Year	Subject Title	Sem.	Sub Code
2018 -19 Onwards	DISCRETE MATHEMATICS FOR COMPUTER SCIENCE	II	18BCS24A

OBJECTIVES:

1. To understand the concepts of basic Discrete structures.
2. To get knowledge about applying the properties of Discrete Mathematical Structures.

UNIT: IV

FORMAL LANGUAGES AND AUTOMATA: Grammers and Languages– Finite state machine – Finite state Acceptors and Regular grammars.

(Chapter III - Section: 3.3; Chapter IV – Section: 4.6; Chapter VI - Section: 6.1)

3-3 GRAMMARS AND LANGUAGES

The basic machine instructions of a digital computer are very primitive compared with the complex operations that must be performed in various disciplines such as engineering, commerce, and mathematics. Although a complex procedure can be programmed in machine language, it is desirable to use a high-level language that contains instructions similar to those required in a particular application. For example, in a payroll application, one wants to manipulate employee records in a master file, generate complex reports, and perform rather simple arithmetic operations on certain data. A language such as COBOL which has high-level commands that manipulate records and generate reports is a definite asset to a programmer.

While high-level programming languages reduce much of the drudgery of machine language programming, they also introduce new problems. A program

(compiler) which converts a program to some object language such as machine language must be written. Also, programming languages must be precisely defined. Sometimes it is the existence of a particular compiler which finally provides the precise definition of a language. The specification of a programming language involves the definition of the following:

1. The set of symbols (or alphabet) that can be used to construct correct programs
2. The set of all correct programs
3. The "meaning" of all correct programs

In this section we shall be concerned with the first two items in the specification of programming languages.

A language L can be considered a subset of the free monoid on an alphabet (see Sec. 3-2.1). The language consisting of the free monoid is not particularly interesting since it is too large. Our definition of a language L is a set of strings or sentences over some finite alphabet V_T, so that $L \subseteq V_T^*$.

How can a language be represented? A language consists of a finite or an infinite set of sentences. Finite languages can be specified by exhaustively enumerating all their sentences. However, for infinite languages, such an enumeration is not possible. On the other hand, any device which specifies a language should be finite. One method of specification which satisfies this requirement uses a generative device called a *grammar*. A grammar consists of a finite set of rules or productions which specify the syntax of the language. In addition, a grammar imposes structure on the sentences of a language. The study of grammars constitutes an important subarea of computer science called formal languages. This area emerged in the mid-1950s as a result of the efforts of Noam Chomsky who gave a mathematical model of a grammar in connection with his study of natural languages. In 1960, the concept of a grammar became important to programmers because the syntax of ALGOL 60 was described by a grammar.

A second method of language specification is to have a machine, called an acceptor, determine whether a given sentence belongs to the language. This approach is discussed further in Chap. 6, along with some very interesting and important relationships that exist between grammars and acceptors.

In this section we are concerned with a grammar as a mathematical system for defining languages and as a device for giving some useful structure to sentences in a language. The problem of syntactic analysis will be discussed briefly.

3-3.1 Discussion of Grammars

It was mentioned earlier that a grammar imposes a structure on the sentences of a language. For a sentence in English such a structure is described in terms of subject, predicate, phrase, noun, and so on. On the other hand, for a program, the structure is given in terms of procedures, statements, expressions, etc. In any case, it may be desirable to describe all such structures and to obtain a set of all the correct or admissible sentences in a language. For example, we may have a set of correct sentences in English or a set of valid ALGOL programs. The grammatical structure of a language helps us determine whether a particular sentence does or does not belong to the set of correct sentences. The grammatical

structure of a sentence is generally studied by analyzing the various parts of a sentence and their relationships to one another; this analysis is called *parsing*.

Consider the sentence "a monkey ate the banana." Its structure, or parse, is shown in Fig. 3-3.1. This diagram of a parse displays the syntax of a sentence in a manner similar to a tree and is therefore called a syntax tree. Each node in the diagram represents a phrase of the syntax. The words such as "the," "monkey," etc., are the basic symbols, or primitives, of the language.

The syntax of a small subset of the English language can be described by using the symbols

S: sentence $\quad V$: verb $\quad O$: object $\quad A$: article $\quad N$: noun

SP: subject phrase $\quad VP$: verb phrase $\quad NP$: noun phrase

in the following rules:

$S \longrightarrow SP\ VP$	$N \longrightarrow$ tree
$SP \longrightarrow A\ N$	$VP \longrightarrow V\ O$
$A \longrightarrow$ a	$V \longrightarrow$ ate
$A \longrightarrow$ the	$V \longrightarrow$ climbs
$N \longrightarrow$ monkey	$O \longrightarrow NP$
$N \longrightarrow$ banana	$NP \longrightarrow A\ N$

These rules state that a sentence is composed of a "subject phrase" followed by a "verb phrase"; the "subject phrase" is composed of an "article" followed by a "noun"; a verb phrase is composed of a "verb" followed by an "object"; and so on.

The structure of a language is discussed by using symbols such as "sen-

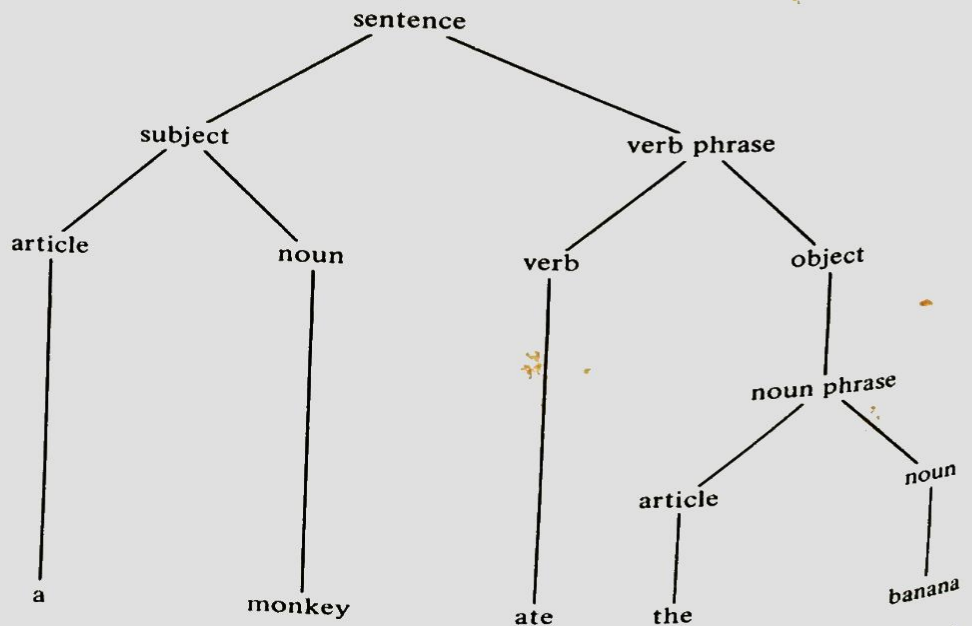

FIGURE 3-3.1

tence," "verb," "subject phrase," and "verb phrase," which represent *syntactic classes* of elements. Each syntactic class consists of a number of alternative structures, and each structure consists of an ordered set of items which are either primitives (of the language) or syntactic classes. These alternative structures are called *productions* or *rules of syntax*. For example, the production, $S \rightarrow SP\ VP$ defines a "sentence" to be composed of a "subject phrase" followed by a "verb phrase." The symbol $\rightarrow$ separates the syntactic class "sentence" from its definition.

The syntactic class and the arrow symbol along with the interpretation of a production enable us to describe a language. A system or language which describes another language is known as a *metalanguage*. The metalanguage used to teach German at most Canadian universities is English, while the metalanguage used to teach English is English. The diagram of a sentence describes its *syntax* but not its meaning or *semantics*. At this time, we are mainly concerned with the syntax of a language, and the device which we have just defined to give the syntactic definition of the language is called a *grammar*.

Using the grammatical rules for our example, we can either produce (generate) or derive a sentence in the language. A computer programmer is concerned with using the productions (grammatical rules) to produce syntactically correct programs. The compiler of a language, on the other hand, is faced with the problem of determining whether a given sentence (source program) is syntactically correct based upon the given grammatical rules. If the syntax is correct, then it produces object code.

Consider the problem of trying to generate or produce the sentence "a monkey ate the banana" from the set of productions given. It is accomplished by starting first with the syntactic class symbol S and looking for a production which has S to the left of the arrow. There is only one such production, namely,

$$S \rightarrow SP\ VP$$

We have replaced the class S by its only possible composition. We then take the string

$$SP\ VP$$

and look for a production whose left-hand side is SP and then replace it with the right-hand side of that production. The application of the only production possible produces the string

$$A\ N\ VP$$

We next look for a production whose left part is A, and two such productions are found. By selecting the production $A \rightarrow$ a and upon substituting its right-hand side in the string $A\ N\ VP$, we obtain the string

$$\text{a}\ N\ VP$$

This process is continued until we arrive at the correct sentence. At this point, the sentence contains only primitive or terminal elements of the language (no classes). A complete derivation or generation of the sentence "a monkey ate the

banana" is as follows:

$$\begin{aligned} S &\Rightarrow SP\ VP \\ &\Rightarrow A\ N\ VP \\ &\Rightarrow \text{a}\ N\ VP \\ &\Rightarrow \text{a monkey}\ VP \\ &\Rightarrow \text{a monkey}\ V\ O \\ &\Rightarrow \text{a monkey ate}\ O \\ &\Rightarrow \text{a monkey ate}\ NP \\ &\Rightarrow \text{a monkey ate}\ A\ N \\ &\Rightarrow \text{a monkey ate the}\ N \\ &\Rightarrow \text{a monkey ate the banana} \end{aligned}$$

Here the symbol $\Rightarrow$ denotes that the string on the right-hand side of the symbol can be obtained by applying one rewriting rule to the previous string.

The rules for the example language can produce a number of sentences. Examples of such sentences are

The monkey ate the banana.

The monkey climbs a tree.

The monkey climbs the tree.

The banana ate the monkey.

The last of these sentences, although grammatically correct, doesn't really make sense. This situation is often allowed in the specification of languages. There are many valid FORTRAN and PL/I programs that do not make sense. It is easier to define languages if certain sentences of questionable validity are allowed by the rewriting rules.

The set of sentences that can be generated by the example rules is finite. Any interesting language usually consists of an infinite set of sentences. As a matter of fact, the importance of a finite device such as a grammar is that it permits the study of the structure of a language consisting of an infinite set of sentences.

Let the symbols L, D, and I denote the classes L: letter, D: digit, and I: identifier. The productions which follow are recursive and produce an infinite set of names because the syntactic class I is present on both the left and the right sides of certain productions:

$$\begin{array}{ll} I \rightarrow L & D \rightarrow 0 \\ I \rightarrow ID & D \rightarrow 1 \\ I \rightarrow IL & \cdots \\ L \rightarrow a & D \rightarrow 9 \\ L \rightarrow b & \\ \cdots & \\ L \rightarrow z & \end{array}$$

It is easily seen that the class I defines an infinite set of strings or names in which each name consists of a letter followed by any number of letters or digits. This set is a consequence of using recursion in the definition of the productions $I \rightarrow ID$ and $I \rightarrow IL$. In fact, recursion is fundamental to the definition of an infinite language by the use of a grammar.

3-3.2 Formal Definition of a Language

Let us now formalize the idea of a grammar and how it is used. For this purpose, let V_T be a finite nonempty set of symbols called the *alphabet*. The symbols in V_T are called *terminal symbols*. The *metalanguage* which is used to generate strings in the language is assumed to contain a set of syntactic classes or variables called *nonterminal symbols*. The set of nonterminal symbols is denoted by V_N, and the elements of V_N are used to define the syntax (structure) of the language. Furthermore, the sets V_N and V_T are assumed to be disjoint. The set $V_N \cup V_T$ consisting of nonterminal and terminal symbols is called the *vocabulary* of the language. We shall use capital letters such as $A, B, C, \ldots, X, Y, Z$ to denote nonterminal symbols, while $S_1, S_2, \ldots$ represent the elements of the vocabulary. The strings of terminal symbols are denoted by lowercase letters $x, y, z, \ldots$, while strings of symbols over the vocabulary are given by $\alpha, \beta, \gamma, \ldots$. The length of a string α will be denoted by $|\alpha|$.

Definition 3-3.1 A (phrase structure) *grammar* is defined by a 4-tuple $G = \langle V_N, V_T, S, \Phi \rangle$ where V_T and V_N are sets of terminal and nonterminal (syntactic class) symbols respectively. S, a distinguished element of V_N and therefore of the vocabulary, is called the starting symbol. Φ is a finite subset of the relation from $(V_T \cup V_N)^* V_N (V_T \cup V_N)^*$ to $(V_T \cup V_N)^*$. In general, an element $\langle \alpha, \beta \rangle$ is written as $\alpha \rightarrow \beta$ and is called a *production rule* or a *rewriting rule*.

For our example given earlier, we may write the grammar as $G_1 = \langle V_N, V_T, S, \Phi \rangle$ in which

$$V_N = \{I, L, D\}$$

$$V_T = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$S = I$$

$$\Phi = \{I \rightarrow L, I \rightarrow IL, I \rightarrow ID, L \rightarrow a, L \rightarrow b, \ldots, L \rightarrow z, D \rightarrow 0, D \rightarrow 1, \ldots, D \rightarrow 9\}$$

Definition 3-3.2 Let $G = \langle V_N, V_T, S, \Phi \rangle$ be a grammar. For $\sigma, \psi \in (V_N \cup V_T)^* - \{\Lambda\}$, σ is said to be a *direct derivative* of ψ, written as $\psi \Rightarrow \sigma$, if there are strings ϕ_1 and ϕ_2 (including possibly empty strings) such that $\psi = \phi_1 \alpha \phi_2$ and $\sigma = \phi_1 \beta \phi_2$ and $\alpha \rightarrow \beta$ is a production of G.

