by merging the accepting state $q_{1f}$ of $M_1$ with the initial state $q_{2i}$ of $M_2$. The initial and final states of the composite accepter are $q_{1i}$ and $q_{2f}$, respectively. A $\lambda$-accepter for $R^\dagger$ is obtained by merging states $q_{1i}$ and $q_{2i}$ of $M_1$, and then adding new initial and accepting states $q_I$ and $q_F$. (These new states are needed so that the resulting accepter is a $\lambda$-accepter.) The primitive elements for the construction are the $\lambda$-accepters shown in Figure 5.20 for the one-symbol regular expressions.

Theorem 5.6: For any regular expression $\alpha$ on an alphabet $V$, one can construct a finite-state accepter $M$ such that $L(M) = R$, where $R$ is the set described by $\alpha$.

Proof: Without loss of generality, we may construct $M$ as a $\lambda$-accepter. The proof is an induction on the length $|\alpha|$ of the regular expression.

Basis: If $|\alpha| = 1$, then $\alpha$ is either $\emptyset$, $\lambda$, or a symbol in $V$, and $M$ is one of the $\lambda$-accepters shown in Figure 5.20.
(a) $M_1$:

\[ q_{l1} \xrightarrow{\lambda} Q_1 \xrightarrow{\lambda} q_{F1} \]

$L(M_1) = R_1$

$M_2$:

\[ q_{l2} \xrightarrow{\lambda} Q_2 \xrightarrow{\lambda} q_{F2} \]

$L(M_2) = R_2$

(b) Accepter for $R_1 \cup R_2$:

\[ q_{l1}, q_{l2} \xrightarrow{\lambda} Q_1 \xrightarrow{\lambda} Q_2 \xrightarrow{\lambda} q_{F1}, q_{F2} \]

(c) Accepter for $R_1 \cdot R_2$:

\[ q_{l1} \xrightarrow{\lambda} Q_1 \xrightarrow{\lambda} q_{F1}, q_{l2} \xrightarrow{\lambda} Q_2 \xrightarrow{\lambda} q_{F2} \]

(d) Accepter for $R_1^*_{\lambda}$:

\[ q_I \xrightarrow{\lambda} Q_1 \xrightarrow{\lambda} q_{l1}, q_{F1} \xrightarrow{\lambda} q_F \]

Figure 5.19. Recursive construction of $\lambda$-accepters.
Figure 5.20. Lambda-accepters for regular expressions of length 1.

**Induction:** Assume that whenever $1 \leq |\alpha| < k$ the theorem is true, and let $\alpha$ be any expression such that $|\alpha| = k$. Then one of

1. $\alpha = (\alpha_1 \cup \alpha_2)$.
2. $\alpha = (\alpha_1 \alpha_2)$.
3. $\alpha = \alpha_1^*$.

must hold by Definition 5.5, where $|\alpha_1| < k$ and $|\alpha_2| < k$. By the induction hypothesis, there exist $\lambda$-accepters $M_1$ and $M_2$ such that $L(M_1)$ is described by $\alpha_1$ and $L(M_2)$ is described by $\alpha_2$. We treat the three cases separately.

1. Let $M$ be constructed as shown in Figure 5.19a. Then

   \[ q_1 \xrightarrow{\omega} q_f \]

   in $M$ if and only if

   \[ q_{f1} \xrightarrow{\alpha} q_{f1} \quad \text{or} \quad q_{f2} \xrightarrow{\omega} q_{f2} \]

   in $M_1$ or in $M_2$, respectively. Hence

   \[ L(M) = L(M_1) \cup L(M_2) \]

   and is described by $\alpha = (\alpha_1 \cup \alpha_2)$. 
2. Let $M$ be constructed as shown in Figure 5.19b. Then

$$q_i \xrightarrow{\omega} q_f$$

in $M$ if and only if $\omega = \varphi \cdot \psi$ such that

$$q_{i_1} \xrightarrow{\varphi} q_{F_1} \quad \text{and} \quad q_{i_2} \xrightarrow{\psi} q_{F_2}$$

in $M_1$ and in $M_2$, respectively. Hence

$$L(M) = L(M_1) \cdot L(M_2)$$

and is described by $\alpha = \alpha_1 \alpha_2$.

3. Let $M$ be constructed as shown in Figure 5.19c. Then

$$q_i \xrightarrow{\omega} q_f$$

in $M$ if and only if $\omega = \lambda$, or $\omega = \varphi_1 \varphi_2 \ldots \varphi_n$ for some $n \geq 1$, and

$$q_{i_1} \xrightarrow{\varphi_i} q_{F_1}, \quad i = 1, 2, \ldots, n$$

in $M_1$. Therefore,

$$L(M) = L(M_1)^*$$

and is described by $\alpha = \alpha_1^*$.

Theorems 5.5 and 5.6 together constitute Kleene's theorem: a language is described by a regular expression if and only if it is recognized by some finite-state accepter.

**Example 5.15:** The construction of a finite-state accepter for the regular set $((b^*a \cup ab^*)c)^*$ is shown in Figure 5.21. In the primitive accepters for $b^*a$, $ab^*$, and $c$, $\lambda$-transitions have been included only where required to isolate an initial or final state. All $\lambda$-transitions introduced in combining the state diagrams have been retained.

