

Modeling Computation

13.1 Languages and Grammars

13.2 Finite-State Machines with Output

13.3 Finite-State Machines with No Output

13.4 Language Recognition

13.5 Turing Machines

Computers can perform many tasks. Given a task, two questions arise. The first is: Can it be carried out using a computer? Once we know that this first question has an affirmative answer, we can ask the second question: How can the task be carried out? Models of computation are used to help answer these questions.

We will study three types of structures used in models of computation, namely, grammars, finite-state machines, and Turing machines. Grammars are used to generate the words of a language and to determine whether a word is in a language. Formal languages, which are generated by grammars, provide models both for natural languages, such as English, and for programming languages, such as Pascal, Fortran, Prolog, C, and Java. In particular, grammars are extremely important in the construction and theory of compilers. The grammars that we will discuss were first used by the American linguist Noam Chomsky in the 1950s.

Various types of finite-state machines are used in modeling. All finite-state machines have a set of states, including a starting state, an input alphabet, and a transition function that assigns a next state to every pair of a state and an input. The states of a finite-state machine give it limited memory capabilities. Some finite-state machines produce an output symbol for each transition; these machines can be used to model many kinds of machines, including vending machines, delay machines, binary adders, and language recognizers. We will also study finite-state machines that have no output but do have final states. Such machines are extensively used in language recognition. The strings that are recognized are those that take the starting state to a final state. The concepts of grammars and finite-state machines can be tied together. We will characterize those sets that are recognized by a finite-state machine and show that these are precisely the sets that are generated by a certain type of grammar.

Finally, we will introduce the concept of a Turing machine. We will show how Turing machines can be used to recognize sets. We will also show how Turing machines can be used to compute number-theoretic functions. We will discuss the Church–Turing thesis, which states that every effective computation can be carried out using a Turing machine. We will explain how Turing machines can be used to study the difficulty of solving certain classes of problems. In particular, we will describe how Turing machines are used to classify problems as tractable versus intractable and solvable versus unsolvable.

13.1 Languages and Grammars

13.1.1 Introduction

Words in the English language can be combined in various ways. The grammar of English tells us whether a combination of words is a valid sentence. For instance, *the frog writes neatly* is a valid sentence, because it is formed from a noun phrase, *the frog*, made up of the article *the* and the noun *frog*, followed by a verb phrase, *writes neatly*, made up of the verb *writes* and the adverb *neatly*. We do not care that this is a nonsensical statement, because we are concerned only with the **syntax**, or form, of the sentence, and not its **semantics**, or meaning. We also note that the combination of words *swims quickly mathematics* is not a valid sentence because it does not follow the rules of English grammar.

The syntax of a **natural language**, that is, a spoken language, such as English, French, German, or Spanish, is extremely complicated. In fact, it does not seem possible to specify all the rules of syntax for a natural language. Research in the automatic translation of one language

to another has led to the concept of a **formal language**, which, unlike a natural language, is specified by a well-defined set of rules of syntax. Rules of syntax are important not only in linguistics, the study of natural languages, but also in the study of programming languages.

We will describe the sentences of a formal language using a grammar. The use of grammars helps when we consider the two classes of problems that arise most frequently in applications to programming languages: (1) How can we determine whether a combination of words is a valid sentence in a formal language? (2) How can we generate the valid sentences of a formal language? Before giving a technical definition of a grammar, we will describe an example of a grammar that generates a subset of English. This subset of English is defined using a list of rules that describe how a valid sentence can be produced. We specify that

1. a **sentence** is made up of a **noun phrase** followed by a **verb phrase**;
2. a **noun phrase** is made up of an **article** followed by an **adjective** followed by a **noun**,
or
3. a **noun phrase** is made up of an **article** followed by a **noun**;
4. a **verb phrase** is made up of a **verb** followed by an **adverb**, or
5. a **verb phrase** is made up of a **verb**;
6. an **article** is *a*, or
7. an **article** is *the*;
8. an **adjective** is *large*, or
9. an **adjective** is *hungry*;
10. a **noun** is *rabbit*, or
11. a **noun** is *mathematician*;
12. a **verb** is *eats*, or
13. a **verb** is *hops*;
14. an **adverb** is *quickly*, or
15. an **adverb** is *wildly*.

From these rules we can form valid sentences using a series of replacements until no more rules can be used. For instance, we can follow the sequence of replacements:

```

sentence
noun phrase  verb phrase
article  adjective  noun  verb phrase
article  adjective  noun  verb  adverb
the  adjective  noun  verb  adverb
the large noun  verb  adverb
the large rabbit verb  adverb
the large rabbit hops adverb
the large rabbit hops quickly

```

to obtain a valid sentence. It is also easy to see that some other valid sentences are: *a hungry mathematician eats wildly*, *a large mathematician hops*, *the rabbit eats quickly*, and so on. Also, we can see that *the quickly eats mathematician* is not a valid sentence.

13.1.2 Phrase-Structure Grammars

Before we give a formal definition of a grammar, we introduce a little terminology.

Definition 1

A *vocabulary* (or *alphabet*) V is a finite, nonempty set of elements called *symbols*. A *word* (or *sentence*) over V is a string of finite length of elements of V . The *empty string* or *null string*, denoted by λ (and sometimes by ϵ), is the string containing no symbols. The set of all words over V is denoted by V^* . A *language over V* is a subset of V^* .

Note that λ , the empty string, is the string containing no symbols. It is different from \emptyset , the empty set. It follows that $\{\lambda\}$ is the set containing exactly one string, namely, the empty string.


Languages can be specified in various ways. One way is to list all the words in the language. Another is to give some criteria that a word must satisfy to be in the language. In this section, we describe another important way to specify a language, namely, through the use of a grammar, such as the set of rules we gave in the introduction to this section. A grammar provides a set of symbols of various types and a set of rules for producing words. More precisely, a grammar has a **vocabulary** V , which is a set of symbols used to derive members of the language. Some of the elements of the vocabulary cannot be replaced by other symbols. These are called **terminals**, and the other members of the vocabulary, which can be replaced by other symbols, are called **nonterminals**. The sets of terminals and nonterminals are usually denoted by T and N , respectively. In the example given in the introduction of the section, the set of terminals is $\{a, \text{the}, \text{rabbit}, \text{mathematician}, \text{hops}, \text{eats}, \text{quickly}, \text{wildly}\}$, and the set of nonterminals is $\{\text{sentence}, \text{noun phrase}, \text{verb phrase}, \text{adjective}, \text{article}, \text{noun}, \text{verb}, \text{adverb}\}$. There is a special member of the vocabulary called the **start symbol**, denoted by S , which is the element of the vocabulary that we always begin with. In the example in the introduction, the start symbol is **sentence**. The rules that specify when we can replace a string from V^* , the set of all strings of elements in the vocabulary, with another string are called the **productions** of the grammar. We denote by $z_0 \rightarrow z_1$ the production that specifies that z_0 can be replaced by z_1 within a string. The productions in the grammar given in the introduction of this section were listed. The first production, written using this notation, is **sentence** \rightarrow **noun phrase verb phrase**. We summarize this terminology in Definition 2.

The notion of a phrase-structure grammar extends the concept of a rewrite system devised by Axel Thue in the early 20th century.

Definition 2

A *phrase-structure grammar* $G = (V, T, S, P)$ consists of a vocabulary V , a subset T of V consisting of terminal symbols, a start symbol S from V , and a finite set of productions P . The set $V - T$ is denoted by N . Elements of N are called *nonterminal symbols*. Every production in P must contain at least one nonterminal on its left side.


EXAMPLE 1

Let $G = (V, T, S, P)$, where $V = \{a, b, A, B, S\}$, $T = \{a, b\}$, S is the start symbol, and $P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\}$. G is an example of a phrase-structure grammar. 

We will be interested in the words that can be generated by the productions of a phrase-structure grammar.

Definition 3


Let $G = (V, T, S, P)$ be a phrase-structure grammar. Let $w_0 = lz_0r$ (that is, the concatenation of l , z_0 , and r) and $w_1 = lz_1r$ be strings over V . If $z_0 \rightarrow z_1$ is a production of G , we say that w_1 is *directly derivable* from w_0 and we write $w_0 \Rightarrow w_1$. If w_0, w_1, \dots, w_n are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \dots, w_{n-1} \Rightarrow w_n$, then we say that w_n is *derivable from w_0* , and we write $w_0 \Rightarrow^* w_n$. The sequence of steps used to obtain w_n from w_0 is called a *derivation*.

EXAMPLE 2 The string $Aaba$ is directly derivable from ABa in the grammar in Example 1 because $B \rightarrow ab$ is a production in the grammar. The string $abababa$ is derivable from ABa because $ABa \Rightarrow Aaba \Rightarrow BBaba \Rightarrow Bababa \Rightarrow abababa$, using the productions $B \rightarrow ab$, $A \rightarrow BB$, $B \rightarrow ab$, and $B \rightarrow ab$ in succession. 


Definition 4 Let $G = (V, T, S, P)$ be a phrase-structure grammar. The *language generated by G* (or the *language of G*), denoted by $L(G)$, is the set of all strings of terminals that are derivable from the starting state S . In other words,

$$L(G) = \{w \in T^* \mid S \xRightarrow{*} w\}.$$


In Examples 3 and 4 we find the language generated by a phrase-structure grammar.

EXAMPLE 3 Let G be the grammar with vocabulary $V = \{S, A, a, b\}$, set of terminals $T = \{a, b\}$, starting symbol S , and productions $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$. What is $L(G)$, the language of this grammar? 

Extra
Examples

Solution: From the start state S we can derive aA using the production $S \rightarrow aA$. We can also use the production $S \rightarrow b$ to derive b . From aA the production $A \rightarrow aa$ can be used to derive aaa . No additional words can be derived. Hence, $L(G) = \{b, aaa\}$. 

EXAMPLE 4 Let G be the grammar with vocabulary $V = \{S, 0, 1\}$, set of terminals $T = \{0, 1\}$, starting symbol S , and productions $P = \{S \rightarrow 11S, S \rightarrow 0\}$. What is $L(G)$, the language of this grammar?


Solution: From S we can derive 0 using $S \rightarrow 0$, or $11S$ using $S \rightarrow 11S$. From $11S$ we can derive either 110 or $1111S$. From $1111S$ we can derive 11110 and $111111S$. At any stage of a derivation we can either add two 1s at the end of the string or terminate the derivation by adding a 0 at the end of the string. We surmise that $L(G) = \{0, 110, 11110, 1111110, \dots\}$, the set of all strings that begin with an even number of 1s and end with a 0. This can be proved using an inductive argument that shows that after n productions have been used, the only strings of terminals generated are those consisting of $n - 1$ concatenations of 11 followed by 0. (This is left as an exercise for the reader.) 

The problem of constructing a grammar that generates a given language often arises. Examples 5, 6, and 7 describe problems of this kind.

EXAMPLE 5 Give a phrase-structure grammar that generates the set $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$.

Solution: Two productions can be used to generate all strings consisting of a string of 0s followed by a string of the same number of 1s, including the null string. The first builds up successively longer strings in the language by adding a 0 at the start of the string and a 1 at the end. The second production replaces S with the empty string. The solution is the grammar $G = (V, T, S, P)$, where $V = \{0, 1, S\}$, $T = \{0, 1\}$, S is the starting symbol, and the productions are

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow \lambda. \end{aligned}$$


The verification that this grammar generates the correct set is left as an exercise for the reader. 

Example 5 involved the set of strings made up of 0s followed by 1s, where the number of 0s and 1s are the same. Example 6 considers the set of strings consisting of 0s followed by 1s, where the number of 0s and 1s may differ.


EXAMPLE 6 Find a phrase-structure grammar to generate the set $\{0^m 1^n \mid m \text{ and } n \text{ are nonnegative integers}\}$.

Solution: We will give two grammars G_1 and G_2 that generate this set. This will illustrate that two different grammars can generate the same language.