If $\psi \Rightarrow \sigma$, we may also say that ψ directly produces σ or σ directly reduces to ψ. For grammar G_1 of our example, we have listed in Table 3-3.1 some illustrations of direct derivations.

Table 3-3.1

ψ	σ	Rule used	ϕ_1	ϕ_2
I	L	$I \to L$	Λ	Λ
Ib	Lb	$I \to L$	Λ	b
Lb	ab	$L \to a$	Λ	b
LD	$L1$	$D \to 1$	L	Λ
LD	aD	$L \to a$	Λ	D

These concepts can now be extended to produce a string σ not necessarily directly but in a number of steps from a string ψ.

Definition 3-3.3 Let $G = \langle V_N, V_T, S, \Phi \rangle$ be a grammar. The string ψ produces σ (σ reduces to ψ, or σ is the derivation of ψ), written as $\psi \overset{+}{\Rightarrow} \sigma$, if there are strings $\phi_0, \phi_1, \ldots, \phi_n$ $(n > 0)$ such that $\psi = \phi_0 \Rightarrow \phi_1$, $\phi_1 \Rightarrow \phi_2, \ldots, \phi_{n-1} \Rightarrow \phi_n$ and $\phi_n = \sigma$. The relation $\overset{+}{\Rightarrow}$ is the transitive closure of the relation $\Rightarrow$. If we let $n = 0$, then we can define the reflexive transitive closure of $\Rightarrow$ as

$$\psi \overset{*}{\Rightarrow} \sigma \quad \Leftrightarrow \quad \psi \overset{+}{\Rightarrow} \sigma \quad \text{or} \quad \psi = \sigma$$

Returning to the grammar G_1, we show that the string $abc12$ is derived from I by following the derivation sequence:

$$I \Rightarrow ID \Rightarrow IDD \Rightarrow ILDD \Rightarrow ILLDD \Rightarrow LLLDD \Rightarrow aLLDD$$
$$\Rightarrow abLDD \Rightarrow abcDD \Rightarrow abc1D \Rightarrow abc12$$

Note that as long as we have a nonterminal character in the string, we can produce a new string from it. On the other hand, if a string contains only terminal symbols, then the derivation is complete, and we cannot produce any further strings from it.

Definition 3-3.4 A *sentential form* is any derivative of the unique nonterminal symbol S. The language L generated by a grammar G is the set of all sentential forms whose symbols are terminal, i.e.,

$$L(G) = \{\sigma \mid S \overset{*}{\Rightarrow} \sigma \text{ and } \sigma \in V_T^*\}$$

Therefore, the language is merely a subset of the set of all terminal strings over V_T.

We shall now give a number of examples of grammars.

EXAMPLE 1 Let $G_2 = \langle \{E, T, F\}, \{a, +, *, (,)\}, E, \Phi \rangle$ where Φ consists of the productions

$$E \to E + T$$
$$E \to T$$
$$T \to T * F$$
$$T \to F$$
$$F \to (E)$$
$$F \to a$$

where the variables E, T, and F represent the names "expression," "term," and "factor" commonly used in conjunction with arithmetic expressions. A derivation for the expression $a * a + a$ is

$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow T + T \\ &\Rightarrow T * F + T \\ &\Rightarrow F * F + T \\ &\Rightarrow a * F + T \\ &\Rightarrow a * a + T \\ &\Rightarrow a * a + F \\ &\Rightarrow a * a + a \end{aligned}$$

EXAMPLE 2 The language $L(G_3) = \{a^n b^n c^n \mid n \geq 1\}$ is generated by the following grammar.

$$G_3 = \langle \{S, B, C\}, \{a, b, c\}, S, \Phi \rangle$$

where Φ consists of the productions

$$\begin{aligned} S &\rightarrow aSBC \\ S &\rightarrow aBC \\ CB &\rightarrow BC \\ aB &\rightarrow ab \\ bB &\rightarrow bb \\ bC &\rightarrow bc \\ cC &\rightarrow cc \end{aligned}$$

The following is a derivation for the string $a^2b^2c^2$:

$$\begin{aligned} S &\Rightarrow aSBC \\ &\Rightarrow aaBCBC \\ &\Rightarrow aaBBCC \\ &\Rightarrow aabBCC \\ &\Rightarrow aabbCC \\ &\Rightarrow aabbcC \\ &\Rightarrow aabbcc \end{aligned}$$

EXAMPLE 3 The language $L(G_4) = \{a^n b a^n \mid n \geq 1\}$ is generated by the grammar

$$G_4 = \langle \{S, C\}, \{a, b\}, S, \Phi \rangle$$

where Φ is the set of productions

$$\begin{aligned} S &\rightarrow aCa \\ C &\rightarrow aCa \\ C &\rightarrow b \end{aligned}$$

A derivation for a^2ba^2 consists of the following steps:

$$\begin{aligned} S &\Rightarrow aCa \\ &\Rightarrow aaCaa \\ &\Rightarrow aabaa \end{aligned}$$

EXAMPLE 4 The language $L(G_5) = \{a^n b a^m \mid n, m \geq 1\}$ is generated by the grammar

$$G_5 = \langle \{S, A, B, C\}, \{a, b\}, S, \Phi \rangle$$

where the set of productions is

$$\begin{aligned} S &\to aS \\ S &\to aB \\ B &\to bC \\ C &\to aC \\ C &\to a \end{aligned}$$

The sentence a^2ba^3 has the following derivation:

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow aaB \\ &\Rightarrow aabC \\ &\Rightarrow aabaC \\ &\Rightarrow aabaaC \\ &\Rightarrow aabaaa \end{aligned}$$

Chomsky classified grammars into four classes by imposing restrictions on the productions.

Definition 3-3.5 A *context-sensitive grammar* contains only productions of the form $\alpha \to \beta$ where $|\alpha| \leq |\beta|$.

This restriction on a production prevents β from being empty. Because of this restriction the set of sentences generated by such a grammar is recursively solvable. The restriction on the productions of a context-sensitive grammar can be equivalently stated as follows:

> α and β in the production $\alpha \to \beta$ can be expressed as $\alpha = \phi_1 A \phi_2$ and $\beta = \phi_1 \psi \phi_2$ (ϕ_1 and/or ϕ_2 are possibly empty) where ψ must be nonempty.

The meaning of "context-sensitive" becomes clearer with this reformulation. The application of a production $\phi_1 A \phi_2 \to \phi_1 \psi \phi_2$ to a sentential form means that A is rewritten as ψ in the context ϕ_1 and ϕ_2. Context-sensitive grammars are said to generate context-sensitive languages. G_3 is an example of a context-sensitive grammar.

We now impose a further restriction on productions to obtain a context-free grammar.

Definition 3-3.6 A *context-free grammar* contains productions of only the form $\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$ and $\alpha \in V_N$.

With such grammars, the rewriting variable in a sentential form is rewritten regardless of the other symbols in its vicinity or context. It has led to the term "context-free" for grammars consisting of productions whose left-hand side consists of a single class symbol. Context-free grammars do not have the power to represent even significant parts of the English language since context dependency is often required in order to properly analyze the structure of a sentence. Context-free grammars are not capable of specifying (or determining) that a certain variable was declared when it is used in some expression in a subsequent statement of a source program. However, these grammars can specify most of the syntax for computer or artificial languages since these are, by and large, simple in structure. Context-free grammars are said to generate context-free languages. Grammars G_1, G_2, and G_4 are examples of context-free grammars.

A final restriction leads to the definition of regular grammars.

Definition 3-3.7 A *regular grammar* contains only productions of the form $\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$, $\alpha \in V_N$, and β has the form aB or a where $a \in V_T$ and $B \in V_N$.

The set of languages generated by such grammars is said to be regular. G_5 is an example of a regular grammar.

Let the unrestricted, context-sensitive, context-free, and regular grammars be denoted by the class symbols T_0, T_1, T_2, and T_3 respectively. If $L(T_i)$ represents the class of languages that can be generated by the class of T_i grammars, it can be shown that

$$L(T_3) \subset L(T_2) \subset L(T_1) \subset L(T_0)$$

and therefore the four classes of grammars form a hierarchy. Corresponding to each class of grammars there is a class of machines (acceptors) that will accept the class of languages generated by the former. We return to this problem in Chap. 6.

We conclude this subsection by introducing a different metalanguage from the one which was previously used. The metavariables or syntactic classes will be enclosed by the symbols ⟨ and ⟩. Using this terminology, the symbol ⟨sentence⟩ is a symbol of V_N and the symbol "sentence" is an element of V_T. In this way, no confusion or ambiguity arises when attempting to distinguish the two symbols. This metalanguage is known as *Backus Naur Form* (*BNF*) and has been used extensively in the formal definition of many programming languages. A popular language described using BNF is ALGOL. For example, the definition of an identifier in BNF is given as

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩

⟨letter⟩ ::= $a \mid b \mid c \mid \cdots \mid y \mid z$

⟨digit⟩ ::= $0 \mid 1 \mid 2 \mid \cdots \mid 8 \mid 9$

The symbol ::= replaces the symbol $\rightarrow$ in the grammar notation, and | is used to separate different right-hand sides of productions corresponding to the same

left-hand side. The symbol ::= is interpreted as "is defined as" and | as "or." BNF gives a much more compact description of a language than could be achieved with the previous metalanguage.

3-3.3 Notions of Syntax Analysis

In the following pages we briefly discuss the problem of syntax analysis or parsing. The *parse* of a sentence is the construction of a derivation for that sentence; that is, a sequence of productions used in generating a given sentence from the starting symbol is required.

An important aid to understanding the syntax of a sentence is a *syntax tree*. The structural relationships between the parts of a sentence are easily seen from its syntax tree. Consider the grammar described earlier for the set of valid identifiers or variable names. The derivation of the identifier $a1$ is $\langle\text{identifier}\rangle \Rightarrow \langle\text{identifier}\rangle\langle\text{digit}\rangle \Rightarrow \langle\text{letter}\rangle\langle\text{digit}\rangle \Rightarrow a\,\langle\text{digit}\rangle \Rightarrow a1$. Let us now illustrate how to construct a syntax tree corresponding to this derivation. This process is shown as a sequence of diagrams in Fig. 3-3.2, where each diagram corresponds to a

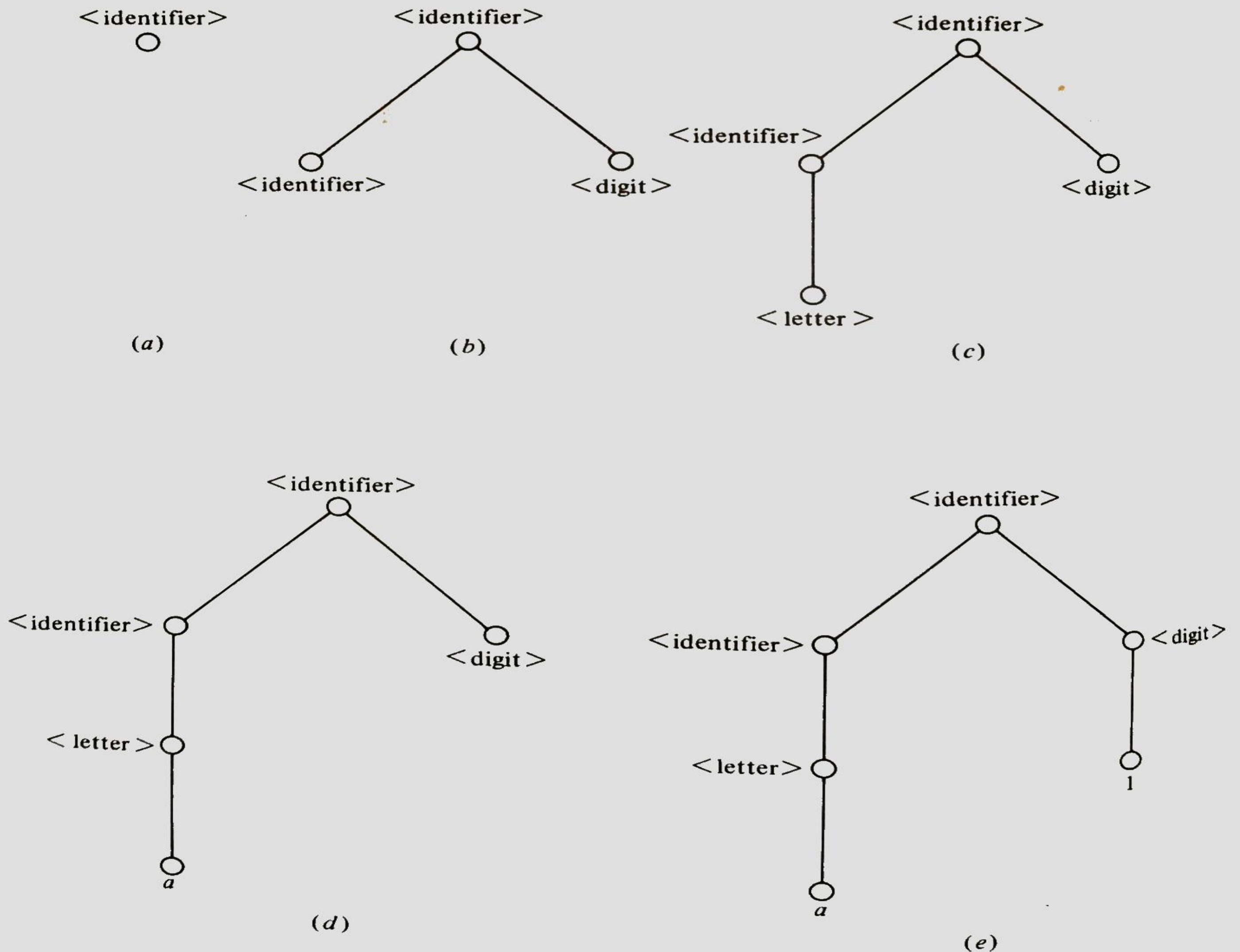

FIGURE 3-3.2 Syntax tree for sentence a1.

sentential form in the derivation of the sentence. The syntax tree has a distinguished point called its root, which is labeled by the starting symbol ⟨identifier⟩ of the grammar. From the root we draw two downward branches (see Fig. 3-3.2*b*) corresponding to the rewriting of ⟨identifier⟩ by ⟨identifier⟩⟨digit⟩. The symbol ⟨identifier⟩ in the sentential form ⟨identifier⟩⟨digit⟩ is then rewritten as ⟨letter⟩ by using the production ⟨identifier⟩ :: = ⟨letter⟩ (see Fig. 3-3.2*c*). This process continues for each production applied until Fig. 3-3.2*e* is obtained.