**Example 5.16:** To show how a loop of $\lambda$-transitions may arise, consider constructing a state diagram for the regular expression

$$\alpha = ((ab)^*(ac)^* \cup a)^*$$

The construction of a $\lambda$-accepter from $\alpha$ is shown in Figure 5.22. Collapsing the $\lambda$-loop gives the machine $M_\lambda$, and eliminating the remaining $\lambda$-transitions yields $M_\kappa$. Acceptor $M_\kappa$ shows that the expression $\alpha$ is equivalent to

$$\beta = (ab \cup ac \cup a)^*$$
Sec. 5.3  
Regular Expressions and Finite-State Accepters  

\[ M_1 : \quad \lambda \xrightarrow{a} \lambda \]
\[ L_1 = b^*a \]

\[ M_2 : \quad \lambda \xrightarrow{b} \lambda \]
\[ L_2 = ab^* \]

\[ M_3 : \quad \lambda \xrightarrow{c} \lambda \]
\[ L_3 = c \]

\[ M_4 : \quad \lambda \xrightarrow{a} \lambda \xrightarrow{c} \lambda \xrightarrow{b} \lambda \]
\[ L_4 = (b^*a \cup ab^*)c \]

\[ M_5 : \quad \lambda \xrightarrow{a} \lambda \xrightarrow{\lambda} \lambda \xrightarrow{b} \lambda \]
\[ L_5 = ((b^*a \cup ab^*)c)^* \]

Figure 5.21. Construction of an accepter for a regular set.

Example 5.16 suggests that the synthesis and analysis of state diagrams according to Kleene's theorem is a useful way of demonstrating the equivalence of regular expressions.
Figure 5.22. Construction that creates a λ-loop.

**Example 5.17:** Figure 5.23 shows the steps involved in proving the equivalence

\[ \alpha^*(\beta\alpha^*)^* = (\alpha \cup \beta)^* \]

by means of state diagrams and Kleene's theorem. In Figure 5.23d, the two states are equivalent; hence this state diagram reduces to Figure 5.23e.
5.4 Properties of Finite-State Languages

The developments in this chapter have established the equivalence of three representations for languages. If $L$ is a language on some alphabet $V$, these three statements are equivalent:

1. $L$ is generated by some regular grammar.
2. $L$ is recognized by some finite-state accepter.
3. $L$ is described by some regular expression.

Thus we may use the terms *regular set*, *regular language*, and *finite-state language* interchangeably.

If we wish to prove an assertion about finite-state languages, three approaches are always available: we may prove the assertion in terms of (1) the languages generated by right- or left-linear grammars, (2) the sets recognized by finite-state accepters, or (3) the languages described by regular expressions. Frequently, arguments expressed in terms of finite-state machines are the most intuitive and the easiest to understand.
5.4.1 Closure Properties

A class of languages \( \mathcal{C} \) is closed under a unary operation

\[
F: \mathcal{C} \rightarrow \mathcal{C}
\]

if \( F(L) \) is a member of \( \mathcal{C} \) whenever \( L \) is a member of \( \mathcal{C} \); it is closed under a binary operation

\[
G: \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}
\]

if \( G(L_1, L_2) \) is a member of \( \mathcal{C} \) whenever \( L_1 \) and \( L_2 \) are members of \( \mathcal{C} \).

Since a finite-state accepter can be constructed for the union or concatenation of any two regular sets, or for the closure of a regular set, the class of finite-state languages is closed under the operations of union, concatenation, and closure. Are they also closed under the set operation of complementation? of intersection? Elementary arguments provide answers to both questions.

Given an accepter \( M = (Q, S, P, I, F) \) for the set \( R \), we may construct an accepter \( M' \) for the set \( R^c = S^* - R \) as follows:

1. Convert \( M \) into a deterministic accepter \( M'' = (Q'', S, P'', I'', F'') \).
2. Interchange the roles of accepting and nonaccepting states in \( M'' \) to obtain \( M' \); that is \( M' = (Q'', S, P'', I'', Q'' - F'') \).

It is clear that \( L(M') = L(M)^c \). (Why is it necessary to perform step 1?) Thus the complement of a regular set is always a regular set.

That the intersection of regular sets is regular follows from the De Morgan law

\[
(1) \quad R_1 \cap R_2 = (R_1^c \cup R_2^c)^c
\]

and the closure of regular sets under complementation and union. Given accepters for sets \( R_1 \) and \( R_2 \), we may construct accepters for sets \( R_1^c \) and \( R_2^c \) as described above, and combine their state diagrams to obtain an accepter \( M \) for \( R_1^c \cup R_2^c \). We may then construct an accepter \( M' \) for \( (L(M))^c \), again as described above. From (1), \( L(M') = R_1 \cap R_2 \), and thus the finite-state languages are closed under intersection.

It is also instructive to obtain these results directly through the parallel combination of accepters. Let us regard the two accepters \( M_1 \) and \( M_2 \) in Figure 5.24 as being physically combined to form a machine \( M \). Because we wish to regard \( M_1 \) and \( M_2 \) as operating independently but on the same input string, they must be deterministic accepters:

\[
M_1 = (Q_1, S, P_1, q_{r1}, F_1)
\]
\[
M_2 = (Q_2, S, P_2, q_{r2}, F_2)
\]
Suppose that the combined machine $\mathbf{M}$ accepts strings that leave $\mathbf{M}_1$ and $\mathbf{M}_2$ in combinations of states for which a given function

$$f : Q_1 \times Q_2 \rightarrow \{0, 1\}$$

has the value 1. The construction of $\mathbf{M}$ is as follows:

1. $Q = Q_1 \times Q_2$.
2. $q_0 = (q_{10}, q_{20})$.
3. $F = \{(q_1, q_2) \mid f(q_1, q_2) = 1\}$.
4. $P = \{(q_1, q_2) \xrightarrow{\delta} (q'_1, q'_2) \mid \begin{array}{l} q_1 \xrightarrow{\delta} q'_1 \text{ is in } P_1 \end{array}$
   \hspace{1cm} \begin{array}{l} q_2 \xrightarrow{\delta} q'_2 \text{ is in } P_2 \end{array}\}$.