The grammar G_1 has alphabet $V = \{S, 0, 1\}$; terminals $T = \{0, 1\}$; and productions $S \rightarrow 0S$, $S \rightarrow S1$, and $S \rightarrow \lambda$. Then, G_1 generates the correct set, because using the first production m times puts m 0s at the beginning of the string, and using the second production n times puts n 1s at the end of the string. Furthermore, every string produced by this grammar is of the form $0^m 1^n$, because the only way to add a symbol to the beginning of a string is to add a 0 by applying the production $S \rightarrow 0S$, and the only way to add a symbol to the end of the string is to add a 1 by applying the production $S \rightarrow S1$.

The grammar G_2 has alphabet $V = \{S, A, 0, 1\}$; terminals $T = \{0, 1\}$; and productions $S \rightarrow 0S$, $S \rightarrow 1A$, $S \rightarrow \lambda$, $A \rightarrow 1A$, $A \rightarrow 1$, and $S \rightarrow \lambda$. The details that this grammar generates the correct set are left as an exercise for the reader. 

Sometimes a set that is easy to describe can be generated only by a complicated grammar. Example 7 illustrates this.

EXAMPLE 7 One grammar that generates the set $\{0^n 1^n 2^n \mid n = 0, 1, 2, 3, \dots\}$ is $G = (V, T, S, P)$ with $V = \{0, 1, 2, S, A, B, C\}$; $T = \{0, 1, 2\}$; starting state S ; and productions $S \rightarrow C$, $C \rightarrow 0CAB$, $S \rightarrow \lambda$, $BA \rightarrow AB$, $0A \rightarrow 01$, $1A \rightarrow 11$, $1B \rightarrow 12$, and $2B \rightarrow 22$. We leave it as an exercise for the reader (Exercise 12) to show that this statement is correct. The grammar given is the simplest type of grammar that generates this set, in a sense that will be made clear later in this section. 

13.1.3 Types of Phrase-Structure Grammars

Links  Phrase-structure grammars can be classified according to the types of productions that are allowed. We will describe the classification scheme introduced by Noam Chomsky. In Section 13.4 we will see that the different types of languages defined in this scheme correspond to the classes of languages that can be recognized using different models of computing machines.

A **type 0** grammar has no restrictions on its productions. A **type 1** grammar can have productions of the form $w_1 \rightarrow w_2$, where $w_1 = lAr$ and $w_2 = lwr$, where A is a nonterminal symbol, l and r are strings of zero or more terminal or nonterminal symbols, and w is a nonempty string of terminal or nonterminal symbols. It can also have the production $S \rightarrow \lambda$ as long as S does not appear on the right-hand side of any other production. A **type 2** grammar can have productions only of the form $w_1 \rightarrow w_2$, where w_1 is a single symbol that is not a terminal symbol. A **type 3** grammar can have productions only of the form $w_1 \rightarrow w_2$ with $w_1 = A$ and either $w_2 = aB$ or $w_2 = a$, where A and B are nonterminal symbols and a is a terminal symbol, or with $w_1 = S$ and $w_2 = \lambda$.

Type 2 grammars are called **context-free grammars** because a nonterminal symbol that is the left side of a production can be replaced in a string whenever it occurs, no matter what else is in the string. A language generated by a type 2 grammar is called a **context-free language**. When there is a production of the form $lw_1r \rightarrow lw_2r$ (but not of the form $w_1 \rightarrow w_2$), the grammar is called type 1 or **context-sensitive** because w_1 can be replaced by w_2 only when it is surrounded by the strings l and r . A language generated by a type 1 grammar is called a **context-sensitive language**. Type 3 grammars are also called **regular grammars**. A language generated by a regular grammar is called **regular**. Section 13.4 deals with the relationship between regular languages and finite-state machines.

Of the four types of grammars we have defined, context-sensitive grammars have the most complicated definition. Sometimes, these grammars are defined in a different way. A production of the form $w_1 \rightarrow w_2$ is called **noncontracting** if the length of w_1 is less than or equal to the length of w_2 . According to our characterization of context-sensitive languages, every production in a type 1 grammar, other than the production $S \rightarrow \lambda$, if it is present, is noncontracting. It follows that the lengths of the strings in a derivation in a context-sensitive language are non-decreasing unless the production $S \rightarrow \lambda$ is used. This means that the only way for the empty string to belong to the language generated by a context-sensitive grammar is for the production $S \rightarrow \lambda$ to be part of the grammar. The other way that context-sensitive grammars are defined is by specifying that all productions are noncontracting. A grammar with this property is called **noncontracting** or **monotonic**. The class of noncontracting grammars is not the same as the class of context-sensitive grammars. However, these two classes are closely related; it can be shown that they define the same set of languages except that noncontracting grammars cannot generate any language containing the empty string λ .

EXAMPLE 8 From Example 6 we know that $\{0^m 1^n \mid m, n = 0, 1, 2, \dots\}$ is a regular language, because it can be generated by a regular grammar, namely, the grammar G_2 in Example 6. ◀

**Extra
Examples** ▶

Context-free and regular grammars play an important role in programming languages. Context-free grammars are used to define the syntax of almost all programming languages. These grammars are strong enough to define a wide range of languages. Furthermore, efficient algorithms can be devised to determine whether and how a string can be generated. Regular grammars are used to search text for certain patterns and in lexical analysis, which is the process of transforming an input stream into a stream of tokens for use by a parser.

EXAMPLE 9 It follows from Example 5 that $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$ is a context-free language, because the productions in this grammar are $S \rightarrow 0S1$ and $S \rightarrow \lambda$. However, it is not a regular language. This will be shown in Section 13.4. ◀

EXAMPLE 10 The set $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$ is a context-sensitive language, because it can be generated by a type 1 grammar, as Example 7 shows, but not by any type 2 language. (This is shown in Exercise 28 in the supplementary exercises at the end of the chapter.) ◀

Table 1 summarizes the terminology used to classify phrase-structure grammars.

13.1.4 Derivation Trees

A derivation in the language generated by a context-free grammar can be represented graphically using an ordered rooted tree, called a **derivation**, or **parse tree**. The root of this tree represents the starting symbol. The internal vertices of the tree represent the nonterminal symbols that

TABLE 1 Types of Grammars.

Type	Restrictions on Productions $w_1 \rightarrow w_2$
0	No restrictions
1	$w_1 = lAr$ and $w_2 = lwr$, where $A \in N$, $l, r, w \in (N \cup T)^*$ and $w \neq \lambda$; or $w_1 = S$ and $w_2 = \lambda$ as long as S is not on the right-hand side of another production
2	$w_1 = A$, where A is a nonterminal symbol
3	$w_1 = A$ and $w_2 = aB$ or $w_2 = a$, where $A \in N$, $B \in N$, and $a \in T$; or $w_1 = S$ and $w_2 = \lambda$

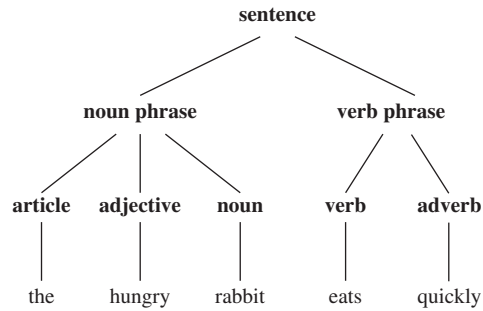


FIGURE 1 A derivation tree.

arise in the derivation. The leaves of the tree represent the terminal symbols that arise. If the production $A \rightarrow w$ arises in the derivation, where w is a word, the vertex that represents A has as children vertices that represent each symbol in w , in order from left to right.

EXAMPLE 11 Construct a derivation tree for the derivation of *the hungry rabbit eats quickly*, given in the introduction of this section.

Solution: The derivation tree is shown in Figure 1. ◀

The problem of determining whether a string is in the language generated by a context-free grammar arises in many applications, such as in the construction of compilers. When given a string, the naive approach for determining whether it is in the language generated by a grammar is to look for a sequence of productions that can be applied, beginning at the start state, that lead to the given string. When following such an approach, it is useful to think a few moves ahead. This approach is known as **top-down parsing**. A second approach, known as **bottom-up parsing**, is to work backward from the given string with the goal of undoing productions one-by-one to reach the start symbol. We illustrate these two approaches to this problem in Example 12.

EXAMPLE 12 Determine whether the word *cbab* belongs to the language generated by the grammar $G = (V, T, S, P)$, where $V = \{a, b, c, A, B, C, S\}$, $T = \{a, b, c\}$, S is the starting symbol, and the productions are

$$\begin{aligned}
 S &\rightarrow AB \\
 A &\rightarrow Ca \\
 B &\rightarrow Ba \\
 B &\rightarrow Cb \\
 B &\rightarrow b \\
 C &\rightarrow cb \\
 C &\rightarrow b.
 \end{aligned}$$

Solution: A top-down approach to this problem is to begin with S and attempt to derive *cbab* using a series of productions. Because there is only one production with S on its left-hand side, we must start with $S \Rightarrow AB$. Next we use the only production that has A on its left-hand side,

Links



©SASCHA SCHUERMANN/
AFP/Getty Images

AVRAM NOAM CHOMSKY (BORN 1928) Noam Chomsky, born in Philadelphia, is the son of a Hebrew scholar. He received his B.A., M.A., and Ph.D. in linguistics, all from the University of Pennsylvania. He was on the staff of the University of Pennsylvania from 1950 until 1951. In 1955 he joined the faculty at M.I.T., beginning his M.I.T. career teaching engineers French and German. Chomsky is currently the Ferrari P. Ward Professor of foreign languages and linguistics at M.I.T. He is known for his many fundamental contributions to linguistics, including the study of grammars. Chomsky is also widely known for his outspoken political activism.

namely, $A \rightarrow Ca$, to obtain $S \Rightarrow AB \Rightarrow CaB$. Because $cbab$ begins with the symbols cb , we use the production $C \rightarrow cb$. This gives us $S \Rightarrow AB \Rightarrow CaB \Rightarrow cbaB$. We finish by using the production $B \rightarrow b$, to obtain $S \Rightarrow AB \Rightarrow CaB \Rightarrow cbaB \Rightarrow cbab$.

We will solve the same problem using bottom-up parsing. Because $cbab$ is the string to be derived, we can use the production $C \rightarrow cb$, so that $Cab \Rightarrow cbab$. Then, we can use the production $A \rightarrow Ca$, so that $Ab \Rightarrow Cab \Rightarrow cbab$. Using the production $B \rightarrow b$ gives $AB \Rightarrow Ab \Rightarrow Cab \Rightarrow cbab$. Finally, using $S \rightarrow AB$ shows that a complete derivation for $cbab$ is $S \Rightarrow AB \Rightarrow Ab \Rightarrow Cab \Rightarrow cbab$. ▶

Example 12 presents the solution of a parsing problem using both top-down and bottom-up parsing. Both approaches were easy to use for that problem. However, parsing problems can be quite challenging. That is, it can be quite challenging to determine whether a string is in the language generated by a context-free grammar. Because parsing is so important, many strategies and algorithms have been devised for top-down and for bottom-up parsing. These algorithms are beyond the scope of this book. The interested reader should consult [AhLaSeUI06] to learn more.