Given a sentence in the language, the construction of a parse can be described pictorially in Fig. 3-3.3 where the root and leaves (which represent the terminal symbols in the sentence) of the tree are known and the rest of the syntax tree must be found. There are a number of ways by which this construction can be accomplished. First, an attempt to construct the tree can be initiated by starting at the root and proceeding downward toward the leaves. This method is called a *top-down parse*. Alternatively, the completion of the tree can be attempted by starting at the leaves and moving upward toward the root. This method is called a *bottom-up parse*. The top-down and bottom-up approaches can be combined to yield other possibilities.

Let us briefly discuss top-down parsing. Consider the identifier *c*2 generated by the BNF grammar of the previous subsection. The first step is to construct the direct derivation ⟨identifier⟩ ⇒ ⟨identifier⟩⟨digit⟩. At each successive step, the leftmost variable A of the current sentential form $\phi_1 A \phi_2$ is replaced by the right part of a production $A ::= \psi$ to obtain the next sentential form. This process is shown for the identifier *c*2 by the five trees of Fig. 3-3.4.

We have very conveniently chosen the rules which generate the given identifier. If the first step had been the construction of the direct derivation ⟨identifier⟩ ⇒ ⟨identifier⟩⟨letter⟩, then we would have eventually produced the sentential form *c*⟨letter⟩ where it would have been impossible to obtain *c*2. At this point, a new alternative would have to be tried by restarting the procedure and choosing the rule ⟨identifier⟩ :: = ⟨identifier⟩⟨digit⟩.

A bottom-up parsing technique begins with a given string and tries to reduce it to the starting symbol of the grammar. The first step in parsing the identifier *c*2 is to reduce *c* to ⟨letter⟩, resulting in the sentential form ⟨letter⟩2. The direct derivation ⟨letter⟩2 ⇒ *c*2 has now been constructed, as shown in Fig. 3-3.5*d*. The next step is to reduce ⟨letter⟩ to ⟨identifier⟩, as represented by Fig. 3-3.5*c*. The process continues until the entire syntax tree of Fig. 3-3.5*a* is constructed. Note that it is possible to construct other derivations, but the re-

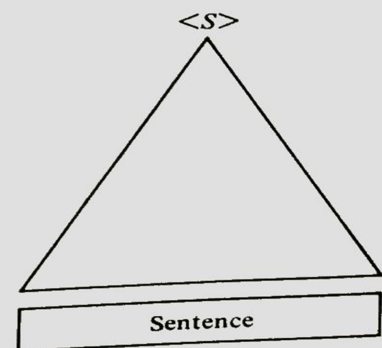

FIGURE 3-3.3

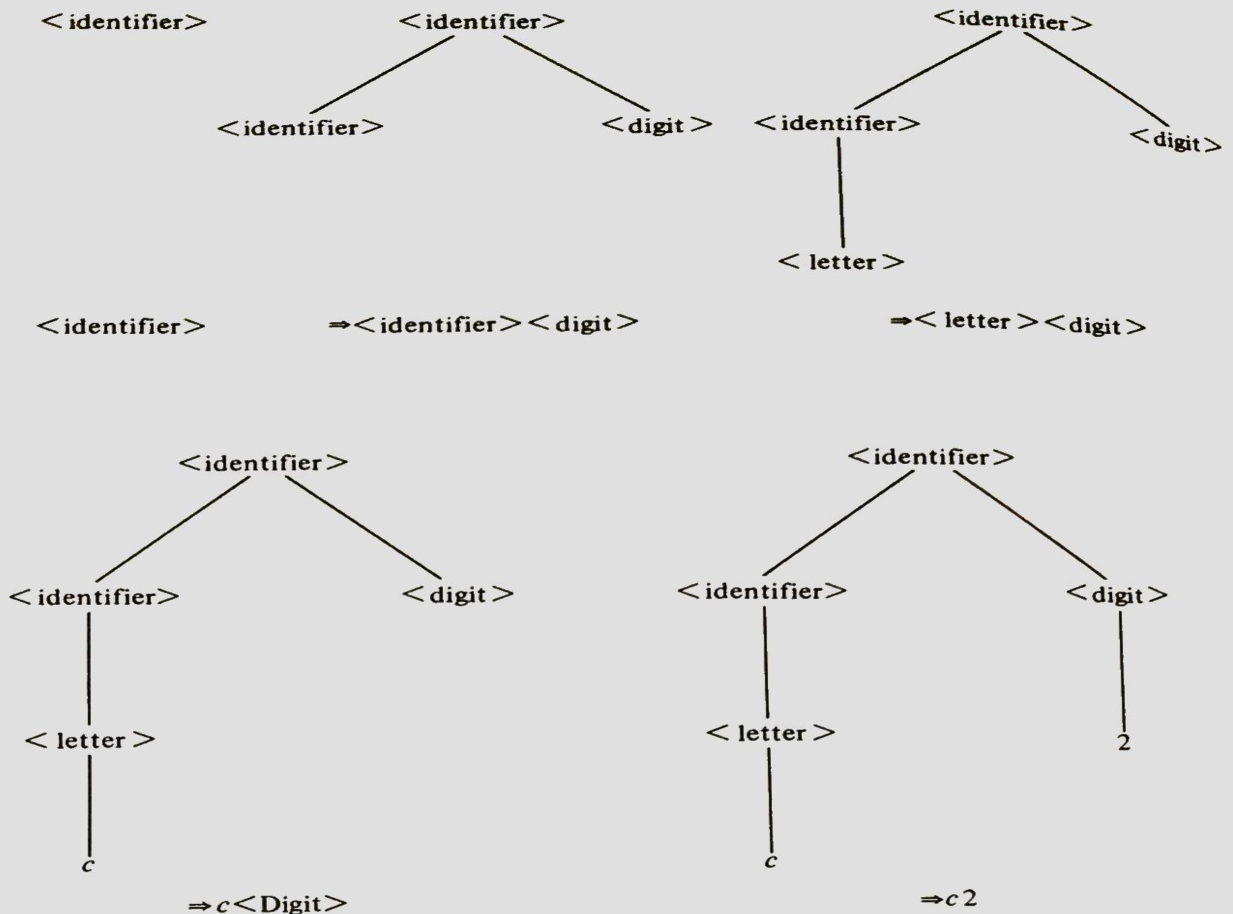

FIGURE 3-3.4 Trace of a top-down parse.

sulting syntax tree is the same. We will return to the very important problem of bottom-up parsing in Chap. 5.

We now turn to a more general discussion of language translation. A compiler for a certain language is concerned with a number of tasks, namely, determining whether a sentence belongs to the language, constructing a syntax tree for the sentence, and generating object code for the given sentence if its syntax and semantics are valid. This process can be represented by Fig. 3-3.6.

The source program is input to a scanner whose purpose is to separate the incoming text into pieces such as constants, variable names, key words (such as DO, IF, and the like in FORTRAN), and operators. This type of analysis is quite simple to perform. Usually the scanner constructs tables which contain variable names, constants, and labels.

The scanner feeds the syntax analyzer whose task is essentially to construct a syntax tree (or its equivalent) for the given sentence. The syntax analyzer is much more complicated than the scanner. The output of the syntax analyzer is fed to the code generation block which uses the syntax tree for the sentence and other things (which are not specified for simplicity) to generate object code for that sentence.

<identifier>

<identifier> <digit>

<letter> 2

c

(*a*)
<identifier>

<identifier>

<letter> <digit>

c 2

(*b*)
⇒<identifier><digit>

<identifier>

<letter>

c 2

(*c*)
⇒<identifier> 2

<letter>

c 2

(*d*)
⇒<letter> 2

c 2

(*e*)
⇒*c*2

FIGURE 3-3.5 Trace of a bottom-up parse.

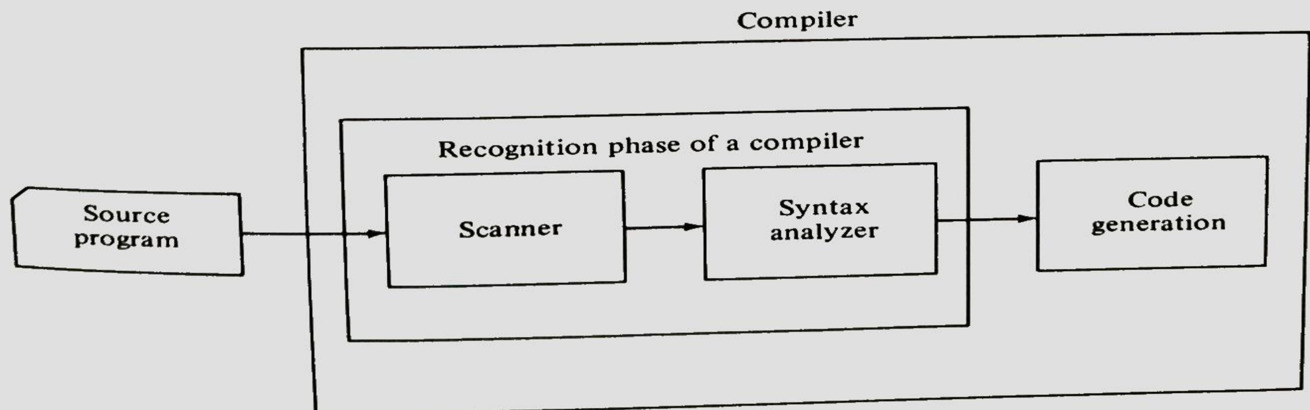

FIGURE 3-3.6 Block diagram of a compiler.

Consider the following scanner problem. It is required to construct a scanner that will split up a sentence into a number of parts. These parts are strings such as identifiers, literal constants, adding operators, multiplying operators, and exponential operators. The syntax of these primitive classes is now given. An identifier is described by the following rules:

⟨identifier⟩ :: = ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩ | ⟨letter⟩

⟨letter⟩ :: = $A \mid B \mid C \mid \cdots Y \mid Z$

⟨digit⟩ :: = 0 | 1 | · · ·8 | 9

Literal constants are described by the rules

⟨literal constant⟩ :: = ⟨digit string⟩ | ⟨digit string⟩.⟨digit string⟩ | ⟨digit string⟩.

⟨digit string⟩: : = ⟨digit⟩ | ⟨digit string⟩ ⟨digit⟩

The different classes of operators are described by the productions

⟨adding operator⟩ :: = + | −

⟨multiplying operator⟩ :: = * | /

⟨exponentiation operator⟩ :: = **

The syntactic classes ⟨identifier⟩, ⟨literal constant⟩, ⟨adding operator⟩, ⟨multiplying operator⟩, ⟨exponentiation operator⟩ in a line of text are to be singled out. For example, an input statement such as

ANSWER+15.5*G+H**F/12−Q

would break down to the following table:

ANSWER	⟨identifier⟩
+	⟨adding operator⟩
15.5	⟨literal constant⟩
*	⟨multiplying operator⟩
G	⟨identifier⟩
+	⟨adding operator⟩
H	⟨identifier⟩
**	⟨exponentiation operator⟩
F	⟨identifier⟩
/	⟨multiplying operator⟩
12	⟨literal constant⟩
−	⟨adding operator⟩
Q	⟨identifier⟩

A PL/I program which performs this scanning is left as an exercise.

EXERCISES 3-3

1 Obtain a grammar which will generate the language $L = \{xx$ where $x = x_1x_2 \cdots x_n$ and $x_i \in \{a, b\}$ for all $1 \le i \le n\}$. For example, if $x = ab$, then $abab$ is in the language.

2 Consider the following grammar with the set of terminal symbols $\{a, b\}$:

$$S \rightarrow a \qquad S \rightarrow Sa \qquad S \rightarrow b \qquad S \rightarrow bS$$

Describe (in a closed form) the set of strings generated by the grammar.