As usual,

$$L(\mathbf{M}) = \{\omega \in S^* \mid q_1 \xrightarrow{\omega} q \text{ and } q \in F\}$$

Particular choices of the function $f$ produce machines for the union, intersection, difference, and complement of the languages $L(\mathbf{M}_1)$ and $L(\mathbf{M}_2)$:

1. Union: let $f(q_1, q_2) = 1$ if and only if $(q_1 \in F_1) \lor (q_2 \in F_2)$. Then
   $$L(\mathbf{M}) = L(\mathbf{M}_1) \cup L(\mathbf{M}_2).$$
2. Intersection: let $f(q_1, q_2) = 1$ if and only if $(q_1 \in F_1) \land (q_2 \in F_2)$. Then
   $$L(\mathbf{M}) = L(\mathbf{M}_1) \cap L(\mathbf{M}_2).$$
3. Difference: let $f(q_1, q_2) = 1$ if and only if $(q_1 \in F_1) \land (q_2 \in Q_2 - F_2)$. Then
   $$L(\mathbf{M}) = L(\mathbf{M}_1) - L(\mathbf{M}_2).$$
4. Complement: let $f(q_1, q_2) = 1$ if and only if $(q_1 \in Q_1 - F_1)$. Then
   $$L(\mathbf{M}) = L(\mathbf{M}_1)^c.$$
In the last two cases, the construction breaks down if either $M_1$ or $M_2$ is nondeterministic; the reader should be sure to understand just where the construction fails.

**Example 5.18:** Figure 5.25 shows how two simple accepters for languages $L_1$ and $L_2$ may be used to derive accepters for $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 - L_2$, according to the choice of accepting states in the state diagram for the parallel combination of the two accepters.

We now know that the class of finite-state languages is closed under the operations of intersection, difference, and complementation, as well as union, concatenation, and closure. Since every finite-state language is also a regular set, the class of finite-state languages is closed under reversal by the arguments of Section 5.3.1. We summarize these facts as a theorem.

**Theorem 5.7:** The class of finite-state languages is closed under the operations of set union, intersection, complementation, difference, concatenation, closure, and reversal. That is, if $L_1$ and $L_2$ are finite-state languages, then so are the following:

1. $L_1 \cup L_2$.
2. $L_1 \cap L_2$.
3. $L_1^c$.
4. $L_1 - L_2$.
5. $L_1 \cdot L_2$.
6. $L_1^*$.
7. $L_1^p$.

It is important to understand the distinction between combining two machines in the manner just described and combining state diagrams as in the proof of Kleene’s theorem. The manner of this section amounts to setting two physical, deterministic machines side by side and considering them as one; in the proof of Kleene’s theorem, we consider two state diagrams as jointly representing the behavior of one machine. Combining state diagrams in the latter case has no convenient interpretation in terms of physical machines. In particular, the new state set is $Q_1 \times Q_2$ when physical machines are combined, but $Q_1 \cup Q_2$ when state diagrams are combined. For this reason, the machines that result from these methods are known as *product machines* and *sum machines*, respectively.

There is no simple way to directly combine nondeterministic accepters to obtain accepters for the intersection or difference of two finite-state languages. This is indicative of the difficulties we shall encounter in trying to apply these operations to context-free languages.
Figure 5.25. Machines for Example 5.18.
5.4.2 Ambiguity

With our present knowledge of finite-state accepters, we can establish several important results concerning ambiguous regular grammars. Let $G$ be a right-linear grammar, and let $M = (Q, S, P, I, F)$ be the finite-state accepter constructed from $G$ according to Table 5.2. A sentence

$$\omega = s_1s_2 \ldots s_k, \quad s_i \in S$$

will have an admissible state sequence

$$q_0 \xrightarrow{s_1} q_1 \xrightarrow{s_2} \ldots \xrightarrow{s_k} q_k, \quad q_0 \in I, q_k \in F$$

in $M$ if and only if the derivation

$$\Sigma \xrightarrow{N(q_0)} s_1N(q_1) \xrightarrow{.} \ldots \xrightarrow{.} s_1 \ldots s_{k-1}N(q_{k-1}) \xrightarrow{.} s_1 \ldots s_{k-1}s_k$$

is permitted in $G$. If $G$ is ambiguous, then some string $\omega$ will have two (or more) distinct derivations in $G$. Since only one nonterminal ever appears in each sentential form, these derivations must replace distinct sequences of nonterminal letters. Thus the corresponding state sequences in $M$ will also be distinct. This can only be the case if $M$ is nondeterministic.

To decide whether $G$ is ambiguous, we must determine whether there is a pair of states $q$ and $q'$ in the corresponding accepter $M$ such that for some string $\omega = \varphi \psi$ the behavior shown in the following diagram is possible:

```
\[ \begin{array}{ccc}
q & \xrightarrow{\varphi} & q' \\
\varphi & \xrightarrow{I} & \psi \\
q' & \xrightarrow{\psi} & \varphi \\
\end{array} \]
```

Recall the definition of the states $X_{[\varphi]}$ reachable for the string $\varphi$:

$$X_{[\varphi]} = \{q' \mid q \xrightarrow{\varphi} q' \text{ for some } q \in I\}$$

Let us call the states $Y_{[\psi]}$ from which a final state can be reached in response to $\psi$ the leavable states for the string $\psi$:

$$Y_{[\psi]} = \{q' \mid q \xrightarrow{\psi} q' \text{ for some } q' \in F\}$$

From the above diagram we see that $M$ has distinct admissible state sequences for a string $\omega$ if and only if $\omega = \varphi \cdot \psi$, where $X_{[\varphi]}$ and $Y_{[\psi]}$ have at least two states in common. This fact yields a procedure for determining whether an arbitrary right-linear grammar $G$ is ambiguous and, if so, for finding multiple derivations of some string in $L(G)$:
1. Let $M$ be the FSA constructed from $G$ using Table 5.2.
2. Construct the tree diagram of reachable sets for $M$ starting from $X_{[A]} = I$.
3. Construct the tree diagram of leavable sets for $M$ starting from $Y_{[A]} = F$.
4. The grammar $G$ is ambiguous if and only if some set of two or more states appears in both tree diagrams.
5. Each string that labels a path from $I$ to $F$, and is incident on the set found in step 4, has ambiguous derivations in $G$.