13.1.5 Backus–Naur Form

Links ▶

There is another notation that is sometimes used to specify a type 2 grammar, called the **Backus–Naur form (BNF)**, after John Backus, who invented it, and Peter Naur, who refined

Links ▶



Courtesy of Louis Bachrach

JOHN BACKUS (1924–2007) John Backus was born in Philadelphia and grew up in Wilmington, Delaware. He attended the Hill School in Pottstown, Pennsylvania. He needed to attend summer school every year because he disliked studying and was not a serious student. But he enjoyed spending his summers in New Hampshire where he attended summer school and amused himself with summer activities, including sailing. He obliged his father by enrolling at the University of Virginia to study chemistry. But he quickly decided chemistry was not for him, and in 1943 he entered the army, where he received medical training and worked in a neuro-surgery ward in an army hospital. Ironically, Backus was soon diagnosed with a bone tumor in his skull and was fitted with a metal plate. His medical work in the army convinced him to try medical school, but he abandoned this after nine months because he disliked the rote memorization required. After dropping out of medical school, he entered a school for radio technicians because he wanted to build his own high fidelity set. A teacher in this school recognized his potential and asked him to help with some mathematical calculations needed for an article in a magazine. Finally, Backus found what he was interested in: mathematics and its applications. He enrolled at Columbia University, from which he received both bachelor's and master's degrees in mathematics. Backus joined IBM as a programmer in 1950. He participated in the design and development of two of IBM's early computers. From 1954 to 1958 he led the IBM group that developed FORTRAN. Backus became a staff member at the IBM Watson Research Center in 1958. He was part of the committees that designed the programming language ALGOL using what is now called the Backus–Naur form for the description of the syntax of this language. (The development of ALGOL led to the development of other programming languages, including Pascal and C.) Later, Backus worked on the mathematics of families of sets and on a functional style of programming. Backus became an IBM Fellow in 1963, and he received the National Medal of Science in 1974 and the prestigious Turing Award from the Association for Computing Machinery in 1977.



©Heidelberg Laureate Forum Foundation

PETER NAUR (1928–2016) Peter Naur was born in Frederiksberg, near Copenhagen. As a boy he became interested in astronomy. Not only did he observe heavenly bodies, but he also computed the orbits of comets and asteroids. Naur attended Copenhagen University, receiving his degree in 1949. He spent 1950 and 1951 in Cambridge, where he used an early computer to calculate the motions of comets and planets. After returning to Denmark, he continued working in astronomy but kept his ties to computing. In 1955 he served as a consultant to the building of the first Danish computer. In 1959 Naur made the switch from astronomy to computing as a full-time activity. His first job as a full-time computer scientist was participating in the development of the programming language ALGOL. From 1960 to 1967 he worked on the development of compilers for ALGOL and COBOL. From 1969 until 1999 he was professor of computer science at Copenhagen University, where he has worked in the area of programming methodology. His research interests included the design, structure, and performance of computer programs. Naur was a pioneer in both the areas of software architecture and software engineering. He rejected the view that computer programming is a branch of mathematics and preferred that computer science be called *datalogy*.

The ancient Indian grammarian Pāṇini specified Sanskrit using 3959 rules; Backus–Naur form is sometimes called Backus–Pāṇini form.

it for use in the specification of the programming language ALGOL. (Surprisingly, a notation quite similar to the Backus–Naur form was used approximately 2500 years ago to describe the grammar of Sanskrit.) The Backus–Naur form is used to specify the syntactic rules of many computer languages, including Java. The productions in a type 2 grammar have a single non-terminal symbol as their left-hand side. Instead of listing all the productions separately, we can combine all those with the same nonterminal symbol on the left-hand side into one statement. Instead of using the symbol \rightarrow in a production, we use the symbol $::=$. We enclose all nonterminal symbols in brackets, $\langle \rangle$, and we list all the right-hand sides of productions in the same statement, separating them by bars. For instance, the productions $A \rightarrow Aa$, $A \rightarrow a$, and $A \rightarrow AB$ can be combined into $\langle A \rangle ::= \langle A \rangle a \mid a \mid \langle A \rangle \langle B \rangle$.


Example 13 illustrates how the Backus–Naur form is used to describe the syntax of programming languages. Our example comes from the original use of Backus–Naur form in the description of ALGOL 60.

EXAMPLE 13

Extra
Examples

In ALGOL 60 an identifier (which is the name of an entity such as a variable) consists of a string of alphanumeric characters (that is, letters and digits) and must begin with a letter. We can use these rules in Backus–Naur to describe the set of allowable identifiers:


$$\begin{aligned}\langle \text{identifier} \rangle &::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle \\ \langle \text{letter} \rangle &::= a \mid b \mid \cdots \mid y \mid z \quad \text{the ellipsis indicates that all 26 letters are included} \\ \langle \text{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

For example, we can produce the valid identifier $x99a$ by using the first rule to replace $\langle \text{identifier} \rangle$ by $\langle \text{identifier} \rangle \langle \text{letter} \rangle$, the second rule to obtain $\langle \text{identifier} \rangle a$, the first rule twice to obtain $\langle \text{identifier} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle a$, the third rule twice to obtain $\langle \text{identifier} \rangle 99a$, the first rule to obtain $\langle \text{letter} \rangle 99a$, and finally the second rule to obtain $x99a$. 

EXAMPLE 14

What is the Backus–Naur form of the grammar for the subset of English described in the introduction to this section?


Solution: The Backus–Naur form of this grammar is

$$\begin{aligned}\langle \text{sentence} \rangle &::= \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle \\ \langle \text{noun phrase} \rangle &::= \langle \text{article} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle \mid \langle \text{article} \rangle \langle \text{noun} \rangle \\ \langle \text{verb phrase} \rangle &::= \langle \text{verb} \rangle \langle \text{adverb} \rangle \mid \langle \text{verb} \rangle \\ \langle \text{article} \rangle &::= a \mid the \\ \langle \text{adjective} \rangle &::= large \mid hungry \\ \langle \text{noun} \rangle &::= rabbit \mid mathematician \\ \langle \text{verb} \rangle &::= eats \mid hops \\ \langle \text{adverb} \rangle &::= quickly \mid wildly\end{aligned}$$


EXAMPLE 15

Give the Backus–Naur form for the production of signed integers in decimal notation. (A **signed integer** is a nonnegative integer preceded by a plus sign or a minus sign.)

Solution: The Backus–Naur form for a grammar that produces signed integers is

$$\begin{aligned}\langle \text{signed integer} \rangle &::= \langle \text{sign} \rangle \langle \text{integer} \rangle \\ \langle \text{sign} \rangle &::= + \mid - \\ \langle \text{integer} \rangle &::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle \\ \langle \text{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$


The Backus–Naur form, with a variety of extensions, is used extensively to specify the syntax of programming languages, such as Java and LISP; database languages, such as SQL; and markup languages, such as XML. Some extensions of the Backus–Naur form that are commonly used in the description of programming languages are introduced in the preamble to Exercise 34.

Exercises

Exercises 1–3 refer to the grammar with start symbol **sentence**, set of terminals $T = \{the, sleepy, happy, tortoise, hare, passes, runs, quickly, slowly\}$, set of nonterminals $N = \{\text{noun phrase, transitive verb phrase, intransitive verb phrase, article, adjective, noun, verb, adverb}\}$, and productions:

sentence \rightarrow **noun phrase** **transitive verb phrase**
 noun phrase
sentence \rightarrow **noun phrase** **intransitive verb phrase**
noun phrase \rightarrow **article** **adjective** **noun**
noun phrase \rightarrow **article** **noun**
transitive verb phrase \rightarrow **transitive verb**
intransitive verb phrase \rightarrow **intransitive verb** **adverb**
intransitive verb phrase \rightarrow **intransitive verb**
article \rightarrow *the*

adjective \rightarrow *sleepy*
adjective \rightarrow *happy*
noun \rightarrow *tortoise*
noun \rightarrow *hare*
transitive verb \rightarrow *passes*
intransitive verb \rightarrow *runs*
adverb \rightarrow *quickly*
adverb \rightarrow *slowly*

- Use the set of productions to show that each of these sentences is a valid sentence.
 - the happy hare runs*
 - the sleepy tortoise runs quickly*
 - the tortoise passes the hare*
 - the sleepy hare passes the happy tortoise*
- Find five other valid sentences, besides those given in Exercise 1.
- Show that *the hare runs the sleepy tortoise* is not a valid sentence.
- Let $G = (V, T, S, P)$ be the phrase-structure grammar with $V = \{0, 1, A, S\}$, $T = \{0, 1\}$, and set of productions P consisting of $S \rightarrow 1S$, $S \rightarrow 00A$, $A \rightarrow 0A$, and $A \rightarrow 0$.
 - Show that 111000 belongs to the language generated by G .
 - Show that 11001 does not belong to the language generated by G .
 - What is the language generated by G ?

- Let $G = (V, T, S, P)$ be the phrase-structure grammar with $V = \{0, 1, A, B, S\}$, $T = \{0, 1\}$, and set of productions P consisting of $S \rightarrow 0A$, $S \rightarrow 1A$, $A \rightarrow 0B$, $B \rightarrow 1A$, $B \rightarrow 1$.
 - Show that 10101 belongs to the language generated by G .
 - Show that 10110 does not belong to the language generated by G .
 - What is the language generated by G ?
- Let $V = \{S, A, B, a, b\}$ and $T = \{a, b\}$. Find the language generated by the grammar (V, T, S, P) when the set P of productions consists of
 - $S \rightarrow AB, A \rightarrow ab, B \rightarrow bb$.
 - $S \rightarrow AB, S \rightarrow aA, A \rightarrow a, B \rightarrow ba$.
 - $S \rightarrow AB, S \rightarrow AA, A \rightarrow aB, A \rightarrow ab, B \rightarrow b$.
 - $S \rightarrow AA, S \rightarrow B, A \rightarrow aaA, A \rightarrow aa, B \rightarrow bB, B \rightarrow b$.
 - $S \rightarrow AB, A \rightarrow aAb, B \rightarrow bBa, A \rightarrow \lambda, B \rightarrow \lambda$.
- Construct a derivation of 0^31^3 using the grammar given in Example 5.
- Show that the grammar given in Example 5 generates the set $\{0^n1^n \mid n = 0, 1, 2, \dots\}$.
- Construct a derivation of 0^21^4 using the grammar G_1 in Example 6.
 - Construct a derivation of 0^21^4 using the grammar G_2 in Example 6.
- Show that the grammar G_1 given in Example 6 generates the set $\{0^m1^n \mid m, n = 0, 1, 2, \dots\}$.
 - Show that the grammar G_2 in Example 6 generates the same set.
- Construct a derivation of $0^21^22^2$ in the grammar given in Example 7.
- Show that the grammar given in Example 7 generates the set $\{0^n1^n2^n \mid n = 0, 1, 2, \dots\}$.
- Find a phrase-structure grammar for each of these languages.
 - the set consisting of the bit strings 0, 1, and 11
 - the set of bit strings containing only 1s
 - the set of bit strings that start with 0 and end with 1
 - the set of bit strings that consist of a 0 followed by an even number of 1s
- Find a phrase-structure grammar for each of these languages.
 - the set consisting of the bit strings 10, 01, and 101
 - the set of bit strings that start with 00 and end with one or more 1s
 - the set of bit strings consisting of an even number of 1s followed by a final 0