3 Write grammars for the following languages:
(a) The set of nonnegative odd integers
(b) The set of nonnegative even integers with no leading zeros permitted

4 Give a grammar which generates

$$L = \{w \mid w \text{ consists of an equal number of } a\text{'s and } b\text{'s}\}$$

5 Give a context-free grammar which generates

$$L = \{w \mid w \text{ contains twice as many 0s as 1s}\}$$

6 Construct a regular grammar which will generate all strings of 0s and 1s having both an odd number of 0s and an odd number of 1s.

7 Obtain a grammar for the language

$$L = \{0^i 1^j \mid i \neq j \quad \text{and} \quad i, j > 0\}$$

8 Obtain a context-sensitive grammar for the language $\{a^{m^2} \mid m \geq 1\}$.

9 Construct a context-sensitive grammar for the language $\{w \mid w \in \{a, b, c\}^* - \{\Lambda\},$ where w contains the same number of a's, b's and c's$\}$.

4-6 FINITE-STATE MACHINES

In Secs. 1-2.15, 4-4, and 4-5 we have discussed switching circuits in which the outputs at a particular instant in time were functions of only the inputs at that time. Such circuits were called combinational circuits. In most digital computers, however, a number of circuits (or elements) are required which enable operations to be performed in a sequential manner. This sequencing of operations is achieved by means of a timed sequence of clock signals. The outputs of these circuits at any given time are functions of the external inputs and the stored information in the computer at that time. Such circuits are called sequential circuits. A computer can be viewed as a network consisting of a finite set of elements. Each of these elements can be in only one of a finite number of states at any one time, so that we may also consider a computer as a network consisting of a finite set of states.

The first subsection briefly introduces the concept of a sequential circuit. The finite-state machine is then introduced as an abstract model for sequential circuits. For reasons such as costs, reliability, etc., it is desirable to obtain a reduced or minimal machine which is equivalent to a given (and usually not minimal) machine. In pursuit of this goal, the second subsection gives a definition of equivalent machines and proceeds to develop an algorithm for determining a minimal machine which is equivalent to a given machine.

4-6.1 Introductory Sequential Circuits

Many sequential circuits are encountered in daily activities. Most people on their way to work encounter the sequential and usually predictable operation of traffic lights; the elevator control which causes the elevator to drop us at some point on the way up before taking on passengers on the way down; the sequential aspects of a combination lock which remembers the sequence of combinational digits; etc. We now proceed to give a concrete example of a sequential circuit for a serial adder.

In Sec. 4-5 a combinational circuit for a binary full-adder was discussed. Each bit position in a 32-bit number, for example, was made available at the same time. Therefore the adder had 64 external inputs available at one time. Let us now consider a binary adder which consists of two external inputs, each corresponding to a binary number. Each number consists of a sequence of bits, but the availability of the bits depends on time. An adder to perform the indicated operation necessarily operates in a sequential manner and is consequently called a *serial binary adder*. The block diagram of such an adder is given in Fig. 4-6.1, where the sequences of bits for x, y, and z are represented by $x_{n-1}x_{n-2}\cdots x_1x_0$, $y_{n-1}y_{n-2}\cdots y_1y_0$ and $z_{n-1}z_{n-2}\cdots z_1z_0$, respectively. The two inputs, x and y, and the output, z, each consists of n bits. The addition of x and y is to be done in a serial

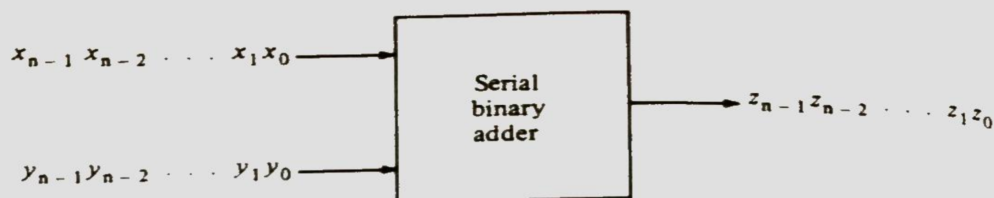

FIGURE 4-6.1 A block diagram of a serial adder.

manner; i.e., the least significant digits, x_0 and y_0, of the inputs arrive simultaneously at the input terminals at time t_0; a unit of time later (typically from 10^{-6} to 10^{-9} sec), the next least significant digits x_1 and y_1 arrive, etc. Finally, the bits x_{n-1} and y_{n-1} arrive at time t_{n-1}. The time interval between the arrival of pairs of input bits is controlled by a very precise clock signal. It will be assumed that the combinational circuit delay is insignificant as compared to the clock frequency. Accordingly, we may say that z_i appears at the output of the adder at the same time that x_i and y_i appear at the input terminals.

Consider the addition, according to Table 4-5.1, of the following binary numbers:

	t_4	t_3	t_2	t_1	t_0
$x =$	0	1	1	1	1
$y =$	0	1	0	1	0
$z =$	1	1	0	0	1

Note the difference between a combinational circuit and the serial adder. In the former case, the output at time t_i is determined from the inputs at that time, while in the latter circuit the output required at a particular time is different from that required at another even though the input combinations are the same. For example, at times t_1 and t_3 the inputs are both 1s, but the required outputs are $z_1 = 0$ and $z_3 = 1$, respectively. A similar situation exists for the input combination at times t_0 and t_2. From these observations, it is evident that the output of the serial adder cannot be specified as merely a function of its inputs.

It is obvious that the output of the adder at time t_i is a function of its inputs x_i and y_i at that time and of the carry bit which was generated at time t_{i-1}. This carry bit will depend on the inputs at time t_{i-2}, and so on. Therefore the serial adder must be able to remember or store information regarding its inputs from time t_0 to t_{i-1}. It is impractical to store all the previously encountered input bits, and we therefore try to establish a relationship between the inputs x_i and y_i and the output z_i.

Observe that in the serial adder there are two different cases arising from past input histories; the first case involves a carry bit of 0, and the second a carry bit of 1. These cases represent the states that the adder can be in at any given time. Of course, the circuit can be in only one state at one time. By having the adder remember the carry digit from the previous time interval, the adder has essentially remembered all its past inputs.

Let s_0 and s_1 denote the state of the adder at time t_i if a carry of 0 and 1 was generated at time t_{i-1}, respectively. We shall denote the *present state* of the adder

Table 4-6.1

Present state	$x_iy_i =$	Next-state function: Input symbols 00	01	11	10	Output function: Input symbols 00	01	11	10
s_0		s_0	s_0	s_1	s_0	0	1	0	1
s_1		s_0	s_1	s_1	s_1	1	0	1	0

as the state of the circuit when the present inputs are applied to the input terminals. The *next state* of the adder is defined as the state to which it goes after the present inputs and the previous carry bit have been examined. The output z_i at time t_i will then be a function of the inputs x_i and y_i and the state of the adder at time t_i. Also, the next state of the adder depends only on the present inputs and the carry bit (denoted by the present state). The behavior of the serial adder can be conveniently described by means of a table as shown in Table 4-6.1.

The table consists of two functions—the next-state function and the output function. Each row of the table corresponds to a state in the adder, and every column for each function corresponds to a combination of inputs. The entries for the next-state function denotes a state transition, and the corresponding entries in the output function denote the output symbols written on the output tape. For example, if the adder is in state s_0, that is, the present carry bit is 0, and it receives the input combination $x_iy_i = 11$, then the adder will go to state s_1, indicating that a carry bit of 1 has been set, and an output bit $z_i = 0$ is generated. The other entries of the table can be interpreted in a similar manner. It now becomes evident that the table completely specifies the operation of the serial adder.

We can easily implement such an adder if some storage device to represent the presence or absence of a carry is used. A number of such devices are available, but we shall only mention one—the unit delay element whose delay is equal to the time interval between two successive clock pulses. The state of the

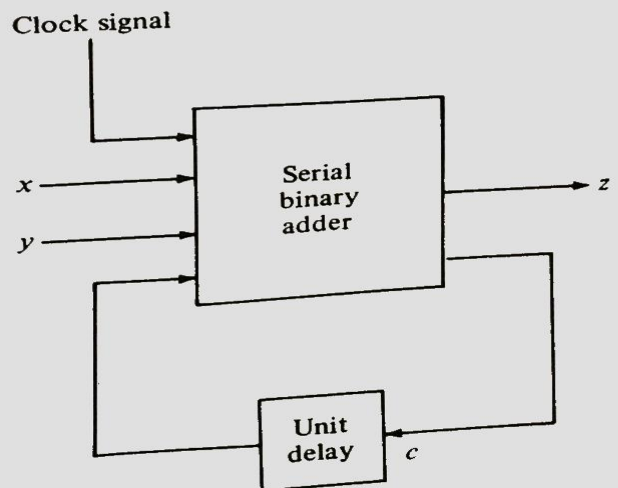

FIGURE 4-6.2 A serial binary adder.

delay element will be represented by c, and it will assume a value of 0 or 1, corresponding to the absence or presence of a carry bit. Since the present value at the input of the unit delay at time t_i is equal to its output at time t_{i+1}, this input is called the next state of the delay element. A block-diagram representation of the adder is given in Fig. 4-6.2, where an additional input line denoting the clock signal is also included. This additional input indicates that the serial adder is a synchronous circuit and that all events occur at discrete points in time.

We have given a very brief introduction to sequential circuits. In the next subsection we formalize these ideas and proceed with the development of an algorithm for determining a minimal equivalent sequential circuit for a given sequential circuit.

4-6.2 Equivalence of Finite-state Machines

In the previous subsection the basic notions of sequential circuits were introduced. We now wish to formalize these concepts by defining a finite-state machine. Furthermore, the important question of equivalent machines is discussed. In particular, we shall define what is meant by equivalent machines and show that for any given machine there exists a minimal equivalent machine. This reduced machine is homomorphic to the given machine. An algorithm for obtaining a minimal machine is developed. Finite-state machines have interesting algebraic properties based on the theory of semigroups, but we will not be concerned with such properties in this subsection. Finite-state machines can do many things, but there are certain operations such as multiplication which are beyond their range. We now proceed to the definition of a finite-state machine.

Definition 4-6.1 A *sequential machine*, or *finite-state machine*, is a system $N = \langle I, S, O, \delta, \lambda \rangle$, where the finite sets I, S, and O are alphabets that represent the input, state, and output symbols of the machine respectively. The alphabets I and O are not necessarily disjoint, but $I \cap S = O \cap S = \varnothing$. We shall denote the alphabets by

$$I = \{a_0, a_1, \ldots, a_n\} \qquad S = \{s_0, s_1, \ldots, s_m\} \qquad O = \{o_0, o_1, \ldots, o_r\}$$

δ is a mapping of $S \times I \to S$ which denotes the next-state function, and λ is a mapping $S \times I \to O$ which denotes the output function. We assume that the machine is in an initial state s_0.

Formally, a finite-state machine therefore consists of three not necessarily distinct alphabets and two functions. An abstract representation of a finite-state machine is given in Fig. 4-6.3. The machine reads a sequence of input symbols that are stored on an *input tape* and stores a sequence of output symbols on an *output tape*. Let the machine be in some state s_i and reading the input symbol a_p under its reading head. The mapping λ is then applied to s_i and a_p, thus causing the writing head to record a symbol o_k on the output tape. The function δ then causes the machine to go into state s_j. The machine proceeds to read the next input symbol and continues its operation until all symbols on the input tape are processed. Observe that the tapes are allowed to move only in one direction. In Fig. 4-6.3 the input symbols are processed from left to right. It should be

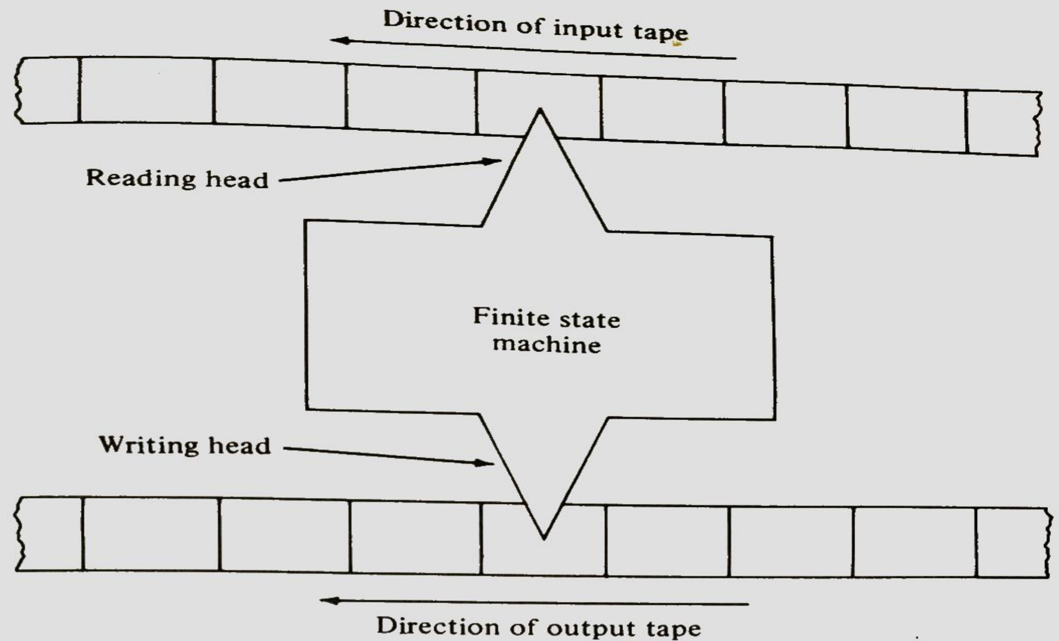

FIGURE 4-6.3 Model of a finite-state machine.

noted, however, that in the serial-adder example the input was assumed to have been read right to left. This is due to the positional property of fixed-radix number systems.