This procedure will always terminate because the tree diagrams can have no more than $2^n$ nodes for an $n$-state accepter $M$, and only a finite number of comparisons of sets will be required.

**Theorem 5.8:** There is a finite procedure for deciding whether an arbitrary right-linear grammar is ambiguous, and, if so, for finding an ambiguous sentence generated by the grammar.

**Example 5.19:** The right-linear grammar

\[
\begin{align*}
\Sigma & \rightarrow A & A & \rightarrow 1B & C & \rightarrow 0B \\
& & B & \rightarrow 0B & C & \rightarrow 1C \\
& & B & \rightarrow 0C & C & \rightarrow 1
\end{align*}
\]

corresponds by Table 5.2 to the accepter in Figure 5.26. The tree of reachable sets and the tree of leavable sets are also shown in the figure. Since the two-element set $\{B, C\}$ appears in both trees, we conclude that the grammar is ambiguous. In particular, the state sequences

\[
\begin{align*}
A & \rightarrow 1 & B & \rightarrow 0 & B & \rightarrow 0 & B & \rightarrow 0 & C & \rightarrow 1 & D \\
A & \rightarrow 1 & B & \rightarrow 0 & C & \rightarrow 0 & B & \rightarrow 0 & C & \rightarrow 1 & D
\end{align*}
\]

correspond to distinct derivations of the string 10001.

A theorem analogous to Theorem 5.8 can be obtained for left-linear grammars, using techniques similar to those of Example 5.11. The details are left as an exercise.

If a regular grammar is ambiguous, we can always find an equivalent grammar that is unambiguous.

**Theorem 5.9:** For any regular grammar $G$, it is possible to construct an unambiguous grammar $G'$ such that $L(G') = L(G)$.

**Proof:** We use Table 5.2 to construct an FSA $M$ such that $L(M) = L(G)$. We then convert $M$ to an equivalent deterministic accepter
M'. Using Table 5.1, we construct a grammar G' from M' such that L(G') = L(M'). Since M' is deterministic, G' must be unambiguous.

Theorems 5.8 and 5.9 are of particular interest because their generalizations to wider classes of grammars are not valid. We shall see in Chapter 12, for example, that there is no general procedure to determine whether a given context-free grammar is ambiguous, or to find an ambiguous string...
even if we know the grammar is ambiguous, or to construct an equivalent
unambiguous grammar.

5.4.3 Decision Problems

Given a class of objects \( C \), a predicate

\[
p: \quad C \rightarrow \{true, false\}
\]

is said to be *decidable* if there is a terminating step-by-step procedure for
deciding whether \( p(x) \) is true or false for any \( x \) in \( C \). A procedure that pro-
duces the correct answer for each member of \( C \) is an *algorithm* or *effective
procedure* for the predicate \( p \). The concept of an effective procedure will be
studied formally in later chapters of this book; for the present, the informal
definition just given will suffice.

Most of the reasonable questions posed about finite-state languages are
decidable. From our present knowledge of finite-state machines, we can
easily devise algorithms for answering some of these questions.

**Theorem 5.10:** Let \( L_1 \) and \( L_2 \) be arbitrary finite-state languages,
and let \( G \) be an arbitrary regular grammar. Then it is decidable
whether

1. \( L_1 = L_2 \).
2. \( L_1 = \emptyset \).
3. \( L_1 \) is finite; \( L_1 \) is infinite.
4. \( L_1 \cap L_2 = \emptyset \).
5. \( L_1 \subseteq L_2 \).
6. \( G \) is ambiguous.

**Proof:** We outline the steps by which each question may be answered.

1. Construct accepters for \( L_1 \) and \( L_2 \), convert them to
deterministic accepters, and test for equivalence.

2. Let \( M_1 \) be an accepter for \( L_1 \). Then \( L_1 = \emptyset \) if and only if
there is no path in \( M_1 \) from any initial state to any final state. This
is always possible to determine, because in an \( n \)-state accepter it is
only necessary to consider paths of length less than \( n \).

3. Let \( M_1 \) be an \( n \)-state accepter for \( L_1 \). The reader may verify
that \( L(M_1) \) is infinite if and only if \( M_1 \) accepts some string \( \omega \) for
which \( |\omega| \geq n \).

4. Construct an accepter \( M \) for \( L_1 \cap L_2 \) (Theorem 5.7). Then
\( L_1 \cap L_2 = \emptyset \) if and only if \( L(M) = \emptyset \), which is decidable by (2).
5. \( L_1 \subseteq L_2 \) if and only if \( L_1 \cap L_2 = \emptyset \). We need only construct an accepter \( M \) for \( L_1 \cap L_2 \) and test whether \( L(M) = \emptyset \).

6. Theorem 5.8.

In subsequent chapters we shall see that many important questions that are decidable for finite-state languages are undecidable for less restricted classes of languages.

Notes and References

Nondeterministic finite-state machines were proposed by M. O. Rabin and D. Scott [1959], who demonstrated that this generalization did not increase the capabilities of the machines. Although we have drawn heavily on the properties of nondeterministic machines in this chapter, our major results (summarized in the first paragraph of Section 5.4) were established prior to 1959 in terms of deterministic accepters.