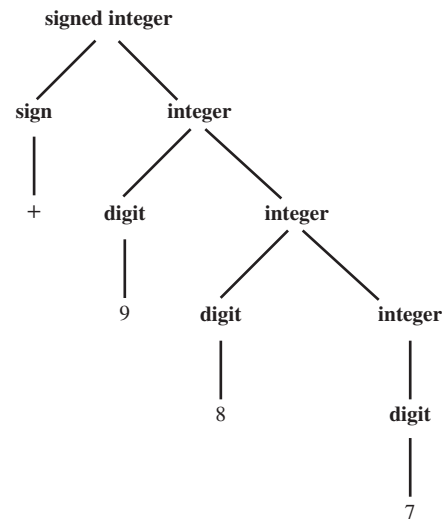
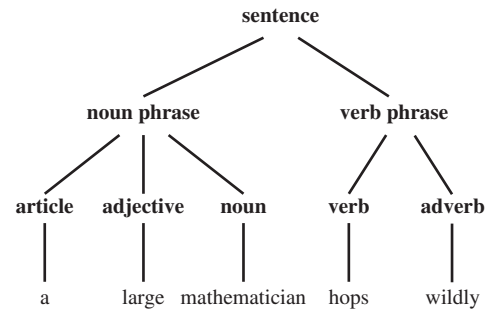
- d) the set of bit strings that have neither two consecutive 0s nor two consecutive 1s
- *15. Find a phrase-structure grammar for each of these languages.
- the set of all bit strings containing an even number of 0s and no 1s
 - the set of all bit strings made up of a 1 followed by an odd number of 0s
 - the set of all bit strings containing an even number of 0s and an even number of 1s
 - the set of all strings containing 10 or more 0s and no 1s
 - the set of all strings containing more 0s than 1s
 - the set of all strings containing an equal number of 0s and 1s
 - the set of all strings containing an unequal number of 0s and 1s
16. Construct phrase-structure grammars to generate each of these sets.
- $\{1^n \mid n \geq 0\}$
 - $\{10^n \mid n \geq 0\}$
 - $\{(11)^n \mid n \geq 0\}$
17. Construct phrase-structure grammars to generate each of these sets.
- $\{0^n \mid n \geq 0\}$
 - $\{1^n 0 \mid n \geq 0\}$
 - $\{(000)^n \mid n \geq 0\}$
18. Construct phrase-structure grammars to generate each of these sets.
- $\{01^{2n} \mid n \geq 0\}$
 - $\{0^n 1^{2n} \mid n \geq 0\}$
 - $\{0^n 1^m 0^n \mid m \geq 0 \text{ and } n \geq 0\}$
19. Let $V = \{S, A, B, a, b\}$ and $T = \{a, b\}$. Determine whether $G = (V, T, S, P)$ is a type 0 grammar but not a type 1 grammar, a type 1 grammar but not a type 2 grammar, or a type 2 grammar but not a type 3 grammar if P , the set of productions, is
- $S \rightarrow aAB, A \rightarrow Bb, B \rightarrow \lambda$.
 - $S \rightarrow aA, A \rightarrow a, A \rightarrow b$.
 - $S \rightarrow ABa, AB \rightarrow a$.
 - $S \rightarrow ABA, A \rightarrow aB, B \rightarrow ab$.
 - $S \rightarrow bA, A \rightarrow B, B \rightarrow a$.
 - $S \rightarrow aA, aA \rightarrow B, B \rightarrow aA, A \rightarrow b$.
 - $S \rightarrow bA, A \rightarrow b, S \rightarrow \lambda$.
 - $S \rightarrow AB, B \rightarrow aAb, aAb \rightarrow b$.
 - $S \rightarrow aA, A \rightarrow bB, B \rightarrow b, B \rightarrow \lambda$.
 - $S \rightarrow A, A \rightarrow B, B \rightarrow \lambda$.

20. A **palindrome** is a string that reads the same backward as it does forward, that is, a string w , where $w = w^R$, where w^R is the reversal of the string w . Find a context-free grammar that generates the set of all palindromes over the alphabet $\{0, 1\}$.

- *21. Let G_1 and G_2 be context-free grammars, generating the languages $L(G_1)$ and $L(G_2)$, respectively. Show that there is a context-free grammar generating each of these sets.

- $L(G_1) \cup L(G_2)$
- $L(G_1)L(G_2)$
- $L(G_1)^*$

22. Find the strings constructed using the derivation trees shown here.



23. Construct derivation trees for the sentences in Exercise 1.

24. Let G be the grammar with $V = \{a, b, c, S\}$; $T = \{a, b, c\}$; starting symbol S ; and productions $S \rightarrow abS$, $S \rightarrow bcS$, $S \rightarrow bbS$, $S \rightarrow a$, and $S \rightarrow cb$. Construct derivation trees for

- $bcbbba$.
- $bbbcbbba$.
- $bcabbbbbbcb$.

- *25. Use top-down parsing to determine whether each of the following strings belongs to the language generated by the grammar in Example 12.

- $baba$
- $abab$
- $cbaba$
- $bbbcba$

- *26. Use bottom-up parsing to determine whether the strings in Exercise 25 belong to the language generated by the grammar in Example 12.

27. Construct a derivation tree for -109 using the grammar given in Example 15.
28. a) Explain what the productions are in a grammar if the Backus–Naur form for productions is as follows:

$$\begin{aligned}\langle expression \rangle &::= (\langle expression \rangle) \mid \\ &\quad \langle expression \rangle + \langle expression \rangle \mid \\ &\quad \langle expression \rangle * \langle expression \rangle \mid \\ &\quad \langle variable \rangle \\ \langle variable \rangle &::= x \mid y\end{aligned}$$

- b) Find a derivation tree for $(x * y) + x$ in this grammar.
29. a) Construct a phrase-structure grammar that generates all signed decimal numbers, consisting of a sign, either $+$ or $-$; a nonnegative integer; and a decimal fraction that is either the empty string or a decimal point followed by a positive integer, where initial zeros in an integer are allowed.
- b) Give the Backus–Naur form of this grammar.
- c) Construct a derivation tree for -31.4 in this grammar.
30. a) Construct a phrase-structure grammar for the set of all fractions of the form a/b , where a is a signed integer in decimal notation and b is a positive integer.
- b) What is the Backus–Naur form for this grammar?
- c) Construct a derivation tree for $+311/17$ in this grammar.
31. Give production rules in Backus–Naur form for an identifier if it can consist of
- one or more lowercase letters.
 - at least three but no more than six lowercase letters.
 - one to six uppercase or lowercase letters beginning with an uppercase letter.
 - a lowercase letter, followed by a digit or an underscore, followed by three or four alphanumeric characters (lower or uppercase letters and digits).
32. Give production rules in Backus–Naur form for the name of a person if this name consists of a first name, which is a string of letters, where only the first letter is uppercase; a middle initial; and a last name, which can be any string of letters.
33. Give production rules in Backus–Naur form that generate all identifiers in the C programming language. In C an identifier starts with a letter or an underscore ($_$) that is followed by one or more lowercase letters, uppercase letters, underscores, and digits.

➤ Several extensions to Backus–Naur form are commonly used to define phrase-structure grammars. In one such extension, a question mark (?) indicates that the symbol, or group of symbols inside parentheses, to its left can appear zero or once (that

is, it is optional), an asterisk (*) indicates that the symbol to its left can appear zero or more times, and a plus (+) indicates that the symbol to its left can appear one or more times. These extensions are part of **extended Backus–Naur form (EBNF)**, and the symbols $?$, $*$, and $+$ are called **metacharacters**. In EBNF the brackets used to denote nonterminals are usually not shown.

34. Describe the set of strings defined by each of these sets of productions in EBNF.
- $string ::= L+D?L+$
 $L ::= a \mid b \mid c$
 $D ::= 0 \mid 1$
 - $string ::= sign D+ \mid D+$
 $sign ::= + \mid -$
 $D ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
 - $string ::= L*(D+)?L*$
 $L ::= x \mid y$
 $D ::= 0 \mid 1$
35. Give production rules in extended Backus–Naur form that generate all decimal numerals consisting of an optional sign, a nonnegative integer, and a decimal fraction that is either the empty string or a decimal point followed by an optional positive integer optionally preceded by some number of zeros.
36. Give production rules in extended Backus–Naur form that generate a sandwich if a sandwich consists of a lower slice of bread; mustard or mayonnaise; optional lettuce; an optional slice of tomato; one or more slices of either turkey, chicken, or roast beef (in any combination); optionally some number of slices of cheese; and a top slice of bread.
37. Give production rules in extended Backus–Naur form for identifiers in the C programming language (see Exercise 33).
38. Describe how productions for a grammar in extended Backus–Naur form can be translated into a set of productions for the grammar in Backus–Naur form.

This is the Backus–Naur form that describes the syntax of expressions in postfix (or reverse Polish) notation.

$$\begin{aligned}\langle expression \rangle &::= \langle term \rangle \mid \langle term \rangle \langle term \rangle \langle addOperator \rangle \\ \langle addOperator \rangle &::= + \mid - \\ \langle term \rangle &::= \langle factor \rangle \mid \langle factor \rangle \langle factor \rangle \langle mulOperator \rangle \\ \langle mulOperator \rangle &::= * \mid / \\ \langle factor \rangle &::= \langle identifier \rangle \mid \langle expression \rangle \\ \langle identifier \rangle &::= a \mid b \mid \cdots \mid z\end{aligned}$$

39. For each of these strings, determine whether it is generated by the grammar given for postfix notation. If it is, find the steps used to generate the string
- $abc*+$
 - $xy++$
 - $xy-z*$
 - $wxyz-* /$
 - $ade-*$

40. Use Backus–Naur form to describe the syntax of expressions in infix notation, where the set of operators and identifiers is the same as in the BNF for postfix expressions given in the preamble to Exercise 39, but parentheses must surround expressions being used as factors.
41. For each of these strings, determine whether it is generated by the grammar for infix expressions from
- Exercise 40. If it is, find the steps used to generate the string.
- a) $x + y + z$ b) $a/b + c/d$
 c) $m * (n + p)$ d) $+m - n + p - q$
 e) $(m + n) * (p - q)$
42. Let G be a grammar and let R be the relation containing the ordered pair (w_0, w_1) if and only if w_1 is directly derivable from w_0 in G . What is the reflexive transitive closure of R ?

13.2 Finite-State Machines with Output

13.2.1 Introduction



Many kinds of machines, including components in computers, can be modeled using a structure called a finite-state machine. Several types of finite-state machines are commonly used in models. All these versions of finite-state machines include a finite set of states, with a designated starting state, an input alphabet, and a transition function that assigns a next state to every state and input pair. Finite-state machines are used extensively in applications in computer science and data networking. For example, finite-state machines are the basis for programs for spell checking, grammar checking, indexing or searching large bodies of text, recognizing speech, transforming text using markup languages such as XML and HTML, and network protocols that specify how computers communicate.

In this section, we will study those finite-state machines that produce output. We will show how finite-state machines can be used to model a vending machine, a machine that delays input, a machine that adds integers, and a machine that determines whether a bit string contains a specified pattern.

Before giving formal definitions, we will show how a vending machine can be modeled. A vending machine accepts nickels (5 cents), dimes (10 cents), and quarters (25 cents). When a total of 30 cents or more has been deposited, the machine immediately returns the amount in excess of 30 cents. When 30 cents has been deposited and any excess refunded, the customer can push an orange button and receive an orange juice or push a red button and receive an apple juice. We can describe how the machine works by specifying its states, how it changes states when input is received, and the output that is produced for every combination of input and current state.

The machine can be in any of seven different states s_i , $i = 0, 1, 2, \dots, 6$, where s_i is the state where the machine has collected $5i$ cents. The machine starts in state s_0 , with 0 cents received. The possible inputs are 5 cents, 10 cents, 25 cents, the orange button (O), and the red button (R). The possible outputs are nothing (n), 5 cents, 10 cents, 15 cents, 20 cents, 25 cents, an orange juice, and an apple juice.

We illustrate how this model of the machine works with this example. Suppose that a student puts in a dime followed by a quarter, receives 5 cents back, and then pushes the orange button for an orange juice. The machine starts in state s_0 . The first input is 10 cents, which changes the state of the machine to s_2 and gives no output. The second input is 25 cents. This changes the state from s_2 to s_6 , and gives 5 cents as output. The next input is the orange button, which changes the state from s_6 back to s_0 (because the machine returns to the start state) and gives an orange juice as its output.

We can display all the state changes and output of this machine in a table. To do this we need to specify for each combination of state and input the next state and the output obtained. Table 1 shows the transitions and outputs for each pair of a state and an input.

Another way to show the actions of a machine is to use a directed graph with labeled edges, where each state is represented by a circle, edges represent the transitions, and edges are labeled

Finite-state machines with output are often called *finite-state transducers*.

TABLE 1 State Table for a Vending Machine.										
	Next State					Output				
State	Input					Input				
	5	10	25	O	R	5	10	25	O	R
s_0	s_1	s_2	s_5	s_0	s_0	n	n	n	n	n
s_1	s_2	s_3	s_6	s_1	s_1	n	n	n	n	n
s_2	s_3	s_4	s_6	s_2	s_2	n	n	5	n	n
s_3	s_4	s_5	s_6	s_3	s_3	n	n	10	n	n
s_4	s_5	s_6	s_6	s_4	s_4	n	n	15	n	n
s_5	s_6	s_6	s_6	s_5	s_5	n	5	20	n	n
s_6	s_6	s_6	s_6	s_0	s_0	5	10	25	OJ	AJ

with the input and the output for that transition. Figure 1 shows such a directed graph for the vending machine.