The mappings δ and λ are defined for all ordered pairs in $S \times I$, and in such a case the finite-state machine is said to be *completely specified*. A machine of this type, when starting in a given initial state s_0, will read the input tape and write a uniquely determinable sequence of symbols on the output tape.

The serial-adder example given in the previous subsection can be described within the framework of this definition as

$$I = \{00, 01, 11, 10\} \qquad S = \{s_0, s_1\} \qquad 0 = \{0, 1\}$$

$$\delta = \{\langle s_0, 00, s_0\rangle, \langle s_0, 01, s_0\rangle, \langle s_0, 11, s_1\rangle, \langle s_0, 10, s_0\rangle, \langle s_1, 00, s_0\rangle, \langle s_1, 01, s_1\rangle, \langle s_1, 11, s_1\rangle, \langle s_1, 10, s_1\rangle\}$$

$$\lambda = \{\langle s_0, 00, 0\rangle, \langle s_0, 01, 1\rangle, \langle s_0, 11, 0\rangle, \langle s_0, 10, 1\rangle, \langle s_1, 00, 1\rangle, \langle s_1, 01, 0\rangle, \langle s_1, 11, 1\rangle, \langle s_1, 10, 0\rangle\}$$

The starting state of the machine is s_0.

As a second example, consider the construction of a finite-state machine which will operate like a pulse divider. The pulse divider is to have as input a sequence of bits. The output, which is also a sequence of bits, is to be determined as follows:

> A 1 is to be written out at time t if and only if the number of 1s on the input tape up to time t is a nonzero even number.

The specification of a machine to perform this task is given in Table 4-6.2, and the initial state is assumed to be s_0. Suppose that the input tape contains the string 101011. The machine starts in state s_0, and when it reads the first input symbol 1, it goes to state s_1 and writes 0 on the output tape. When the second

Table 4-6.2

	δ		λ	
Present state ↓	**Input symbols** 0	1	**Input symbols** 0	1
s_0	s_0	s_1	0	0
s_1	s_1	s_0	0	1

input symbol 0 is read, the machine remains in state s_1. The third symbol causes the machine to enter s_0 and output a 1. This process is continued until the end of the input tape is encountered. It is easily verified that the input 101011 is transformed to the output 001001.

It is frequently convenient to use a directed graph instead of describing the functions δ and λ by means of a table called the *transition table*, as was just done. In the design of a finite-state machine, it is usually easier to formulate a design based on a graph rather than one based on a state table. The graph associated with a machine is called its transition diagram.

> **Definition 4-6.2** The *transition diagram* of a finite-state machine M is a directed graph in which there is a node for each state symbol in S, and each node is labeled by the state symbol with which it is associated. Furthermore, for each ordered pair $\langle s_i, s_j \rangle$ such that there exist the 3-tuples $\langle s_i, a_p, s_j \rangle$ and $\langle s_i, a_p, o_k \rangle$, there is a branch originating at node s_i and terminating at node s_j, and each such branch is labeled by the pair a_p/o_k.

As an example, the transition diagram of our serial adder is shown in Fig. 4-6.4.

Any finite-state machine can be viewed as a "black box" which reads input words and generates output words. It is natural to think of a finite-state machine as a device which transforms words into words. More precisely, if I^* and O^* are the sets of words on the input and output alphabets, I and O, respectively, the operation of the machine can be described by the function $g: I^* \to O^*$. Observe that this function has an infinite domain and range. Since machine equivalence

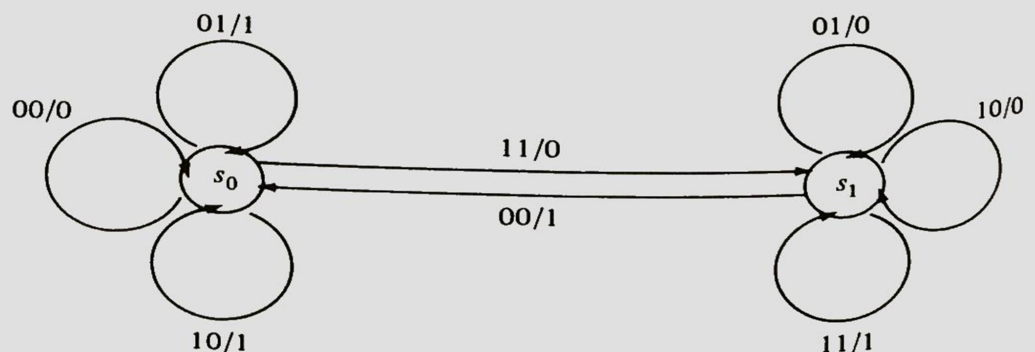

FIGURE 4-6.4 Transition diagram for serial adder.

can be described in terms of such a mapping, we proceed to generalize the functions δ and λ.

Consider a sequence $x = x_0x_1x_2 \cdots$ of input symbols. Let s_0 be the initial state of the machine. The state s_1 of the machine for the input x_0 is given by

$$s_1 = \delta(s_0, x_0) = \delta_1(s_0, x_0)$$

where $\delta = \delta_1: S \times I \to S$. The reason for introducing δ_1 will become obvious as we proceed. Next, consider the change in state due to the second input symbol x_1; the next state s_2 is given by

$$s_2 = \delta(s_1, x_1) = \delta(\delta_1(s_0, x_0), x_1) = \delta_2(s_0, x_0x_1)$$

where $\delta_2: S \times I^2 \to S$. The next state after the input x_2 is

$$s_3 = \delta(s_2, x_2) = \delta(\delta_2(s_0, x_0x_1), x_2) = \delta_3(s_0, x_0x_1x_2)$$

where $\delta_3: S \times I^3 \to S$. Continuing this procedure, one can now define a function $\delta_n: S \times I^n \to S$ such that

$$s_n = \delta_n(s_0, x_0x_1 \cdots x_{n-1}) = \delta(\delta_{n-1}(s_0, x_0x_1 \cdots x_{n-2}), x_{n-1})$$

Similarly, the output symbols $o_0, o_1, \ldots$ can also be described with the help of the function λ, such that

$$\begin{aligned}
o_0 &= \lambda(s_0, x_0) = \lambda_1(s_0, x_0) \\
o_1 &= \lambda(s_1, x_1) = \lambda(\delta_1(s_0, x_0), x_1) = \lambda_2(s_0, x_0x_1) \\
o_2 &= \lambda(s_2, x_2) = \lambda(\delta_2(s_0, x_0x_1), x_2) = \lambda_3(s_0, x_0x_1x_2) \\
&\vdots \\
o_{n-1} &= \lambda(s_{n-1}, x_{n-1}) = \lambda(\delta_{n-1}(s_0, x_0x_1 \cdots x_{n-2}), x_{n-1}) \\
&= \lambda_n(s_0, x_0x_1 \cdots x_{n-1})
\end{aligned}$$

Continuing in this manner, we can therefore define a function $\lambda_n: S \times I^n \to O$.

There is no likelihood of any ambiguity if we suppress the subscript n from δ_n and λ_n and write these functions as δ and λ. It would be clear from the length of the string as to what function is involved.

Definition 4-6.3 Let $x = x_0x_1 \cdots x_{n-1}$ be any input sequence containing n symbols, and let a be any input symbol. Then the mappings δ and λ can be extended recursively as follows:

(a) $\delta(s_i, xa) = \delta(\delta(s_i, x), a)$
(b) $\lambda(s_i, xa) = \lambda(\delta(s_i, x), a)$
(c) $g(s_i, x_0x_1 \cdots x_{n-1}) = \lambda(s_i, x_0)\, \lambda(s_i, x_0x_1) \cdots \lambda(s_i, x_0x_1 \cdots x_{n-1})$

As an example, consider the pulse-divider example of Table 4-6.2 when it is subjected to an input sequence of 11011. Computing the output function g, starting in state s_0, we have

$$\begin{aligned}
g(s_0, 11011) &= \lambda(s_0, 1)\lambda(s_0, 11)\lambda(s_0, 110)\lambda(s_0, 1101)\lambda(s_0, 11011) \\
&= \lambda(s_0, 1)\lambda(\delta(s_0, 1), 1)\lambda(\delta(s_0, 11), 0)\, \lambda(\delta(s_0, 110), 1)\lambda(\delta(s_0, 1101), 1) \\
&= \lambda(s_0, 1)\lambda(\delta(s_0, 1), 1)\lambda(\delta(\delta(s_0, 1), 1), 0)\, \lambda(\delta(\delta(\delta(s_0, 1), 1), 0), 1) \\
&\quad \lambda(\delta(\delta(\delta(\delta(s_0, 1), 1), 0), 1), 1) \\
&= 01001
\end{aligned}$$

Since our ultimate goal in this subsection is to develop an algorithm for obtaining an equivalent minimal machine for some given machine, we first formulate what is meant by equivalent states. Intuitively, two states are equivalent iff they produce the same output for any input sequence.

Definition 4-6.4 Let $M = \langle I, S, O, \delta, \lambda \rangle$ be a finite-state machine. Two states $s_i, s_j \in S$ are said to be *equivalent*, written $s_i \equiv s_j$, iff $\lambda(s_i, x) = \lambda(s_j, x)$ for every word $x \in I^*$.

It is easy to show that the relation $\equiv$ is an equivalence relation.

Theorem 4-6.1 Let s be any state in a finite-state machine and x and y be any words. Then

$$\delta(s, xy) = \delta(\delta(s, x), y) \qquad \text{and} \qquad \lambda(s, xy) = \lambda(\delta(s, x), y)$$

PROOF The proof will be by induction on the length of y. Let $y = a$. Then

$$\delta(s, xa) = \delta(\delta(s, x), a) \qquad \text{by Definition 4-6.3}a$$

Assume that the equation is true for any y of length n; that is, the induction hypothesis is

$$\delta(s, xy) = \delta(\delta(s, x), y)$$

We want to show that it is true for y having $n + 1$ symbols. From Definition 4-6.3a we can write

$$\delta(s, xya) = \delta(\delta(s, xy), a)$$

The right side of this identity can be rewritten as

$$\delta(\delta(s, xy), a) = \delta(\delta(\delta(s, x), y), a)$$

by the induction hypothesis. By letting $s' = \delta(s, x)$, the right side of this equation can be expressed as

$$\begin{aligned} \delta(\delta(\delta(s, x), y), a) &= \delta(\delta(s', y), a) \\ &= \delta(s', ya) \qquad \text{by Definition 4-6.3}a \\ &= \delta(\delta(s, x), ya) \end{aligned}$$

The other part of the proof can be obtained in a similar manner. ////

We next prove an important theorem which states that if two states are equivalent, then their next states will also be equivalent.

Theorem 4-6.2 If $s_i \equiv s_j$, then for any input sequence x, $\delta(s_i, x) \equiv \delta(s_j, x)$.

PROOF It is clear from Definition 4-6.3 that if $s_i \equiv s_j$, then $\lambda(s_i, xy) = \lambda(s_j, xy)$ for any input word xy. Moreover, from Theorem 4-6.1, we have

$$\lambda(\delta(s_i, x), y) = \lambda(\delta(s_j, x), y)$$

for any $y \in I^*$. It therefore follows by the definition of equivalence that

$$\delta(s_i, x) \equiv \delta(s_j, x)$$

////

Table 4-6.3

Present state ↓	δ Input symbols 0	δ Input symbols 1	λ Input symbols 0	λ Input symbols 1
s_0	s_5	s_3	0	1
s_1	s_1	s_4	0	0
s_2	s_1	s_3	0	0
s_3	s_1	s_2	0	0
s_4	s_5	s_2	0	1
s_5	s_4	s_1	0	1

Definition 4-6.4, which defines equivalent states in a finite-state machine, makes the identification of equivalent states virtually impossible. The definition implies that the equivalence of s_i and s_j can be determined by placing an input tape into the machine when it is in an initial state s_i, repeating the operation for an initial state s_j, and comparing the output tapes. Since the number of possible input words is infinite, this exhaustive procedure of comparing corresponding output words for every input word is endless. Therefore, a more realistic and efficient method for determining state equivalence must be used.

Definition 4-6.5 Let $M = \langle I, S, O, \delta, \lambda \rangle$ be a finite-state machine. Then for some positive integer k, s_i is said to be *k-equivalent* to s_j, that is, $s_i \overset{k}{\equiv} s_j$, if

$$s_i \overset{k}{\equiv} s_j \Leftrightarrow \lambda(s_i, x) = \lambda(s_j, x) \qquad \text{for all } x \text{ such that } |x| \leq k$$

Clearly, Definition 4-6.4 is a generalization of Definition 4-6.5 for all k, and hence $s_i \equiv s_j$ implies $s_i \overset{k}{\equiv} s_j$, but not conversely.

The k-equivalence relation is also an equivalence relation, and it defines a corresponding k-partition P_k on the set of states S whose k-equivalence is described as follows:

$$[s_i]_k = \{s_j \mid s_i \overset{k}{\equiv} s_j\} \qquad \text{and} \qquad P_k = \bigcup_{s \in S} [s]_k$$

As an example, consider the machine given in Table 4-6.3. Let us compute the partition P_1. It can be easily done by placing all states that have the same outputs for all input words of length 1 in the same equivalence class. From Table 4-6.3 we obtain

$$[s_0]_1 = \{s_0, s_4, s_5\} \qquad \text{and} \qquad [s_1]_1 = \{s_1, s_2, s_3\}$$

and $P_1 = \{\{s_0, s_4, s_5\}, \{s_1, s_2, s_3\}\}$.