The equivalence of the class of languages accepted by finite-state accepters and that generated by type 3 grammars was demonstrated by Chomsky and Miller [1958]. The relation between these languages and those described by regular expressions was discovered by Kleene [1956]. (His regular expressions differ slightly from ours, and his finite-state accepters were the “nerve nets” of McCulloch and Pitts [1943]; see also Minksy [1967].) McNaughton and Yamada [1960] reaffirmed Kleene’s result, and extended it to include regular expressions in which the operations of set intersection and complement were permitted. Brzozowski [1964] described the construction of the characteristic set equations, which we have used in several parts of this chapter.

Most of the closure properties of regular sets described in Theorem 5.7 follow immediately from Kleene’s theorem and the relations among the various set operations. The closure of regular languages under reversal was pointed out by Rabin and Scott [1959]; other closure properties (some of which are explored in Problems 5-30 through 5-33) were given by Bar-Hillel, Perles, and Shamir [1961], by Ginsburg and Rose [1963], and by Ginsburg and Spanier [1963]. The decidability result expressed as part 3 of Theorem 5.10 was established by Rabin and Scott [1959].

Problems

5.1. Design a nondeterministic finite-state accepter that accepts
a. Any binary string containing an occurrence of one of the substrings
   10100, 10110, 010111.
   b. The language \( L = \{10, 101, 110\}^* \).
In each case, try to construct directly (that is, without recourse to the
   techniques of Section 5.1.2) an equivalent deterministic accepter.
5.2. Convert each of machines a to c to an equivalent deterministic finite-state accepter. In each case, describe informally the language accepted by the machines.

(a)  
\[
\begin{array}{c}
\text{A} \\
0 \quad 1 \\
1 \quad 0 \\
1 \\
\end{array}
\quad \begin{array}{c}
\text{B} \\
1 \\
\end{array}
\quad \begin{array}{c}
\text{C} \\
0 \quad 1 \quad 0 \\
1 \\
\end{array}
\]

(b)  
\[
\begin{array}{c}
\text{A} \\
0 \\
\end{array}
\quad \begin{array}{c}
\text{B} \\
1 \\
\end{array}
\quad \begin{array}{c}
\text{C} \\
1, 0 \\
1 \\
0 \\
1 \\
\end{array}
\quad \begin{array}{c}
\text{D} \\
1 \\
\end{array}
\]

(c)  
\[
\begin{array}{c}
\text{A} \\
0 \\
1 \\
\end{array}
\quad \begin{array}{c}
\text{B} \\
0 \\
1 \\
\end{array}
\quad \begin{array}{c}
\text{D} \\
0, 1 \\
1 \\
\end{array}
\quad \begin{array}{c}
\text{C} \\
0 \\
\end{array}
\]

5.3. Let \( S = \{a, b, c, d\} \), and let \( L \) be the language consisting of all strings in \( S^* \) in which at least one of the symbols in \( S \) appears an odd number of times.

a. Design an eight-state nondeterministic FSA for \( L \). Using the techniques of Section 2.5.1, construct an equivalent deterministic accepter and eliminate redundant and nonaccessible states. How many states are in the deterministic machine?

b. Generalize the result of part a. That is, prove that if \( S \) contains \( n \) symbols, there exists a \( 2n \)-state nondeterministic accepter for \( L \) and a \( 2^n \)-state deterministic accepter for \( L \).

*c. Prove that if \( S \) has \( n \) symbols, no deterministic accepter for \( L \) can have fewer than \( 2^n \) states. [Hint: Show that if \( M \) is a deterministic accepter for \( L \) with fewer than \( 2^n \) states, there exist input strings \( \omega, \gamma \) such that (1) both strings lead \( M \) to the same state, and (2) some symbol in \( S \) occurs in \( \omega \) an odd number of times]
and in \( y \) an even number of times. Show that the existence of such strings implies incorrect behavior on the part of \( M \).]

*5.4. Let \( M \) be an arbitrary \( n \)-state FSA, and let \( M' \) be the deterministic FSA constructed from \( M \) according to the techniques of Section 5.1.2. Since each state of \( M' \) corresponds to some subset of the states of \( M \), the accepter \( M' \) can contain no more than \( 2^n \) states. That is, \( 2^n \) is an upper bound on the number of states in the reduced deterministic accepter equivalent to an arbitrary \( n \)-state FSA.

Prove that this bound is achievable for all \( n > 0 \); that is, show that for each \( n > 0 \) there exists an \( n \)-state FSA which is equivalent to no deterministic FSA with fewer than \( 2^n \) states. (*Hint:* For any \( n > 0 \), let \( M_n \) be an FSA with input alphabet \( \{0, 1\} \), state set \( \{q_0, q_1, \ldots, q_{n-1}\} \), and the following transitions:

\[
q_i \xrightarrow{1} q_{i+1}, \quad 0 \leq i < n - 1
\]

\[
q_{n-1} \xrightarrow{1} q_0
\]

\[
q_i \xrightarrow{0} q_i, \quad 0 \leq i \leq n - 1
\]

\[
q_i \xrightarrow{0} q_0, \quad 0 < i \leq n - 1
\]

Show that every subset of \( \{q_0, \ldots, q_{n-1}\} \) corresponds to an accessible state in the deterministic accepter constructed from \( M_n \). Show in addition that the deterministic accepter is reduced if the final states of \( M_n \) are chosen appropriately.)

The relative "economy of description" of a number of formal systems for specifying regular languages has been studied by Meyer and Fischer [1971]. The automaton given in the hint was described there.