13.2.2 Finite-State Machines with Outputs

We will now give the formal definition of a finite-state machine with output.

Definition 1

A *finite-state machine* $M = (S, I, O, f, g, s_0)$ consists of a finite set S of *states*, a finite *input alphabet* I , a finite *output alphabet* O , a *transition function* f that assigns to each state and input pair a new state, an *output function* g that assigns to each state and input pair an output, and an *initial state* s_0 .

Let $M = (S, I, O, f, g, s_0)$ be a finite-state machine. We can use a **state table** to represent the values of the transition function f and the output function g for all pairs of states and input. We previously constructed a state table for the vending machine discussed in the introduction to this section.

EXAMPLE 1 The state table shown in Table 2 describes a finite-state machine with $S = \{s_0, s_1, s_2, s_3\}$, $I = \{0, 1\}$, and $O = \{0, 1\}$. The values of the transition function f are displayed in the first two columns, and the values of the output function g are displayed in the last two columns. ◀

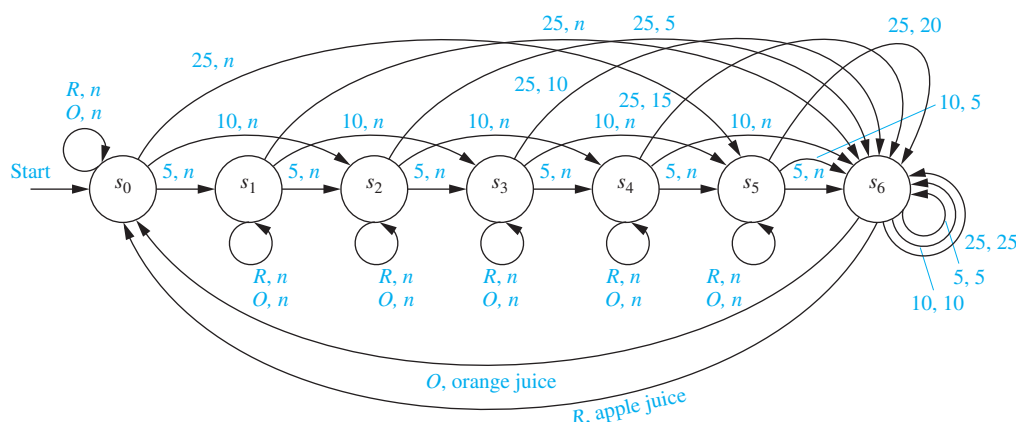


FIGURE 1 A vending machine.

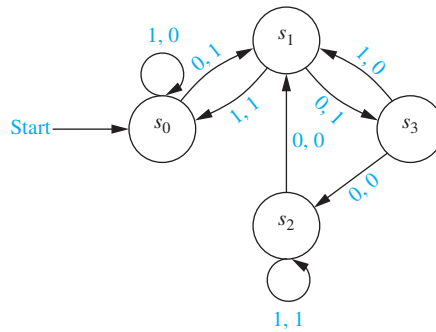


TABLE 2				
State	f		g	
	Input		Input	
	0	1	0	1
s_0	s_1	s_0	1	0
s_1	s_3	s_0	1	1
s_2	s_1	s_2	0	1
s_3	s_2	s_1	0	0

FIGURE 2 The state diagram for the finite-state machine shown in Table 2.

Another way to represent a finite-state machine is to use a **state diagram**, which is a directed graph with labeled edges. In this diagram, each state is represented by a circle. Arrows labeled with the input and output pair are shown for each transition.

EXAMPLE 2 Construct the state diagram for the finite-state machine with the state table shown in Table 2.

Solution: The state diagram for this machine is shown in Figure 2. ◀

EXAMPLE 3 Construct the state table for the finite-state machine with the state diagram shown in Figure 3.

Solution: The state table for this machine is shown in Table 3. ◀

An input string takes the starting state through a sequence of states, as determined by the transition function. As we read the input string symbol by symbol (from left to right), each input symbol takes the machine from one state to another. Because each transition produces an output, an input string also produces an output string.

Suppose that the input string is $x = x_1x_2 \dots x_k$. Then, reading this input takes the machine from state s_0 to state s_1 , where $s_1 = f(s_0, x_1)$, then to state s_2 , where $s_2 = f(s_1, x_2)$, and so on, with $s_j = f(s_{j-1}, x_j)$ for $j = 1, 2, \dots, k$, ending at state $s_k = f(s_{k-1}, x_k)$. This sequence of transitions produces an output string $y_1y_2 \dots y_k$, where $y_1 = g(s_0, x_1)$ is the output corresponding to the transition from s_0 to s_1 , $y_2 = g(s_1, x_2)$ is the output corresponding to the transition from s_1 to s_2 , and so on. In general, $y_j = g(s_{j-1}, x_j)$ for $j = 1, 2, \dots, k$. Hence, we can extend the definition of

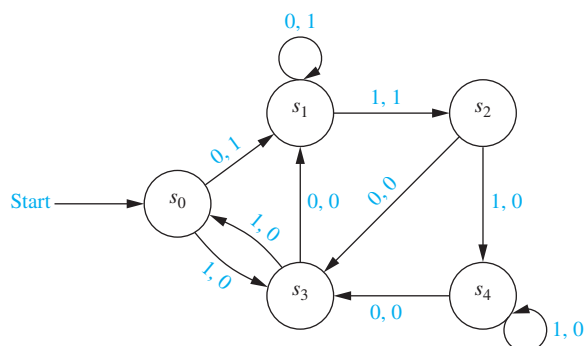


TABLE 3				
State	f		g	
	Input		Input	
	0	1	0	1
s_0	s_1	s_3	1	0
s_1	s_1	s_2	1	1
s_2	s_3	s_4	0	0
s_3	s_1	s_0	0	0
s_4	s_3	s_4	0	0

FIGURE 3 A finite-state machine.

the output function g to input strings so that $g(x) = y$, where y is the output corresponding to the input string x . This notation is useful in many applications.

EXAMPLE 4 Find the output string generated by the finite-state machine in Figure 3 if the input string is 101011.

Solution: The output obtained is 001000. The successive states and outputs are shown in Table 4. ◀

We can now look at some examples of useful finite-state machines. Examples 5, 6, and 7 illustrate that the states of a finite-state machine give it limited memory capabilities. The states can be used to remember the properties of the symbols that have been read by the machine. However, because there are only finitely many different states, finite-state machines cannot be used for some important purposes. This will be illustrated in Section 13.4.

EXAMPLE 5 An important element in many electronic devices is a *unit-delay machine*, which produces as output the input string delayed by a specified amount of time. How can a finite-state machine be constructed that delays an input string by one unit of time, that is, produces as output the bit string $0x_1x_2 \dots x_{k-1}$ given the input bit string $x_1x_2 \dots x_k$?

Solution: A delay machine can be constructed that has two possible inputs, namely, 0 and 1. The machine must have a start state s_0 . Because the machine has to remember whether the previous input was a 0 or a 1, two other states s_1 and s_2 are needed, where the machine is in state s_1 if the previous input was 1 and in state s_2 if the previous input was 0. An output of 0 is produced for the initial transition from s_0 . Each transition from s_1 gives an output of 1, and each transition from s_2 gives an output of 0. The output corresponding to the input of a string $x_1 \dots x_k$ is the string that begins with 0, followed by x_1 , followed by x_2, \dots , ending with x_{k-1} . The state diagram for this machine is shown in Figure 4. ▶

EXAMPLE 6 Produce a finite-state machine that adds two positive integers using their binary expansions.



Solution: When $(x_n \dots x_1x_0)_2$ and $(y_n \dots y_1y_0)_2$ are added, the following procedure (as described in Section 4.2) is followed. First, the bits x_0 and y_0 are added, producing a sum bit z_0 and a carry bit c_0 . This carry bit is either 0 or 1. Then, the bits x_1 and y_1 are added, together with the carry c_0 . This gives a sum bit z_1 and a carry bit c_1 . This procedure is continued until the n th stage, where x_n, y_n , and the previous carry c_{n-1} are added to produce the sum bit z_n and the carry bit c_n , which is equal to the sum bit z_{n+1} .

A finite-state machine to carry out this addition can be constructed using just two states. For simplicity we assume that both the initial bits x_n and y_n are 0 (otherwise we have to make special arrangements concerning the sum bit z_{n+1}). The start state s_0 is used to remember that the previous carry is 0 (or for the addition of the rightmost bits). The other state, s_1 , is used to remember that the previous carry is 1.

Because the inputs to the machine are pairs of bits, there are four possible inputs. We represent these possibilities by 00 (when both bits are 0), 01 (when the first bit is 0 and the second is 1), 10 (when the first bit is 1 and the second is 0), and 11 (when both bits are 1). The transitions

TABLE 4							
Input	1	0	1	0	1	1	—
State	s_0	s_3	s_1	s_2	s_3	s_0	s_3
Output	0	0	1	0	0	0	—

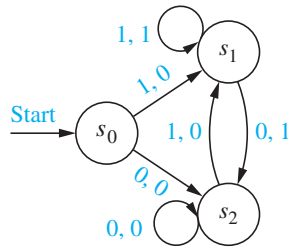


FIGURE 4 A unit-delay machine.

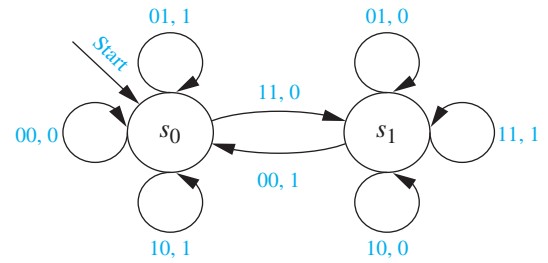


FIGURE 5 A finite-state machine for addition.

and the outputs are constructed from the sum of the two bits represented by the input and the carry represented by the state. For instance, when the machine is in state s_1 and receives 01 as input, the next state is s_1 and the output is 0, because the sum that arises is $0 + 1 + 1 = (10)_2$. The state diagram for this machine is shown in Figure 5. ◀

EXAMPLE 7 In a certain coding scheme, when three consecutive 1s appear in a message, the receiver of the message knows that there has been a transmission error. Construct a finite-state machine that gives a 1 as its current output bit if and only if the last three bits received are all 1s.

Solution: Three states are needed in this machine. The start state s_0 remembers that the previous input value, if it exists, was not a 1. The state s_1 remembers that the previous input was a 1, but the input before the previous input, if it exists, was not a 1. The state s_2 remembers that the previous two inputs were 1s.

An input of 1 takes s_0 to s_1 , because now a 1, and not two consecutive 1s, has been read; it takes s_1 to s_2 , because now two consecutive 1s have been read; and it takes s_2 to itself, because at least two consecutive 1s have been read. An input of 0 takes every state to s_0 , because this breaks up any string of consecutive 1s. The output for the transition from s_2 to itself when a 1 is read is 1, because this combination of input and state shows that three consecutive 1s have been read. All other outputs are 0. The state diagram of this machine is shown in Figure 6. ◀

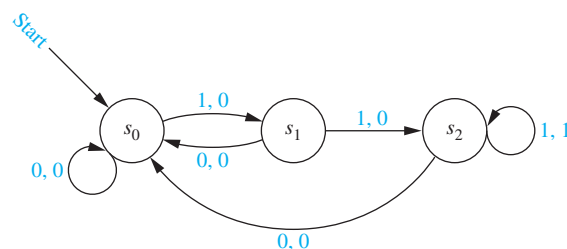


FIGURE 6 A finite-state machine that gives an output of 1 if and only if the input string read so far ends with 111.

The final output bit of the finite-state machine we constructed in Example 7 is 1 if and only if the input string ends with 111. Because of this, we say that this finite-state machine **recognizes** the set of bit strings that end with 111. This leads us to Definition 2.

Definition 2

Let $M = (S, I, O, f, g, s_0)$ be a finite-state machine and $L \subseteq I^*$. We say that M *recognizes* (or *accepts*) L if an input string x belongs to L if and only if the last output bit produced by M when given x as input is a 1.