Let P represent the partition generated by the equivalence relation $\equiv$; then we have the following.

Theorem 4-6.3 If for some integer k, $P_{k+1} = P_k$, then $P_k = P$ and conversely.

PROOF It is obvious that $P_k = P$ implies $P_{k+1} = P_k$ for any k; therefore the converse is established.

We next want to prove that $P_{k+1} = P_k \Rightarrow P_k = P$. This will be done by contradiction. Assume that $P_k \neq P$ and then show that $P_{k+1} \neq P_k$. By definition, there exist states s_i and s_j such that $s_i \overset{k}{\equiv} s_j$ but $s_i \not\equiv s_j$. Let q be the smallest integer such that $s_i \overset{q}{\not\equiv} s_j$. It is obvious that $q > k$. If $q = k + 1$, then $s_i \overset{k+1}{\not\equiv} s_j$, and consequently $P_{k+1} \neq P_k$.

Now suppose that $q > k + 1$. Let x be an input sequence of length q, and let $x = wy$ with w and y having lengths of $q - k - 1$ and $k + 1$, respectively. Because q was chosen to be the smallest integer such that $s_i \overset{q}{\not\equiv} s_j$, it directly follows that

$$\lambda(s_i, wy) \neq \lambda(s_j, wy)$$

Then by Theorem 4-6.1 with $s_i' = \delta(s_i, w)$ and $s_j' = \delta(s_j, w)$, we obtain

$$\lambda(s_i', y) \neq \lambda(s_j', y)$$

Since y contains $k + 1$ symbols, we have shown that $s_i' \overset{k+1}{\not\equiv} s_j'$.

Finally, assume that y consists of k or fewer symbols. Since q was the smallest integer violating the definition of k-equivalence, it follows that

$$\lambda(s_i, wy) = \lambda(s_j, wy)$$

Again by Theorem 4-6.1 we obtain $s_i' \overset{k}{\equiv} s_j'$. In summary, we have found two states s_i' and s_j' such that $s_i' \overset{k+1}{\not\equiv} s_j'$ and $s_i' \overset{k}{\equiv} s_j'$, and it therefore follows that $P_{k+1} \neq P_k$. ////

This theorem provides us with a simple algorithm for obtaining the partition of the states of a finite-state machine induced by the equivalence relation $\equiv$. The algorithm consists of the following steps.

1 Obtain P_1 from the output function (λ) for the finite-state machine.
2 $i \leftarrow 2$.
3 Obtain P_i from P_{i-1}.
4 If $P_i = P_{i-1}$, then Halt.
5 $i \leftarrow i + 1$, go to step 3.

The only step in this algorithm that we are unable to perform is step 3. The following theorem shows how to construct P_i from P_{i-1}.

Theorem 4-6.4 Let $s_i, s_j \in S$. Then $s_i \overset{k+1}{\equiv} s_j$ iff $s_i \overset{k}{\equiv} s_j$ and for all $a \in I$, $\delta(s_i, a) \overset{k}{\equiv} \delta(s_j, a)$.

PROOF Clearly $s_i \overset{k+1}{\equiv} s_j$ implies $s_i \overset{k}{\equiv} s_j$. We need only prove that

$$s_i \overset{k+1}{\equiv} s_j \qquad \text{if and only if} \qquad \delta(s_i, a) \overset{k}{\equiv} \delta(s_j, a)$$

By Definition 4-6.5, Theorem 4-6.1, and Theorem 4-6.2, this statement is equivalent to

$$\lambda(s_i, ax) = \lambda(s_j, ax) \qquad \text{if and only if} \qquad \delta(s_i, a) \overset{k}{\equiv} \delta(s_j, a)$$

where the string ax is of length $k + 1$. ////

Let us now apply the previous algorithm to the finite-state machine described in Table 4-6.3. We have already obtained $P_1 = \{\{s_0, s_4, s_5\}, \{s_1, s_2, s_3\}\}$. The next step is to obtain partition P_2 whose equivalence classes are 2-equivalent, i.e., equivalent for any input sequence of length 2. From Theorem 4-6.4, two states are 2-equivalent if and only if they are 1-equivalent and their next-state successors are 1-equivalent for all $a \in \{0, 1\}$. This step is performed by splitting equivalence classes in P_1 whenever their next-state successors do not all fall within an equivalence class of P_1. Therefore $P_2 = \{\{s_0, s_4, s_5\}, \{s_1\}, \{s_2, s_3\}\}$. Similarly, we can obtain P_3 by using the fact that two states are 3-equivalent if and only if they are 2-equivalent and their next-state successors are 2-equivalent for all $a \in \{0, 1\}$. Such a computation yields $P_3 = \{\{s_0, s_4\}, \{s_1\}, \{s_2, s_3\}, \{s_5\}\}$. This process is continued for one more step, giving $P_4 = P_3$. Therefore the required partition is

$$P = \{\{s_0, s_4\}, \{s_1\}, \{s_2, s_3\}, \{s_5\}\}$$

A question of considerable importance concerning this algorithm is whether the process will always terminate. It can be shown that for any finite-state machine with $n > 1$ states, there exists some integer $k \leq n - 1$ such that $P_k = P$. This proof is left as an exercise.

We have discussed at some length the notion of state equivalence. In the remainder of this subsection we will be concerned with machine equivalence.

Definition 4-6.6 Let $M = \langle I, S, O, \delta, \lambda \rangle$ and $M' = \langle I, S', O, \delta', \lambda' \rangle$ be finite-state machines. Then M and M' are *equivalent*, written $M \equiv M'$, iff for all $s_i \in S$ there exists an $s_j \in S'$ such that $s_i \equiv s_j$, and for all $s_j \in S'$ there exists an $s_i \in S$ such that $s_i \equiv s_j$.

Again it is easily shown that $\equiv$ is an equivalence relation.

Table 4-6.4 contains a finite-state machine which is equivalent to the machine given in Table 4-6.3. Observe that s_0' in M' is equivalent to s_0 and s_4 in M; s_1' in M' is equivalent to s_1 in M; s_2' in M' is equivalent to s_2 and s_3 in M; and s_3' in M' is equivalent to s_5 in M. Also note that the functions λ and λ' are the same for the indicated correspondence, but this is only a necessary condition for equivalence, not a sufficient one.

Table 4-6.4

	δ'		λ'	
Present state ↓	**Input symbols** **0**	**1**	**Input symbols** **0**	**1**
s_0'	s_3'	s_2'	0	1
s_1'	s_1'	s_0'	0	0
s_2'	s_1'	s_2'	0	0
s_3'	s_0'	s_1'	0	1

Definition 4-6.7 A finite-state machine $M = \langle I, S, O, \delta, \lambda \rangle$ is said to be *reduced* if and only if $s_i \equiv s_j$ implies that $s_i = s_j$ for all states $s_i, s_j \in S$.

In other words, a reduced finite-state machine is one in which each state is equivalent to itself and to no other. The partition of S in such a machine has all its equivalence classes consisting of a single element.

We shall now show how to construct a reduced finite-state machine M' which is equivalent to some given machine M. Let S in M be partitioned in a set of equivalence classes $[s]$ such that $P = \cup [s]$. Let the function ϕ be defined on the partition P such that $\phi([s]) = s'$, where s' is an arbitrary fixed element of $[s]$, called a representative. It is clear that $s' \equiv s$ in M. Let S' in M' be defined as

$$S' = \{s' \mid (\exists\, s)[s \in S \text{ and } \phi([s]) = s']\}$$

and let $I' = I$ and $O' = O$; that is, both machines will have the same input and output alphabets. The functions δ' and λ' are defined as follows:

$$\delta'(s', a) = \phi([\delta(s', a)])$$

and

$$\lambda'(s', a) = \lambda(s', a)$$

where s' is both in S and S'. Therefore the reduced machine is $M' = \langle I, S', O, \delta', \lambda' \rangle$.

Applying this procedure to the machine given in Table 4-6.3 gives the equivalent reduced machine in Table 4-6.4.

We shall now state a theorem without proof which shows the existence of a reduced equivalent machine.

Theorem 4-6.5 Let $M = \langle I, S, O, \delta, \lambda \rangle$ be a finite-state machine. Then there exists an equivalent machine M' with a set of states S' such that $S' \subseteq S$ and M' is reduced.

The idea of one finite-state machine simulating another is very important in a number of applications. This notion is formalized in the next definition.

Definition 4-6.8 Let $M = \langle I, S, O, \delta, \lambda \rangle$ and $M' = \langle I, S', O, \delta', \lambda' \rangle$ be two finite-state machines. Let function ϕ be a mapping from S into S'. A *finite-state homomorphism* is defined as

$$\left.\begin{aligned} \phi(\delta(s, a)) &= \delta'(\phi(s), a) \\ \lambda(s, a) &= \lambda'(\phi(s), a) \end{aligned}\right\} \quad \text{for all } a \in I$$

If ϕ is a one-one and onto function, then M is *isomorphic* to M'.

Finite-state machines are often used in compilers where they usually perform the task of a scanner (Sec. 3-3). The machine in such a case does lower-level syntax analysis such as identifying variable names, operators, constants, etc. A machine which performs this scanning task is called an *acceptor*. In Chap. 6 we show that the set of languages that can be recognized by an acceptor is exactly the set of those languages that can be generated by a regular grammar.

EXERCISES 4-6

1 Design a parity-check machine which is to read a sequence of 0s and 1s from an input tape. The machine is to output a 1 if the input tape contains an even number of 1s, or 0 otherwise.

2 Design a sequential machine which has one input line x in addition to a clock input and one output line z. z is to have a value of 0 unless the input contains three consecutive 1s or three consecutive 0s. z must be 1 at the time of the third consecutive identical input.

3 A sequential circuit is to be designed which will identify a particular sequence of inputs and provide an output to trigger a combinational lock. The inputs to the circuit are three switches labeled x_1, x_2, and x_3. The output of the circuit is to be 0 unless the input switches are in the position 010 where this position occurs at the conclusion of a sequence of input positions 101, 111, 011, 010. Design a sequential machine which will have only one output of 1 at the conclusion of the described sequence of switch positions. A correct sequence may begin every time the switches are set to 101.

4 The output of a sequential machine is to be 1 if and only if the last four input symbols are of the following form:

Time	t_i	t_{i+1}	t_{i+2}	t_{i+3}
Input symbol	1	0	1	1

Design such a machine.

5 Draw transition diagrams for the single-input, single-output sequential machines whose operations are specified as follows:

(*a*) An output $z = 1$ is to be written if on the input tape a 1 is preceded by exactly two 0s, for example, $\cdots$ 1001 $\cdots$

(*b*) An output $z = 1$ is to be produced if a 1 on the input tape follows two or more 0s.

6 Prove the second part of Theorem 4-6.1.

7 A serial parity-bit machine is to be designed. This machine is to receive coded messages and is to add a parity bit to every 4-bit message, so that the output of the machine is an error-detecting coded message. The machine is to have a single input consisting of strings of four symbols spaced apart by a single time unit. The parity bits are to be inserted at the appropriate spaces, so that the resulting output is a continuous sequence of symbols without spaces. Odd parity is to be used; i.e., a parity bit is inserted if and only if the number of 1s in the preceding four input symbols is even.

8 Reduce the following machine, if possible.

	δ		λ	
Present state ↓	**Input symbols** $x = 0$	**1**	**Input symbols** $x = 0$	**1**
s_0	s_1	s_7	0	0
s_1	s_7	s_0	0	1
s_2	s_8	s_7	0	1
s_3	s_7	s_5	0	1
s_4	s_3	s_2	0	0
s_5	s_6	s_7	0	0
s_6	s_8	s_5	0	1
s_7	s_3	s_7	0	1
s_8	s_2	s_0	0	1

9 Obtain a reduced machine for the machine obtained in Prob. 3.

10 Determine the reduced equivalent machine which corresponds to the machine described by the following table:

Present state ↓	δ Input symbols a	b	c	d	λ Input symbols a	b	c	d
s_0	s_4	s_2	s_1	s_4	1	0	1	1
s_1	s_2	s_5	s_4	s_1	0	1	1	0
s_2	s_1	s_0	s_3	s_5	1	0	1	1
s_3	s_6	s_5	s_4	s_1	0	1	1	0
s_4	s_2	s_5	s_3	s_4	0	1	1	0
s_5	s_2	s_5	s_3	s_7	1	1	0	0
s_6	s_3	s_0	s_1	s_5	1	0	1	1
s_7	s_1	s_2	s_4	s_5	1	0	1	1

11 Prove that $g(s_i, x) = g(s_j, x)$ for every x if and only if $\lambda(s_i, x) = \lambda(s_j, x)$, that is, if and only if $s_i \equiv s_j$.

12 Prove that if $P_k \neq P$, then the cardinality of P is greater than or equal to $k + 1$.

13 Prove that if a finite-state machine has n states where $n \geq 2$, then there exists an integer $k \leq n - 1$ such that $P_k = P$.