5.5. From each machine in Problem 5.2, construct a right-linear grammar that generates the language accepted by the machine.

5.6. Let \( M \) be the following finite-state accepter:

\[
\begin{array}{c|ccc}
& 0 & 1 \\
\hline
A & B & C & 1 \\
B & D & E & 0 \\
C & C & A & 1 \\
D & C & E & 0 \\
E & C & E & 0 \\
\end{array}
\]
Using the rules in Table 5.1, construct a right-linear grammar $G$ such that $L(G) = L(M)$. Derive the string 00101 in the grammar and indicate the correspondence of the derivation with $M$’s state sequence for the input 00101.

5.7. Consider the following accepters $M_1$ and $M_2$:

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>A A B 1</td>
<td>A A B 1</td>
</tr>
<tr>
<td>B C B 0</td>
<td>B C D 0</td>
</tr>
<tr>
<td>C B B 1</td>
<td>C B D 1</td>
</tr>
<tr>
<td></td>
<td>D C D 0</td>
</tr>
</tbody>
</table>

Using Table 5.1, construct right-linear grammars for $L(M_1)$ and $L(M_2)$. Prove that the grammars generate the same language.

5.8. Let $M = (Q, S, P, I, F)$ be a finite-state accepter. The begin set $B(q)$ of a state $q$ of $M$ is the collection of strings that can lead to $q$ from some initial state of $M$:

$$B(q) = \{ \omega | \omega \xrightarrow{\infty} q, q' \in I \}$$

a. Verify the following assertions:

1. $\lambda \in B(q)$ if and only if $q \in I$.
2. Suppose that $\omega = \varphi s$. Then $\omega \in B(q)$ if and only if $\varphi \in B(q')$ and $q' \xrightarrow{s} q$ is a transition in $M$ for some $q' \in Q$.
3. $L(M) = \bigcup_{q \in F} B(q)$.

Compare the assertions with those of Proposition 5.1.

b. A table of rules for constructing a left-linear grammar from a finite-state accepter follows:

<table>
<thead>
<tr>
<th>Rule</th>
<th>If $M$ has</th>
<th>then $G$ has</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$I \cap F = \emptyset$</td>
<td>$\Sigma \to \lambda$</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>$q \in F$</td>
<td>$\Sigma \to N(q)$</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>$q \xrightarrow{s} q', q \in I$</td>
<td>$N(q') \to s$</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>$q \xrightarrow{s} q'$</td>
<td>$N(q') \to N(q)s$</td>
<td>?</td>
</tr>
</tbody>
</table>
Using the results of part a, provide the justification for each of rules 1 to 4.

c. Using the preceding construction, construct left-linear grammars from the accepters \( M_1 \) and \( M_2 \) of Problem 5.7. Derive the string 10010 in each grammar, and compare the derivations with the state sequences of the corresponding accepters for input 10010.

5.9. Let \( M \) be an arbitrary finite-state accepter. Let \( G_R \) be the right-linear grammar obtained from \( M \) according to the rules of Table 5.1, and let \( G_L \) be the left-linear grammar obtained from \( M \) according to the rules developed in Problem 5.8. If \( G_R \) is ambiguous, must \( G_L \) be ambiguous? If \( G_L \) is ambiguous, must \( G_R \) be ambiguous?

5.10. Construct a finite-state accepter for \( L(G) \), where \( G \) is as follows:

\[
\begin{align*}
a. \quad G: \quad & \Sigma \rightarrow T \quad \quad \quad S \rightarrow 1T \quad \quad \quad b. \quad G: \quad & \Sigma \rightarrow A \quad \quad \quad D \rightarrow aC \\
& \Sigma \rightarrow \lambda \quad \quad \quad R \rightarrow 1S \quad \quad \quad & \Sigma \rightarrow aB \quad \quad \quad D \rightarrow bC \\
P \rightarrow 0Q \quad S \rightarrow 0P \quad & A \rightarrow aC \quad \quad \quad & D \rightarrow bB \\
P \rightarrow 1R \quad T \rightarrow 0R \quad A \rightarrow bC \quad \quad \quad & B \rightarrow a \\
T \rightarrow 1P \quad S \rightarrow 1 \quad & B \rightarrow aD \quad \quad \quad & C \rightarrow a \\
R \rightarrow 0S \quad Q \rightarrow 0 \quad B \rightarrow bC \quad \quad \quad & D \rightarrow b \\
Q \rightarrow 1P \quad \quad & B \rightarrow aA \\
Q \rightarrow 0T \quad \quad & C \rightarrow aC \\
\end{align*}
\]

c. \( G: \quad \Sigma \rightarrow aB \quad \quad \quad D \rightarrow bC \\
\Sigma \rightarrow aC \quad \quad \quad D \rightarrow bD \\
B \rightarrow aA \quad \quad \quad D \rightarrow aA \\
C \rightarrow bD \quad \quad \quad A \rightarrow cA \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad A \rightarrow c
\]

5.11. Using the concepts developed in Problem 5.8, establish rules for the construction of a finite-state accepter from an arbitrary left-linear grammar. (Your rules will be analogous to those presented in Table 5.2.) Using these rules, construct a finite-state accepter for the language \( L(G) \), where \( G \) is the following grammar:

\[
\begin{align*}
G: \quad & \Sigma \rightarrow D \quad B \rightarrow Ab \quad D \rightarrow Da \quad B \rightarrow Da \\
& \Sigma \rightarrow C \quad B \rightarrow Dc \quad D \rightarrow Bc \quad A \rightarrow a \\
A \rightarrow Cb \quad C \rightarrow Aa \quad C \rightarrow Db \quad D \rightarrow c \\
A \rightarrow Ba \quad C \rightarrow Bb \quad B \rightarrow Au \quad C \rightarrow b
\end{align*}
\]

5.12. For each of the following finite-state accepters, construct and solve the corresponding right-linear set equations to obtain a regular expression for the accepted language:
5.13. For each grammar $G$ of Problem 5.10, find a regular expression for $L(G)$. (You should be able to construct the systems of set equations from the grammars themselves, rather than from corresponding accepters.)