TYPES OF FINITE-STATE MACHINES Many different kinds of finite-state machines have been developed to model computing machines. In this section we have given a definition of one type of finite-state machine. In the type of machine introduced in this section, outputs correspond to transitions between states. Machines of this type are known as **Mealy machines**, because they were first studied by G. H. Mealy in 1955. There is another important type of finite-state machine with output, where the output is determined only by the state. This type of finite-state machine is known as a **Moore machine**, because E. F. Moore introduced this type of machine in 1956. Moore machines are considered in a sequence of exercises.

In Example 7 we showed how a Mealy machine can be used for language recognition. However, another type of finite-state machine, giving no output, is usually used for this purpose. Finite-state machines with no output, also known as finite-state automata, have a set of final states and recognize a string if and only if it takes the start state to a final state. We will study this type of finite-state machine in Section 13.3.

Exercises

1. Draw the state diagrams for the finite-state machines with these state tables.

a)

State	<i>f</i>		<i>g</i>	
	<i>Input</i>		<i>Input</i>	
	0	1	0	1
s_0	s_1	s_0	0	1
s_1	s_0	s_2	0	1
s_2	s_1	s_1	0	0

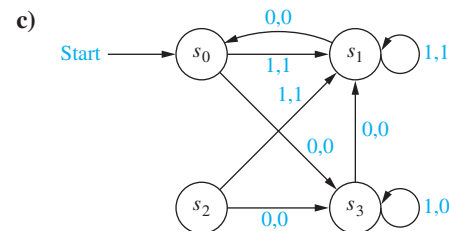
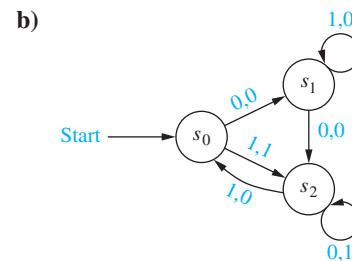
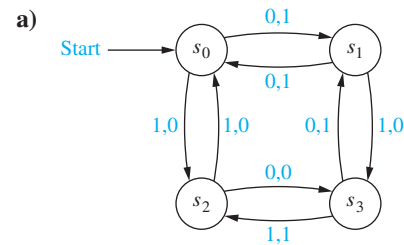
b)

State	<i>f</i>		<i>g</i>	
	<i>Input</i>		<i>Input</i>	
	0	1	0	1
s_0	s_1	s_0	0	0
s_1	s_2	s_0	1	1
s_2	s_0	s_3	0	1
s_3	s_1	s_2	1	0

c)

State	<i>f</i>		<i>g</i>	
	<i>Input</i>		<i>Input</i>	
	0	1	0	1
s_0	s_0	s_4	1	1
s_1	s_0	s_3	0	1
s_2	s_0	s_2	0	0
s_3	s_1	s_1	1	1
s_4	s_1	s_0	1	0

2. Give the state tables for the finite-state machines with these state diagrams.



3. Find the output generated from the input string 01110 for the finite-state machine with the state table in
- Exercise 1(a).
 - Exercise 1(b).
 - Exercise 1(c).
4. Find the output generated from the input string 10001 for the finite-state machine with the state diagram in
- Exercise 2(a).
 - Exercise 2(b).
 - Exercise 2(c).

5. Find the output for each of these input strings when given as input to the finite-state machine in Example 2.
a) 0111 b) 11011011 c) 01010101010
6. Find the output for each of these input strings when given as input to the finite-state machine in Example 3.
a) 0000 b) 101010 c) 11011100010
7. Construct a finite-state machine that models an old-fashioned soda machine that accepts nickels, dimes, and quarters. The soda machine accepts change until 35 cents has been put in. It gives change back for any amount greater than 35 cents. Then the customer can push buttons to receive either a cola, a root beer, or a ginger ale.
8. Construct a finite-state machine that models a newspaper vending machine that has a door that can be opened only after either three dimes (and any number of other coins) or a quarter and a nickel (and any number of other coins) have been inserted. Once the door can be opened, the customer opens it and takes a paper, closing the door. No change is ever returned no matter how much extra money has been inserted. The next customer starts with no credit.
9. Construct a finite-state machine that delays an input string two bits, giving 00 as the first two bits of output.
10. Construct a finite-state machine that changes every other bit, starting with the second bit, of an input string, and leaves the other bits unchanged.
11. Construct a finite-state machine for the log-on procedure for a computer, where the user logs on by entering a user identification number (a string of digits), which is considered to be a single input, and then a password (a string of characters), which is considered to be a single input. If the password is incorrect, the user is asked for the user identification number again.
12. Construct a finite-state machine for a combination lock that contains numbers 1 through 40 and that opens only when the correct combination, 10 right, 8 second left, 37 right, is entered. Each input is a triple consisting of a number, the direction of the turn, and the number of times the lock is turned in that direction.
13. Construct a finite-state machine for a toll machine that opens a gate after 25 cents, in nickels, dimes, or quarters, has been deposited. No change is given for overpayment, and no credit is given to the next driver when more than 25 cents has been deposited.
14. Construct a finite-state machine for entering a security code into an automatic teller machine (ATM) that implements these rules: A user enters a string of four digits, one digit at a time. If the user enters the correct four digits of the password, the ATM displays a welcome screen. When the user enters an incorrect string of four digits, the ATM displays a screen that informs the user that an incorrect password was entered. If a user enters the incorrect password three times, the account is locked.
15. Construct a finite-state machine for a restricted telephone switching system that implements these rules. Only calls to the telephone numbers 0, 911, and the digit 1 followed by 10-digit telephone numbers that begin with 212,

800, 866, 877, and 888 are sent to the network. All other strings of digits are blocked by the system and the user hears an error message.

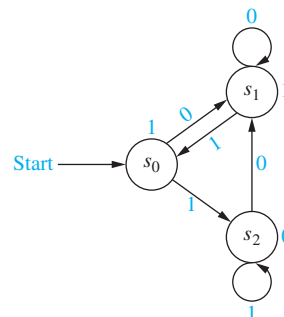
16. Construct a finite-state machine that gives an output of 1 if the number of input symbols read so far is divisible by 3 and an output of 0 otherwise.
17. Construct a finite-state machine that determines whether the input string has a 1 in the last position and a 0 in the third to the last position read so far.
18. Construct a finite-state machine that determines whether the input string read so far ends in at least five consecutive 1s.
19. Construct a finite-state machine that determines whether the word *computer* has been read as the last eight characters in the input read so far, where the input can be any string of English letters.

A **Moore machine** $M = (S, I, O, f, g, s_0)$ consists of a finite set of states, an input alphabet I , an output alphabet O , a transition function f that assigns a next state to every pair of a state and an input, an output function g that assigns an output to every state, and a starting state s_0 . A Moore machine can be represented either by a table listing the transitions for each pair of a state and input and the outputs for each state, or by a state diagram that displays the states, the transitions between states, and the output for each state. In the diagram, transitions are indicated with arrows labeled with the input, and the outputs are shown next to the states.

20. Construct the state diagram for the Moore machine with this state table.

<i>State</i>	<i>f</i>		<i>g</i>
	<i>Input</i>		
	0	1	
s_0	s_0	s_2	0
s_1	s_3	s_0	1
s_2	s_2	s_1	1
s_3	s_2	s_0	1

21. Construct the state table for the Moore machine with the state diagram shown here. Each input string to a Moore machine M produces an output string. In particular, the output corresponding to the input string $a_1 a_2 \dots a_k$ is the string $g(s_0)g(s_1) \dots g(s_k)$, where $s_i = f(s_{i-1}, a_i)$ for $i = 1, 2, \dots, k$.



22. Find the output string generated by the Moore machine in Exercise 20 with each of these input strings.
 a) 0101 b) 111111 c) 11101110111
23. Find the output string generated by the Moore machine in Exercise 21 with each of the input strings in Exercise 22.
24. Construct a Moore machine that gives an output of 1 whenever the number of symbols in the input string read so far is divisible by 4 and an output of 0 otherwise.
25. Construct a Moore machine that determines whether an input string contains an even or odd number of 1s. The machine should give 1 as output if an even number of 1s are in the string and 0 as output if an odd number of 1s are in the string.

13.3 Finite-State Machines with No Output

13.3.1 Introduction



One of the most important applications of finite-state machines is in language recognition. This application plays a fundamental role in the design and construction of compilers for programming languages. In Section 13.2 we showed that a finite-state machine with output can be used to recognize a language, by giving an output of 1 when a string from the language has been read and a 0 otherwise. However, there are other types of finite-state machines that are specially designed for recognizing languages. Instead of producing output, these machines have final states. A string is recognized if and only if it takes the starting state to one of these final states.

13.3.2 Set of Strings

Before discussing finite-state machines with no output, we will introduce some important background material on sets of strings. The operations that will be defined here will be used extensively in our discussion of language recognition by finite-state machines.

Definition 1

Suppose that A and B are subsets of V^* , where V is a vocabulary. The *concatenation* of A and B , denoted by AB , is the set of all strings of the form xy , where x is a string in A and y is a string in B .

EXAMPLE 1 Let $A = \{0, 11\}$ and $B = \{1, 10, 110\}$. Find AB and BA .

Solution: The set AB contains every concatenation of a string in A and a string in B . Hence, $AB = \{01, 010, 0110, 111, 1110, 11110\}$. The set BA contains every concatenation of a string in B and a string in A . Hence, $BA = \{10, 111, 100, 1011, 1100, 11011\}$. ◀

Note that it is not necessarily the case that $AB = BA$ when A and B are subsets of V^* , where V is an alphabet, as Example 1 illustrates.

From the definition of the concatenation of two sets of strings, we can define A^n , for $n = 0, 1, 2, \dots$. This is done recursively by specifying that

$$\begin{aligned} A^0 &= \{\lambda\}, \\ A^{n+1} &= A^n A \quad \text{for } n = 0, 1, 2, \dots \end{aligned}$$

EXAMPLE 2 Let $A = \{1, 00\}$. Find A^n for $n = 0, 1, 2$, and 3.

Solution: We have $A^0 = \{\lambda\}$ and $A^1 = A^0 A = \{\lambda\} A = \{1, 00\}$. To find A^2 we take concatenations of pairs of elements of A . This gives $A^2 = \{11, 100, 001, 0000\}$. To find A^3 we

take concatenations of elements in A^2 and A ; this gives $A^3 = \{111, 1100, 1001, 10000, 0011, 00100, 00001, 000000\}$. ◀

Definition 2 Suppose that A is a subset of V^* . Then the *Kleene closure* of A , denoted by A^* , is the set consisting of concatenations of arbitrarily many strings from A . That is, $A^* = \bigcup_{k=0}^{\infty} A^k$.

EXAMPLE 3 What are the Kleene closures of the sets $A = \{0\}$, $B = \{0, 1\}$, and $C = \{11\}$?

Solution: The Kleene closure of A is the concatenation of the string 0 with itself an arbitrary finite number of times. Hence, $A^* = \{0^n \mid n = 0, 1, 2, \dots\}$. The Kleene closure of B is the concatenation of an arbitrary number of strings, where each string is either 0 or 1. This is the set of all strings over the alphabet $V = \{0, 1\}$. That is, $B^* = V^*$. Finally, the Kleene closure of C is the concatenation of the string 11 with itself an arbitrary number of times. Hence, C^* is the set of strings consisting of an even number of 1s. That is, $C^* = \{1^{2n} \mid n = 0, 1, 2, \dots\}$. ◀

13.3.3 Finite-State Automata

We will now give a definition of a finite-state machine with no output. Such machines are also called **finite-state automata**, and that is the terminology we will use for them here. (*Note:* The singular of *automata* is *automaton*.) These machines differ from the finite-state machines studied in Section 13.2 in that they do not produce output, but they do have a set of final states. As we will see, they recognize strings that take the starting state to a final state.