14 Prove Theorem 4-6.5.

6-1 FINITE-STATE ACCEPTORS AND REGULAR GRAMMARS

In Sec. 3-3, a generating device called a grammar was introduced as a means of specifying infinite languages. The importance of a grammar is twofold: first, it is a finite device, and second, the rules of the grammar impose structure on the strings of the language. Another method of finitely specifying languages is by using an acceptor or recognizer. An *acceptor* is a machine which can identify strings of a language. It is possible to define four classes of acceptors which correspond, in terms of specifying languages, to the four types of grammars introduced in Sec. 3-3. In this section we are concerned with the simplest type of acceptor—the finite-state acceptor (automaton). This class of machine is computationally equivalent to the family of finite-state machines introduced in Sec. 4-6. Finite-state machines are important, since the more complex machines, such as the Turing machine which is discussed in Sec. 6-2, are controlled by a finite-state machine.

It will be shown that the family of languages that can be generated by T_3 (regular) grammars contains precisely those languages that can be accepted by finite automata. In order to accomplish this task, a distinction will have to be made between a deterministic and a nondeterministic finite automaton. They are, however, equivalent in that they accept the same family of languages.

In Secs. 3-3 and 5-3 the membership question, i.e., whether a given string was in the language defined by a grammar, was discussed. We will now look at this question and concern ourselves with the following problems throughout the remainder of the section:

1 Given a regular grammar which generates a language, obtain a finite automaton which will accept exactly that language.

2 Conversely, given a machine which accepts a particular language, obtain a grammar which generates exactly this language.

Before attempting to solve these problems, we proceed to give a definition of a finite-state acceptor.

> **Definition 6-1.1** A *finite-state acceptor* or *finite automaton* M is a 5-tuple $\langle I, Q, q_0, \delta, F\rangle$, where I is a finite set of input symbols called the *input alphabet*, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state* of the machine, δ is a mapping of $Q \times I$ into Q, and $F \subseteq Q$ is a set of *final states*.

This machine is similar to the finite-state machine which was defined in Sec. 4-6.2 except that the finite automaton does not have an output alphabet; instead it has a set of acceptance states, F. The acceptor reads an input tape from left to right in a sequential manner. The finite automaton is, initially, in state q_0. The interpretation of $\delta(p, a) = q$, where $p, q \in Q$ and $a \in I$, is that M, in state p, scans the tape symbol a, moves its read head one position to the right, and enters state q.

As was done in Sec. 4-6.2, the domain of δ can be extended from $Q \times I$ to $Q \times I^*$ by defining a new mapping $\hat{\delta}$ as

$$\hat{\delta}(q, \Lambda) = q$$
$$\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a) \qquad \text{for every } x \in I^* \text{ and } a \in I$$

Since there is no chance of confusion, the function $\hat{\delta}$ will be simply written as δ in the remaining pages.

A string y is *accepted* by a finite automaton M if $\delta(q_0, y) = p$ for some $p \in F$. The set of all such y's accepted by M is called the *language accepted* by M and is denoted by $T(M)$, that is,

$$T(M) = \{y \mid \delta(q_0, y) \in F\}$$

The language accepted by a finite automaton is often called a *regular* language.

As an example, consider a finite-state acceptor that will accept the set of natural numbers x which are divisible by 3. A machine M to accomplish this recognition is $M = \langle I, Q, q_0, \delta, F\rangle$, where $I = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $Q = \{q_0, q_1, q_2\}$, $F = \{q_0\}$, and δ is defined as

$$\begin{array}{llll}
\delta(q_0, a) = q_0 & \delta(q_1, a) = q_1 & \delta(q_2, a) = q_2 & \text{for } a \in \{0, 3, 6, 9\} \\
\delta(q_0, b) = q_1 & \delta(q_1, b) = q_2 & \delta(q_2, b) = q_0 & \text{for } b \in \{1, 4, 7\} \\
\delta(q_0, c) = q_2 & \delta(q_1, c) = q_0 & \delta(q_2, c) = q_1 & \text{for } c \in \{2, 5, 8\}
\end{array}$$

The transition diagram for this machine is given in Fig. 6-1.1 where the final state q_0 is denoted by a double circle. The initial state is marked by an arrow. Observe that the edges have multiple labels. For example, the edge labeled "1, 4, 7" which originates at state q_0 and terminates at state q_1 is interpreted to mean that if the acceptor is in state q_0, then on scanning 1, or 4, or 7, it will enter state q_1. The other labeled edges can be interpreted in a similar manner.

Assuming an input string of 150, the computation

$$\begin{aligned}\delta(q_0, 150) &= \delta(\delta(\delta(q_0, 1), 5), 0) = \delta(\delta(q_1, 5), 0) \\ &= \delta(q_0, 0) = q_0 \quad \text{(accept)}\end{aligned}$$

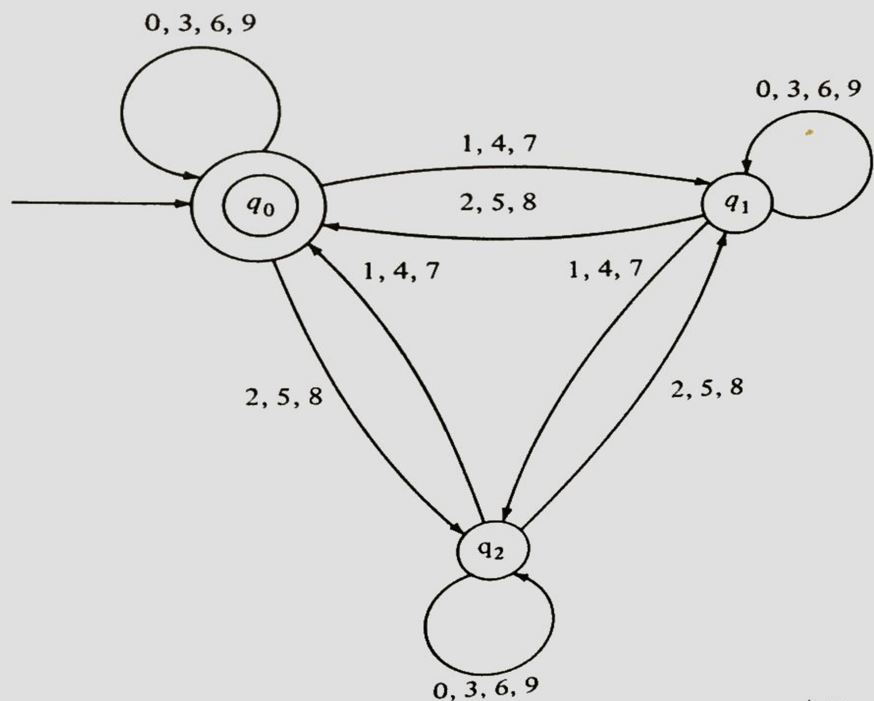

FIGURE 6-1.1 An example of a finite-state acceptor.

yields an acceptance state, while the string 136 is rejected since

$$\begin{aligned}\delta(q_0, 136) &= \delta(\delta(\delta(q_0, 1), 3), 6) = \delta(\delta(q_1, 3), 6)\\ &= \delta(q_1, 6) = q_1 \qquad \text{(reject state)}\end{aligned}$$

In this example let us define a relation R on the set I^* such that $x \, R \, y$ iff $\delta(q_0, x) = \delta(q_0, y)$. It is obvious that this relation is an equivalence relation. Therefore, R partitions the set I^* into three equivalence classes corresponding to the three states of the acceptor. The equivalence class associated with the final state is denoted by $[q_0]$ and consists of all natural numbers which are divisible by 3. A simple characterization can be given for the sets $[q_1]$ and $[q_2]$. Furthermore, if $x \, R \, y$, then we have for some $z \in I^*$ that $xz \, R \, yz$, since

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

It will be shown that the equivalence relation R will permit an association to be made between the states of an acceptor and the nonterminals of the grammar, providing that the grammar is derived in some way from the acceptor.

A precise form of the second recognition problem mentioned earlier will now be given. Assume that a finite automaton M, which accepts a language $T(M)$, is given. The input alphabet of this machine is the set V_T (the terminal alphabet of the language). We shall derive from this acceptor a grammar G such that $L(G) = T(M)$. That is, the sentences generated by the grammar are precisely those which cause M to reach an accepting state if it starts in its initial state.

The algorithm for finding the productions of a regular grammar which is equivalent to a given finite-state acceptor is as follows:

For an acceptor $M = \langle V_T, Q, q_0, \delta, F\rangle$,

1 If $\delta(q_i, a_{ij}) = q_j$, then construct the production $A_i \rightarrow a_{ij}A_j$.
2 If $q_j \in F$, then include the production $A_i \rightarrow a_{ij}$ for all i.

This simple procedure generates all rules of the required grammar. Observe that the rules thus obtained are indeed T_3 rules and also that the start symbol of the grammar is the initial state of the machine. We have essentially formed rewriting rules which are similar to the states of the machine.

The regular grammar obtained from Fig. 6-1.1 by using the algorithm is $G = \langle V_N, V_T, S, \Phi\rangle$, where $V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $V_N = \{A_0, A_1, A_2\}$, $S = A_0$, and Φ is the set

$$\begin{array}{lll}\{A_0 \rightarrow 1A_1, & A_1 \rightarrow 2A_0, & A_2 \rightarrow 1A_0,\\ A_0 \rightarrow 4A_1, & A_1 \rightarrow 5A_0, & A_2 \rightarrow 4A_0,\\ A_0 \rightarrow 7A_1, & A_1 \rightarrow 8A_0, & A_2 \rightarrow 7A_0,\\ A_0 \rightarrow 0A_0, & A_1 \rightarrow 0A_1, & A_2 \rightarrow 0A_2,\\ A_0 \rightarrow 3A_0, & A_1 \rightarrow 3A_1, & A_2 \rightarrow 3A_2,\\ A_0 \rightarrow 6A_0, & A_1 \rightarrow 6A_1, & A_2 \rightarrow 6A_2,\\ A_0 \rightarrow 9A_0, & A_1 \rightarrow 9A_1, & A_2 \rightarrow 9A_2,\\ A_0 \rightarrow 2A_2, & A_1 \rightarrow 1A_2, & A_2 \rightarrow 2A_1,\end{array}$$

Table 6-1.1

Input symbol	Present state	Sentential form
		A_0
1	q_0	$1A_1$
5	q_1	$15A_0$
0	q_0	$150A_0$
3	q_0	$1503A_0$
6	q_0	$15036A_0$
3	q_0	150363
	q_0	

$$A_0 \to 5A_2, \quad A_1 \to 4A_2, \quad A_2 \to 5A_1,$$
$$A_0 \to 8A_2, \quad A_1 \to 7A_2, \quad A_3 \to 8A_1,$$
$$A_0 \to 0, \quad A_1 \to 2, \quad A_2 \to 1,$$
$$A_0 \to 3, \quad A_1 \to 5, \quad A_2 \to 4,$$
$$A_0 \to 6, \quad A_1 \to 8, \quad A_2 \to 7,$$
$$A_0 \to 9\}$$

The nonterminals A_0, A_1, and A_2 of G correspond to the state symbols q_0, q_1, and q_2 respectively. Also, the start state of the grammar A_0 corresponds to the initial state q_0 of the machine. Table 6-1.1 illustrates the relationship between the operation of the acceptor and the derivation for the string 150363, which is a sentence in the language.

Assume that we wish to formulate the converse recognition problem. That is, given a grammar G, we want to obtain a machine M which has an input alphabet V_T (the same terminal alphabet as the grammar). Whenever a sentence is in $L(G)$ and the machine is started in its initial state, the machine should accept this sentence; otherwise, the machine should reject it.

A natural approach to solving this problem is to try to reverse the procedure given earlier. For example, if the grammar contains the rule $A_i \to aA_j$, then we could define $\delta(q_i, a) = q_j$. However, assume that the grammar also contains a rule $A_i \to aA_i$. We should also have a transition $\delta(q_i, a) = q_i$ with input a. If the machine is in state q_i and the input symbol a is read, it can either stay in q_i or enter a new state q_j; consequently there are two possible moves. This machine is nondeterministic. The machine that was defined in Definition 6-1.1 was deterministic; i.e., at each step in the acceptor's operation the input symbol and the current state uniquely determined the next state by using the δ mapping. This property was preserved for all moves made by the finite automaton. We shall therefore define a nondeterministic finite automaton and show that the class of languages accepted by such machines is exactly the same as that accepted by deterministic finite-state acceptors.

Definition 6-1.2 A *nondeterministic finite automaton* (acceptor) M is a 5-tuple $\langle I, Q, q_0, \delta, F\rangle$, where Q is a finite set of states, I is a finite *input alphabet*, $q_0 \in Q$ is the *initial state*, δ is a mapping of $Q \times I$ into subsets of Q, and $F \subseteq Q$ is the set of *final states*.

An important distinction between a deterministic and a nondeterministic acceptor should be made. In the case of the latter, $\delta(q, a)$ is a (perhaps empty) set of states, while in the former it is a single state. The meaning of $\delta(q, a) = \{p_1, p_2, \ldots, p_n\}$ is that the nondeterministic machine, when in state q and scanning the symbol a, moves right and chooses any one of $p_1, p_2, \ldots, p_n$ as its next state.