5.14. Solve the following systems of set equations for the strings in $A$:

\begin{align*}
\text{a. } A &= 0A \cup 0B \cup 1C \cup \lambda \\
B &= 1A \cup 0B \\
C &= 0C \cup 1B
\end{align*}

\begin{align*}
\text{b. } A &= 0A \cup 1B \cup \lambda \\
B &= 0A \cup 0B \cup 1C \\
C &= 0C \cup 1A
\end{align*}

In each case, present the state diagram of an accepter corresponding to the system of equations. Use the accepters to decide if the expressions obtained from the systems are equivalent.

5.15. Consider the definition of begin sets presented at the end of Section 5.2 and again in Problem 5.8.

a. Show that the begin sets of a finite-state accepter satisfy a system of left-linear set equations, that is, a system of equations of the form

\[ X = XP \cup Q \]

where $P$ and $Q$ are arbitrary sets. Describe the system in a manner analogous to that of Proposition 5.2.

b. Following the proof of Theorem 5.4, show that $X = QP^*$ is a unique solution to the set equation $X = XP \cup Q$ whenever $P$ does not contain the empty string.

c. Using the results of parts a and b, construct and solve the system of left-linear set equations corresponding to the accepters shown in Problem 5.2. In each case, provide a regular expression for the accepted language.
5.16. Let $M$ be a deterministic finite-state accepter with input alphabet $S$ and states $\{q_0, \ldots, q_n\}$.
   a. Prove that the begin sets $B(q_0), \ldots, B(q_n)$ (see Problem 5.8) are a partition of $S^*$.
   b. Prove that the end sets $E(q_0), \ldots, E(q_n)$ need not be a partition of $S^*$. Under what conditions will they be a partition?
   c. Must the begin sets be a partition of $S^*$ if $M$ is a nondeterministic accepter?

5.17. Convert each of the following accepters to $\lambda$-accepters:

(a)

(b)

5.18. Convert the following $\lambda$-accepter to an equivalent finite-state accepter:
5.19. Let \( M \) be an arbitrary \( \lambda \)-accepter, and suppose that \( M' \) is the finite-state accepter without \( \lambda \)-transitions constructed from \( M \) in the manner described in Section 5.3.5. Prove that \( L(M) = L(M') \).

5.20. Let \( M \) be a \( \lambda \)-accepter, and let \( M' \) be a finite-state accepter obtained from \( M \) according to the construction of Section 5.3.5. Under what conditions on \( M \) will \( M' \) be a deterministic accepter?

5.21. Construct \( \lambda \)-accepters for the following languages \( L \):
   a. \( L = (00 \cup (10)(101)^* \cup 00)^*01 \).
   b. \( L = \{00, 010, 1001, 10011, 100111\}^* \).
   c. \( L = ((00 \cup 11)^* \cup (001 \cup 110)^* \cup (0011 \cup 1100)^*)^* \).

5.22. Construct a \( \lambda \)-accepter for the language \( L = (a^* \cup (ab)^* \cup (abc)^*)^*dc \). Using the methodology of Section 5.3.5, construct a finite-state accepter for \( L \).

5.23. Using Kleene's theorem, prove the following identities:
   a. \((00 \cup (00)(11)^*(01 \cup 10))^* = (00 \cup (11)^*10 \cup (11)^*01)^* \).
   b. \(0^*(11)^*(00)^*0^*(11)^*0^*0 - 0 \cup (0 \cup 11 \cup 00)(0 \cup 11 \cup 00)^* \).

5.24. Find a finite-state machine accepting the complement of \( L(M) \), where \( M \) is as follows:

![Diagram](image)

5.25. Using the concept of parallel machine operation, construct a deterministic finite-state accepter for the language \( L = (110(0 \cup 1)^*) \cap ((0 \cup 1)^*101) \).

5.26. In Section 5.4.1, we showed that particular choices of the function \( f \) for the parallel operating accepters of Figure 5.24 produce machines for union, intersection, difference, and complement. If we permit the machines operating in parallel to be nondeterministic accepters, will the indicated choice of \( f \) still produce a machine for union? for intersection? for complement? for difference? In each case where the answer is no, will any choice of \( f \) produce a machine for the given operation?
5.27. Let \( M_1 \) and \( M_2 \) be finite-state machines with input and output alphabets \([0, 1]\), and let \( M \) be the following parallel machine:

For any Boolean function \( F \), we say that \( M_1 \) and \( M_2 \) are \( F \)-related if the output from \( M \) is always 1, that is, if \( \varphi \in \mathbf{1}^* \) for all \( \omega \in \{0, 1\}^* \). Let \( F_1, F_2, F_3, \) and \( F_4 \) be defined as follows:

For \( i = 1, \ldots, 4 \), let \( \rho_i \) be the relation defined as follows:

\[ M_1 \rho_i M_2 \] just if \( M_1 \) and \( M_2 \) are \( F_i \)-related

Which of the relations \( \rho_i \) are equivalence relations? If \( M_1 \) and \( M_2 \) are deterministic finite-state accepters, prove that they are equivalent if and only if they are \( F_1 \)-related. If \( M_1 \) and \( M_2 \) are nondeterministic accepters, is it possible to define a function \( F \) such that \( M_1 \) and \( M_2 \) are equivalent just if they are \( F \)-related?