Definition 3 A *finite-state automaton* $M = (S, I, f, s_0, F)$ consists of a finite set S of *states*, a finite *input alphabet* I , a *transition function* f that assigns a next state to every pair of state and input (so that $f : S \times I \rightarrow S$), an *initial* or *start state* s_0 , and a subset F of S consisting of *final* (or *accepting states*).

We can represent finite-state automata using either state tables or state diagrams. Final states are indicated in state diagrams by using double circles.

EXAMPLE 4 Construct the state diagram for the finite-state automaton $M = (S, I, f, s_0, F)$, where $S = \{s_0, s_1, s_2, s_3\}$, $I = \{0, 1\}$, $F = \{s_0, s_3\}$, and the transition function f is given in Table 1.

Solution: The state diagram is shown in Figure 1. Note that because both the inputs 0 and 1 take s_2 to s_0 , we write 0,1 over the edge from s_2 to s_0 . ◀

Links

STEPHEN COLE KLEENE (1909–1994) Stephen Kleene was born in Hartford, Connecticut. His mother, Alice Lena Cole, was a poet, and his father, Gustav Adolph Kleene, was an economics professor. Kleene attended Amherst College and received his Ph.D. from Princeton in 1934, where he studied under the famous logician Alonzo Church. Kleene joined the faculty of the University of Wisconsin in 1935, where he remained except for several leaves, including stays at the Institute for Advanced Study in Princeton. During World War II he was a navigation instructor at the Naval Reserve's Midshipmen's School and later served as the director of the Naval Research Laboratory. Kleene made significant contributions to the theory of recursive functions, investigating questions of computability and decidability, and proved one of the central results of automata theory. He served as the Acting Director of the Mathematics Research Center and as Dean of the College of Letters and Sciences at the University of Wisconsin. Kleene was a student of natural history. He discovered a previously undescribed variety of butterfly that is named after him. He was an avid hiker and climber. Kleene was also noted as a talented teller of anecdotes, using a powerful voice that could be heard several offices away.

TABLE 1		
State	f	
	Input	
	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_0
s_3	s_2	s_1

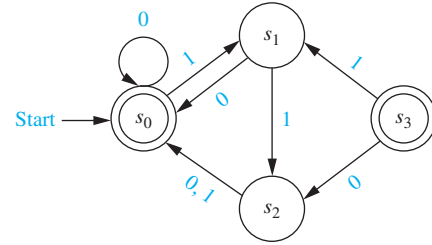


FIGURE 1 The state diagram for a finite-state automaton.

EXTENDING THE TRANSITION FUNCTION The transition function f of a finite-state machine $M = (S, I, f, s_0, F)$ can be extended so that it is defined for all pairs of states and strings; that is, f can be extended to a function $f : S \times I^* \rightarrow S$. Let $x = x_1x_2 \dots x_k$ be a string in I^* . Then $f(s_1, x)$ is the state obtained by using each successive symbol of x , from left to right, as input, starting with state s_1 . From s_1 we go on to state $s_2 = f(s_1, x_1)$, then to state $s_3 = f(s_2, x_2)$, and so on, with $f(s_1, x) = f(s_k, x_k)$. Formally, we can define this extended transition function f recursively for the deterministic finite-state machine $M = (S, I, f, s_0, F)$ by

- (i) $f(s, \lambda) = s$ for every state $s \in S$; and
- (ii) $f(s, xa) = f(f(s, x), a)$ for all $s \in S$, $x \in I^*$, and $a \in I$.

We can use structural induction and this recursive definition to prove properties of this extended transition function. For example, in Exercise 15 we ask you to prove that

$$f(s, xy) = f(f(s, x), y)$$

for every state $s \in S$ and strings $x \in I^*$ and $y \in I^*$.

13.3.4 Language Recognition by Finite-State Machines

Next, we define some terms that are used when studying the recognition by finite-state automata of certain sets of strings.

Definition 4

A string x is said to be *recognized* or *accepted* by the machine $M = (S, I, f, s_0, F)$ if it takes the initial state s_0 to a final state, that is, $f(s_0, x)$ is a state in F . The *language recognized* or *accepted* by the machine M , denoted by $L(M)$, is the set of all strings that are recognized by M . Two finite-state automata are called *equivalent* if they recognize the same language.

In Example 5 we will find the languages recognized by several finite-state automata.

EXAMPLE 5 Determine the languages recognized by the finite-state automata M_1 , M_2 , and M_3 in Figure 2.

Solution: The only final state of M_1 is s_0 . The strings that take s_0 to itself are those consisting of zero or more consecutive 1s. Hence, $L(M_1) = \{1^n \mid n = 0, 1, 2, \dots\}$.

The only final state of M_2 is s_2 . The only strings that take s_0 to s_2 are 1 and 01. Hence, $L(M_2) = \{1, 01\}$.

The final states of M_3 are s_0 and s_3 . The only strings that take s_0 to itself are λ , 0, 00, 000, \dots , that is, any string of zero or more consecutive 0s. The only strings that take s_0 to s_3 are a string of zero or more consecutive 0s, followed by 10, followed by any string. Hence, $L(M_3) = \{0^n, 0^n10x \mid n = 0, 1, 2, \dots, \text{ and } x \text{ is any string}\}$. ◀

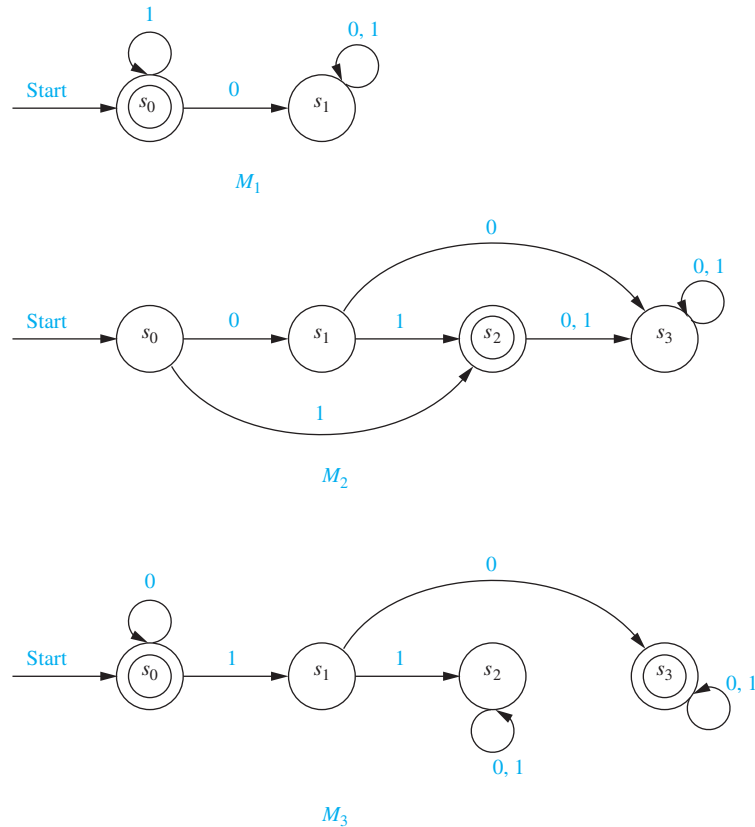


FIGURE 2 Some finite-state automata.

DESIGNING FINITE-STATE AUTOMATA We can often construct a finite-state automaton that recognizes a given set of strings by carefully adding states and transitions and determining which of these states should be final states. When appropriate we include states that can keep track of some of the properties of the input string, providing the finite-state automaton with limited memory. Examples 6 and 7 illustrate some of the techniques that can be used to construct finite-state automata that recognize particular types of sets of strings.

EXAMPLE 6 Construct deterministic finite-state automata that recognize each of these languages.

Extra Examples ➤

- the set of bit strings that begin with two 0s
- the set of bit strings that contain two consecutive 0s
- the set of bit strings that do not contain two consecutive 0s
- the set of bit strings that end with two 0s
- the set of bit strings that contain at least two 0s

Solution: (a) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that begin with two 0s. Besides the start state s_0 , we include a nonfinal state s_1 ; we move to s_1 from s_0 if the first bit is a 0. Next, we add a final state s_2 , which we move to from s_1 if the second bit is a 0. When we have reached s_2 we know that the first two input bits are both 0s, so we stay in the state s_2 no matter what the succeeding bits (if any) are. We move to a nonfinal state s_3 from s_0 if the first bit is a 1 and from s_1 if the second bit is a 1. The reader should verify that the finite-state automaton in Figure 3(a) recognizes the set of bit strings that begin with two 0s.

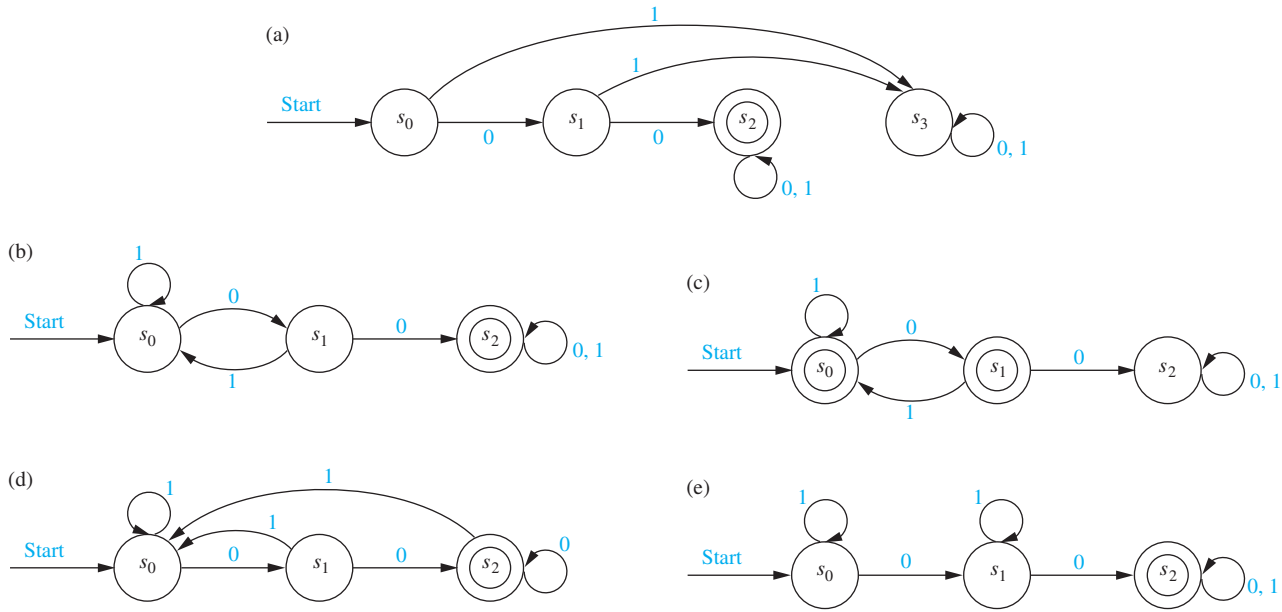


FIGURE 3 Deterministic finite-state automata recognizing the languages in Example 6.

(b) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that contain two consecutive 0s. Besides the start state s_0 , we include a nonfinal state s_1 , which tells us that the last input bit seen is a 0, but either the bit before it was a 1, or this bit was the initial bit of the string. We include a final state s_2 that we move to from s_1 when the next input bit after a 0 is also a 0. If a 1 follows a 0 in the string (before we encounter two consecutive 0s), we return to s_0 and begin looking for consecutive 0s all over again. The reader should verify that the finite-state automaton in Figure 3(b) recognizes the set of bit strings that contain two consecutive 0s.

(c) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that do not contain two consecutive 0s. Besides the start state s_0 , which should be a final state, we include a final state s_1 , which we move to from s_0 when 0 is the first input bit. When an input bit is a 1, we return to, or stay in, state s_0 . We add a state s_2 , which we move to from s_1 when the input bit is a 0. Reaching s_2 tells us that we have seen two consecutive 0s as input bits. We stay in state s_2 once we have reached it; this state is not final. The reader should verify that the finite-state automaton in Figure 3(c) recognizes the set of bit strings that do not contain two consecutive 0s. [The astute reader will notice the relationship between the finite-state automaton constructed here and the one constructed in part (b). See Exercise 39.]