The domain of the mapping δ can be extended to $Q \times I^*$, as done earlier, by defining

$$\hat{\delta}(q, \Lambda) = \{q\}$$

and
$$\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a) \qquad \text{for each } x \in I^* \text{ and } a \in I$$

Furthermore, $\hat{\delta}$ can be extended to the domain $2^Q \times I^*$ by defining

$$\hat{\hat{\delta}}(\{p_1, p_2, \ldots, p_n\}, x) = \bigcup_{i=1}^{n} \hat{\delta}(p_i, x)$$

We will simply denote $\hat{\hat{\delta}}$ as δ in the remaining discussion. A sentence x is said to be *accepted* by the acceptor M if there exists some state in both F and $\delta(q_0, x)$. That is,

$$T(M) = \{x \mid \delta(q_0, x) \cap F \neq \varnothing\}$$

As an example of a nondeterministic finite automaton, we shall construct a machine that will accept the set of strings in $\{a, b, c\}^*$ such that the last symbol in the input string also appears earlier in the string. For example, *bab* is accepted, but *cbbca* is not. State q_0 will denote the initial state of the machine. States q_1, q_2, and q_3 are "guess" states, and the final state is q_4. A state diagram for the machine is given in Fig. 6-1.2. For δ defined by Table 6-1.2, the machine is given as

$$M = \langle \{a, b, c\}, \{q_0, q_1, q_2, q_3, q_4\}, q_0, \delta, \{q_4\} \rangle$$

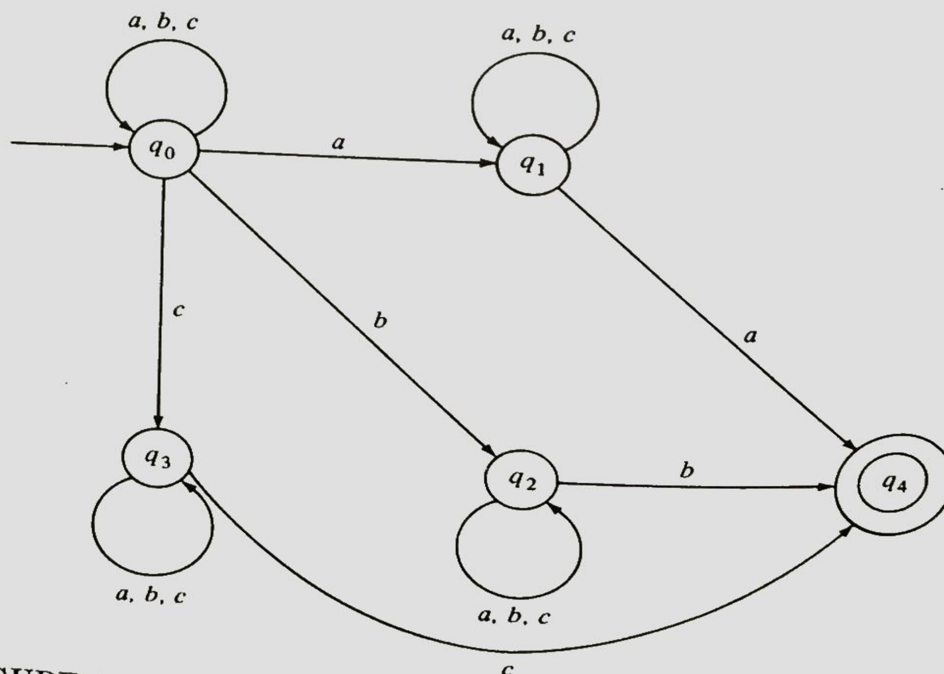

FIGURE 6-1.2 A transition diagram of a nondeterministic acceptor.

Table 6-1.2 NEXT-STATE MAPPING FOR NONDETERMINISTIC ACCEPTOR

Present state	Input symbol a	b	c
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
q_1	$\{q_1, q_4\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_4\}$	$\{q_2\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$
q_4	$\varnothing$	$\varnothing$	$\varnothing$

On input aca, the value of $\delta(q_0, aca)$ is obtained in the following manner:

$$\begin{aligned}
\delta(q_0, aca) &= \delta(\delta(q_0, a), ca) = \delta(\{q_0, q_1\}, ca) \\
&= \delta(q_0, ca) \cup \delta(q_1, ca) \\
\delta(q_0, ca) &= \delta(\delta(q_0, c), a) = \delta(\{q_0, q_3\}, a) \\
&= \delta(q_0, a) \cup \delta(q_3, a) \\
\delta(q_0, a) &= \{q_0, q_1\} \\
\delta(q_3, a) &= \{q_3\} \\
\delta(q_1, ca) &= \delta(\delta(q_1, c), a) = \delta(q_1, a) \\
\delta(q_1, a) &= \{q_1, q_4\}
\end{aligned}$$

Therefore,

$$\begin{aligned}
\delta(q_0, aca) &= \delta(q_0, a) \cup \delta(q_3, a) \cup \delta(q_1, a) \\
&= \{q_0, q_1\} \cup \{q_3\} \cup \{q_1, q_4\} \\
&= \{q_0, q_1, q_3, q_4\}
\end{aligned}$$

and since $\delta(q_0, aca) \cap \{q_4\} \neq \varnothing$, the sentence aca is accepted.

We wish to emphasize that although a nondeterministic machine doesn't have unique moves, it does not contain a random device which chooses its moves. Rather than making some guess and possibly obtaining the wrong answer to the membership question for a given input string, the acceptor explores all possible move sequences that the input string could cause. If at least one of these sequences leads to an acceptance state, then the input string is in the language.

The question that naturally arises at this point is, Are nondeterministic finite-state acceptors more powerful than deterministic finite-state acceptors? The answer to this question is contained in the following theorem.

Theorem 6-1.1 Let a language L be accepted by a nondeterministic finite-state acceptor. Then there exists an equivalent deterministic finite-state acceptor that accepts L.

PROOF We shall give a constructive proof. Let $M = \langle I, Q, q_0, \delta, F\rangle$ be a nondeterministic finite-state acceptor with n states that accepts L, that is, $T(M) = L$. Let us define a deterministic finite-state acceptor $M' = \langle I, Q', q_0', \delta', F'\rangle$ in the following way. The states of M' are all the possible nonempty subsets of the set of states for M. That is, M' has $2^n - 1$ states which will permit

it to simulate all the states in which M could be at any particular point in time. The state symbols in Q' will be denoted by $[q_0]$, $[q_1]$, ..., $[q_{n-1}]$, $[q_0, q_1]$, $[q_0, q_2]$, ..., $[q_0, q_{n-1}]$, $[q_1, q_2]$, ..., $[q_1, q_{n-1}]$, ..., $[q_{n-2}, q_{n-1}]$, ..., $[q_0, q_1, q_2]$, ..., $[q_{n-3}, q_{n-2}, q_{n-1}]$, ..., $[q_0, q_1, \ldots, q_{n-1}]$.

Both machines have the same alphabet and $q_0' = [q_0]$. We define the mapping δ' as follows:

$$\delta'([q_1, q_2, \ldots, q_j], a) = [p_1, p_2, \ldots, p_i]$$

if and only if

$$\delta(\{q_1, q_2, \ldots, q_j\}, a) = \{p_1, p_2, \ldots, p_i\}$$

That is, δ', when applied to an element of Q', is evaluated by applying δ to each state $q_1, q_2, \ldots, q_j$ in M and by then taking the union to yield a new set of states $\{p_1, p_2, \ldots, p_i\}$. This new set of states has a corresponding element $[p_1, p_2, \ldots, p_i]$ in Q', and this particular element is the value of $\delta'([q_1, q_2, \ldots, q_j], a)$. Also, $\delta'(q_0', x) \in F'$ exactly when $\delta(q_0, x) \cap F \neq \varnothing$, that is, $F' = \{[p_1, p_2, \ldots, p_k] \mid \{p_1, p_2, \ldots, p_k\} \cap F \neq \varnothing\}$. By construction it is clear that M' is deterministic.

We must now show that $T(M') = T(M)$. It will be done by showing first that $T(M') \supseteq T(M)$ and second that $T(M') \subseteq T(M)$.

1 $T(M') \supseteq T(M)$

The mapping δ' is constructed so that Q' contains all possible states of M at the next move. Therefore M' simulates all possibilities that M could explore. Also, M' will reach an acceptance state whenever there exists at least one sequence of state transitions in M which leads to a final state in that machine. Thus M' will accept any sentence accepted by M.

2 $T(M') \subseteq T(M)$

The proof is left as an exercise. ////

As an example, let $M = \langle\{a, b\}, \{q_0, q_1, q_2\}, q_0, \delta, \{q_2\}\rangle$ be a nondeterministic finite-state acceptor, where δ is given as follows:

$$\delta(q_0, a) = \{q_0, q_1\} \qquad \delta(q_0, b) = \{q_2\}$$
$$\delta(q_1, a) = \{q_1\} \qquad \delta(q_1, b) = \{q_0\}$$
$$\delta(q_2, a) = \{q_0\} \qquad \delta(q_2, b) = \{q_1, q_2\}$$

We shall construct an equivalent deterministic finite-state acceptor, $M' = \langle\{a, b\}, Q', [q_0], \delta', F'\rangle$, which accepts $T(M)$ as follows: Q' contains all subsets of $\{q_0, q_1, q_2\}$ (except the empty set); that is, $Q' = \{[q_0], [q_1], [q_2], [q_0, q_1], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$. Since $\delta(q_0, a) = \{q_0, q_1\}$, then

$$\delta'([q_0], a) = [q_0, q_1]$$

Similarly,

$$\delta'([q_0], b) = [q_2] \qquad \delta'([q_1], a) = [q_1] \qquad \delta'([q_1], b) = [q_0]$$
$$\delta'([q_2], a) = [q_0] \qquad \delta'([q_2], b) = [q_1, q_2]$$

Table 6-1.3 TRANSITION FUNCTION FOR AN EQUIVALENT DETERMINISTIC ACCEPTOR

Present state	Input symbol a	b
$[q_0]$	$[q_0, q_1]$	$[q_2]$
$[q_1]$	$[q_1]$	$[q_0]$
$[q_2]$	$[q_0]$	$[q_1, q_2]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_0, q_1]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2]$	$[q_0, q_1]$	$[q_0, q_1, q_2]$

Let us now consider $\delta'([q_0, q_2], a)$. Since

$$\delta(\{q_0, q_1\}, a) = \delta(q_0, a) \cup \delta(q_1, a) = \{q_0, q_1\} \cup \{q_1\}$$
$$= \{q_0, q_1\}$$

then

$$\delta'([q_0, q_1], a) = [q_0, q_1]$$

Also, since

$$\delta(\{q_0, q_1\}, b) = \delta(q_0, b) \cup \delta(q_1, b) = \{q_0, q_2\}$$

then

$$\delta'([q_0, q_1], b) = [q_0, q_2]$$

The remainder of the mapping is easily obtained in a similar fashion and is summarized in Table 6-1.3. The set of final states for M' is given by

$$F' = \{[q_2], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$$

We are now able to obtain from a nondeterministic machine an equivalent deterministic one. We now return to the relationship between the class of languages generated by T_3 grammars and the class of languages accepted by finite automata.

Theorem 6-1.2 Let $G \langle V_N, V_T, S, \Phi \rangle$ be a T_3 grammar which generates the language $L(G)$. Then there exists a finite-state acceptor $M = \langle V_T, Q, S, \delta, F \rangle$ such that $T(M) = L(G)$.

PROOF The machine M that will be constructed is nondeterministic with $Q = V_N \cup \{X\}$, where $X \notin V_N$. The initial state of the acceptor is S (the start symbol of the grammar), and its final state is X. For each production of the grammar, construct the mapping δ in the following manner:

1 $A_j \in \delta(A_i, a)$ if there is a production $A_i \to aA_j$ in G.
2 $X \in \delta(A_i, a)$ if there is a production $A_i \to a$ in G.

The acceptor M, when processing a sentence x, simulates a derivation of x in the grammar G. We want to show that $T(M) = L(G)$. Let $x = a_1a_2 \cdots a_m$, $m \geq 1$, be in the language $L(G)$. Then there exists some derivation in G such that

$$S \Rightarrow a_1A_1 \Rightarrow \cdots \Rightarrow a_1a_2 \cdots a_{m-1}A_{m-1} \Rightarrow a_1a_2 \cdots a_m$$

Table 6-1.4

Present state	Input symbol a	b
q_0	$\{q_0, q_1\}$	$\{q_2\}$
q_1	$\{q_0\}$	$\{q_1\}$
q_2	$\{q_1\}$	$\{q_0, q_1\}$

for a sequence of nonterminals $A_1, A_2, \ldots, A_{m-1}$. From the construction of δ, it is clear that $\delta(S, a_1)$ contains A_1, $\delta(A_1, a_2)$ contains A_2, $\ldots$, and that $\delta(A_{m-1}, a_m)$ contains X. Therefore, $x \in T(M)$ since $\delta(S, x)$ contains X and $X \in F$.

Conversely, if $x \in T(M)$, then we can easily obtain a derivation in G which simulates the acceptance of x in M, thereby concluding that $x \in L(G)$. ////

It can be shown that finite-state acceptors can be designed to accept the union, intersection, complementation, concatenation, etc., of sets. This property carries over to the regular grammar as well. That is, if L_1 and L_2 are regular languages, so are $L_1 \cup L_2$, $L_1 \cap L_2$, $\sim L_1$, $L_1 \circ L_2$, L_1^*, etc. In context-free languages, the intersection or complementation of two context-free languages is not necessarily context-free.

EXERCISES 6-1

1 Design a deterministic finite-state acceptor for sentences in $\{a, b\}$ such that every a has a b immediately to its right.

2 From the acceptor obtained in the previous problem, construct a type 3 grammar which generates that language.

3 Find a deterministic finite-state acceptor equivalent to the nondeterministic one given as

$$M = \langle \{a, b\}, \{q_0, q_1, q_2\}, q_0, \delta, \{q_2\} \rangle$$

where δ is given by Table 6-1.4.

4 Complete the proof of part 2 of Theorem 6-1.1.