5.28. Let \( M \) be a finite-state accepter for the regular set \( X \), and let \( X^R \) be the reverse of \( X \). Describe a method of obtaining from \( M \) an accepter \( M' \) for \( X^R \). Is the method valid if \( M \) is nondeterministic?

5.29. A proper prefix of a string \( \omega \) is any string \( \varphi \) such that \( \omega = \varphi \psi, \psi \neq \lambda \). A proper suffix of a string \( \omega \) is any string \( \psi \) such that \( \omega = \varphi \psi, \varphi \neq \lambda \). Let \( R \) be a regular set.

a. Show that \( R' = \{ \omega | \omega \text{ is the proper prefix of some string in } R \} \) is regular.

b. Show that \( R'' = \{ \omega | \omega \text{ is the proper suffix of some string in } R \} \) is regular.

c. Suppose that we define \( R''' \) to be the set \( \{ \omega | \omega = \varphi \psi, \text{ where } \varphi \text{ is the proper prefix of some string } \alpha \in R, \text{ and } \psi \text{ is the proper suffix of some string } \beta \in R \} \). Is \( R''' \) a regular set?
5.30. Let $\mathcal{R}$ be a regular set and define

$$Min(\mathcal{R}) = \{\omega \in \mathcal{R} \mid \text{no string in } \mathcal{R} \text{ is a proper prefix of } \omega \text{ (see Problem 5.29)}\}$$

$$Max(\mathcal{R}) = \{\omega \in \mathcal{R} \mid \text{no string in } \mathcal{R} \text{ is a proper suffix of } \omega\}$$

a. Find $Min(\mathcal{R})$ and $Max(\mathcal{R})$ for $\mathcal{R} = 10^*; \text{ for } \mathcal{R} = (0 \cup 1)^*10$.

b. Show that both $Min(\mathcal{R})$ and $Max(\mathcal{R})$ are regular for any regular set $\mathcal{R}$.

5.31. Let $\mathcal{R} \subseteq V^*$ be a regular set on some alphabet $V$, and let $\alpha$ be a fixed string in $V^*$. Define the **quotient of $\mathcal{R}$ with respect to $\alpha$** to be the set $\mathcal{R}/\alpha = \{\beta \mid \beta \alpha \text{ is in } \mathcal{R}\}$. Prove that $\mathcal{R}/\alpha$ is regular. (*Hint: Consider tracing paths for $\alpha$ back from the final states of an accepter for $\mathcal{R}$.*)

5.32. Let $\mathcal{R} \subseteq V^*$ be a regular set on some alphabet $V$, and let $\alpha$ be a fixed string in $V^*$. Define the **derivative of $\mathcal{R}$ with respect to $\alpha$** to be the set $\mathcal{R}_\alpha = \{\beta \mid \alpha \beta \text{ is in } \mathcal{R}\}$. Prove that $\mathcal{R}_\alpha$ is regular. Given that there exists an $n$-state accepter for $\mathcal{R}$, how many distinct derivatives can $\mathcal{R}$ possess? (*Hint: Consider adding $\lambda$-transitions to an accepter for $\mathcal{R}$.*)

5.33. Theorem 5.7 states that the family $\mathcal{A}$ of regular sets on a given alphabet $V$ is closed under the operations of set union, concatenation, and closure. In addition, we know that $\mathcal{A}$ contains all finite sets in $V^*$, since each such set is accepted by some finite-state accepter. Show that if $\mathcal{S}$ is any family of sets in $V^*$ that contains the finite sets and is closed under the operations of set union, concatenation, and closure, then $\mathcal{S}$ contains $\mathcal{A}$. (That is, $\mathcal{A}$ is the smallest family of sets in $V^*$ that is closed under these operations and contains the finite sets.)

5.34. Which of the grammars of Problem 5.10 are ambiguous? For each ambiguous grammar, exhibit multiple derivations of some string generated by the grammar. Convert each ambiguous grammar to an unambiguous grammar generating the same language.

5.35. We say that a grammar $G$ is **minimal** if no grammar for $L(G)$ contains fewer productions than $G$. Let $G$ be a right-linear grammar generating a regular set $\mathcal{R}$, and suppose that $G$ has $n$ productions. What is the maximum number of productions in a minimal unambiguous grammar for $L(G)$? (*Hint: See Problem 5.4.*)

5.36. Prove the assertion in part 3 of the proof of Theorem 5.10: an $n$-state FSA $M$ accepts infinitely many strings just if $M$ accepts a string of length at least $n$.

5.37. A word $\omega$ is a **palindrome** if it is its own reverse (that is, if $\omega = \omega^R$); a language is a **palindrome language** just if each of its elements is a palindrome. (See Problem 3.3.) Define a decision procedure for
deciding whether a finite-state accepter accepts a palindrome language. In terms of their accepters, how might such languages be characterized?

*5.38. Let $G$ and $G'$ be right-linear grammars. We say that $G'$ is a reduction of $G$ if $L(G') = L(G)$, and $G'$ has fewer productions than $G$, or $G'$ has the same number of productions as $G$ but has fewer nonterminal symbols. We say that a right-linear grammar is choice free if, during any derivation permitted by the grammar, only one production is applicable at any but the first step of the derivation. We say that a grammar $G$ is irreducible if it is choice free and there is no choice-free reduction of $G$.

a. Give a procedure for determining whether a given right-linear grammar is choice free.

b. Give a procedure for determining whether a given right-linear grammar is irreducible.

c. Show how to construct from a given right-linear grammar $G$ an irreducible right-linear grammar $G'$ such that $L(G') = L(G)$.

(*Hint: Consider the interpretations of reduced and choice free in terms of a finite-state accepter corresponding to $G$.*)