(d) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that end with two 0s. Besides the start state s_0 , we include a nonfinal state s_1 , which we move to if the first bit is 0. We include a final state s_2 , which we move to from s_1 if the next input bit after a 0 is also a 0. If an input of 0 follows a previous 0, we stay in state s_2 because the last two input bits are still 0s. Once we are in state s_2 , an input bit of 1 sends us back to s_0 , and we begin looking for consecutive 0s all over again. We also return to s_0 if the next input is a 1 when we are in state s_1 . The reader should verify that the finite-state automaton in Figure 3(d) recognizes the set of bit strings that end with two 0s.

(e) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that contain two 0s. Besides the start state, we include a state s_1 , which is not final; we stay in s_0 until an input bit is a 0 and we move to s_1 when we encounter the first 0 bit in the input. We add a final state s_2 , which we move to from s_1 once we encounter a second 0 bit. Whenever we encounter a 1 as input, we stay in the current state. Once we have reached s_2 , we remain

there. Here, s_1 and s_2 are used to tell us that we have already seen one or two 0s in the input string so far, respectively. The reader should verify that the finite-state automaton in Figure 3(e) recognizes the set of bit strings that contain two 0s. ◀

EXAMPLE 7 Construct a deterministic finite-state automaton that recognizes the set of bit strings that contain an odd number of 1s and that end with at least two consecutive 0s.

Solution: We can build a deterministic finite-state automaton that recognizes the specified set by including states that keep track of both the parity of the number of 1 bits and whether we have seen no, one, or at least two 0s at the end of the input string.

The start state s_0 can be used to tell us that the input read so far contains an even number of 1s and ends with no 0s (that is, is empty or ends with a 1). Besides the start state, we include five more states. We move to states s_1 , s_2 , s_3 , s_4 , and s_5 , respectively, when the input string read so far contains an even number of 1s and ends with one 0; when it contains an even number of 1s and ends with at least two 0s; when it contains an odd number of 1s and ends with no 0s; when it contains an odd number of 1s and ends with one 0; and when it contains an odd number of 1s and ends with two 0s. The state s_5 is a final state.

The reader should verify that the finite-state automaton in Figure 4 recognizes the set of bit strings that contain an odd number of 1s and end with at least two consecutive 0s. ◀

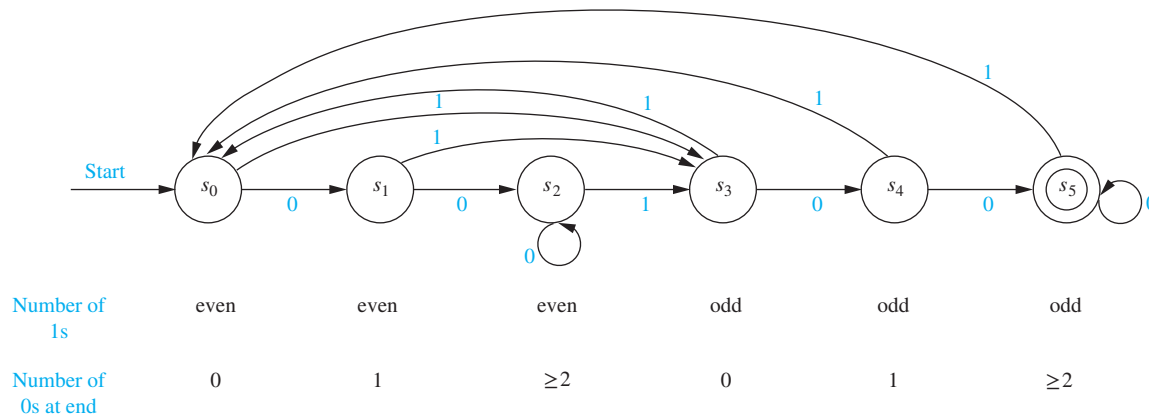


FIGURE 4 A deterministic finite-state automaton recognizing the set of bit strings containing an odd number of 1s and ending with at least two 0s.

EQUIVALENT FINITE-STATE AUTOMATA In Definition 4 we specified that two finite-state automata are equivalent if they recognize the same language. Example 8 provides an example of two equivalent deterministic finite-state machines.

EXAMPLE 8 Show that the two finite-state automata M_0 and M_1 shown in Figure 5 are equivalent.

Solution: For a string x to be recognized by M_0 , x must take us from s_0 to the final state s_1 or the final state s_4 . The only string that takes us from s_0 to s_1 is the string 1. The strings that take us from s_0 to s_4 are those strings that begin with a 0, which takes us from s_0 to s_2 , followed by zero or more additional 0s, which keep the machine in state s_2 , followed by a 1, which takes us from state s_2 to the final state s_4 . All other strings take us from s_0 to a state that is not final. (We leave it to the reader to fill in the details.) We conclude that $L(M_0)$ is the set of strings of zero or more 0 bits followed by a final 1.

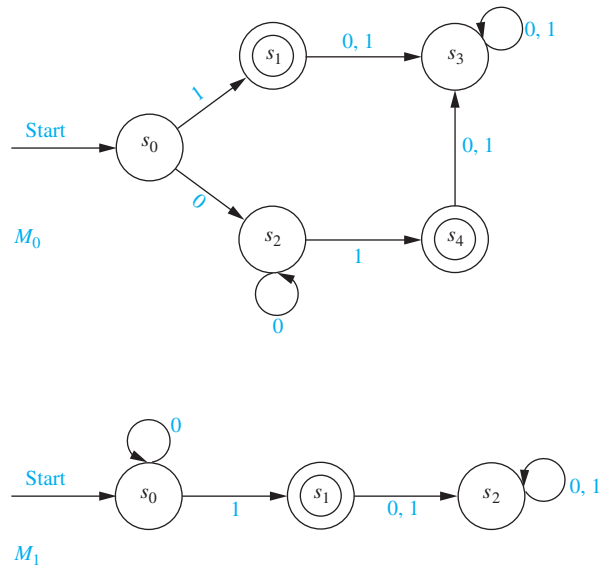


FIGURE 5 M_0 and M_1 are equivalent finite-state automata.

For a string x to be recognized by M_1 , x must take us from s_0 to the final state s_1 . So, for x to be recognized, it must begin with some number of 0s, which leave us in state s_0 , followed by a 1, which takes us to the final state s_1 . A string of all zeros is not recognized because it leaves us in state s_0 , which is not final. All strings that contain a 0 after 1 are not recognized because they take us to state s_2 , which is not final. It follows that $L(M_1)$ is the same as $L(M_0)$. We conclude that M_0 and M_1 are equivalent.

Note that the finite-state machine M_1 only has three states. No finite state machine with fewer than three states can be used to recognize the set of all strings of zero or more 0 bits followed by a 1 (see Exercise 37). ◀

As Example 8 shows, a finite-state automaton may have more states than one equivalent to it. In fact, algorithms used to construct finite-state automata to recognize certain languages may have many more states than necessary. Using unnecessarily large finite-state machines to recognize languages can make both hardware and software applications inefficient and costly. This problem arises when finite-state automata are used in compilers, which translate computer programs to a language a computer can understand (object code).

Exercises 58–61 develop a procedure that constructs a finite-state automaton with the fewest states possible among all finite-state automata equivalent to a given finite-state automaton. This procedure is known as **machine minimization**. The minimization procedure described in these exercises reduces the number of states by replacing states with equivalence classes of states with respect to an equivalence relation in which two states are equivalent if every input string either sends both states to a final state or sends both to a state that is not final. Before the minimization procedure begins, all states that cannot be reached from the start state using any input string are first removed; removing these does not change the language recognized.

13.3.5 Nondeterministic Finite-State Automata

The finite-state automata discussed so far are **deterministic**, because for each pair of state and input value there is a unique next state given by the transition function. There is another important type of finite-state automaton in which there may be several possible next states for

each pair of input value and state. Such machines are called **nondeterministic**. Nondeterministic finite-state automata are important in determining which languages can be recognized by a finite-state automaton.

Definition 5

Links 

A *nondeterministic finite-state automaton* $M = (S, I, f, s_0, F)$ consists of a set S of states, an input alphabet I , a transition function f that assigns a set of states to each pair of state and input (so that $f : S \times I \rightarrow P(S)$), a starting state s_0 , and a subset F of S consisting of the final states.

We can represent nondeterministic finite-state automata using state tables or state diagrams. When we use a state table, for each pair of state and input value we give a list of possible next states. In the state diagram, we include an edge from each state to all possible next states, labeling edges with the input or inputs that lead to this transition.

EXAMPLE 9

Find the state diagram for the nondeterministic finite-state automaton with the state table shown in Table 2. The final states are s_2 and s_3 .

Solution: The state diagram for this automaton is shown in Figure 6. 

Links 



©Bettmann/Getty Images

GRACE BREWSTER MURRAY HOPPER (1906–1992) Grace Hopper, born in New York City, displayed an intense curiosity as a child with how things worked. At the age of seven, she disassembled alarm clocks to discover their mechanisms. She inherited her love of mathematics from her mother, who received special permission to study geometry (but not algebra and trigonometry) at a time when women were actively discouraged from such study. Hopper was inspired by her father, a successful insurance broker, who had lost his legs from circulatory problems. He told his children they could do anything if they put their minds to it. He inspired Hopper to pursue higher education and not conform to the usual roles for females. Her parents made sure that she had an excellent education; she attended private schools for girls in New York. Hopper entered Vassar College in 1924, where she majored in mathematics and physics; she graduated in 1928. She received a masters degree in mathematics from Yale University in 1930. In 1930 she also married an English instructor at the New York School of Commerce; she later divorced and did not have children. Hopper was a mathematics professor at Vassar from 1931 until 1943, earning a Ph.D. from Yale in 1934.

After the attack on Pearl Harbor, Hopper, coming from a family with strong military traditions, decided to leave her academic position and join the Navy WAVES. To enlist, she needed special permission to leave her strategic position as a mathematics professor, as well as a waiver for weighing too little. In December 1943, she was sworn into the Navy Reserve and trained at the Midshipman's School for Women. Hopper was assigned to work at the Naval Ordnance Laboratory at Harvard University. She wrote programs for the world's first large-scale automatically sequenced digital computer, which was used to help aim Navy artillery in varying weather. Hopper has been credited with coining the term "bug" to refer to a hardware glitch, but it was used at Harvard prior to her arrival there. However, it is true that Hopper and her programming team found a moth in one of the relays in the computer hardware that shut the system down. This famous moth was pasted into a lab book. In the 1950s Hopper coined the term "debug" for the process of removing programming errors.

In 1946, when the Navy told her that she was too old for active service, Hopper chose to remain at Harvard as a civilian research fellow. In 1949 she left Harvard to join the Eckert–Mauchly Computer Corporation, where she helped develop UNIVAC, the first commercial computer. Hopper remained with this company when it was taken over by Remington Rand and when Remington Rand merged with the Sperry Corporation. She was a visionary for the potential power of computers; she understood that computers would become widely used if tools that were both programmer-friendly and application-friendly could be developed. In particular, she believed that computer programs could be written in English, rather than using machine instructions. To help achieve this goal, she developed the first compiler. She published the first research paper on compilers in 1952. Hopper is also known as the mother of the computer language COBOL; members of Hopper's staff helped to frame the basic language design for COBOL using their earlier work as a basis.

In 1966, Hopper retired from the Navy Reserve. However, only seven months later, the Navy recalled her from retirement to help standardize high-level naval computer languages. In 1983 she was promoted to the rank of Commodore by special Presidential appointment, and in 1985 she was elevated to the rank of Rear Admiral. Her retirement from the Navy, at the age of 80, was held on the *USS Constitution*.

TABLE 2		
State	f	
	Input	
	0	1
s_0	s_0, s_1	s_3
s_1	s_0	s_1, s_3
s_2		s_0, s_2
s_3	s_0, s_1, s_2	s_1

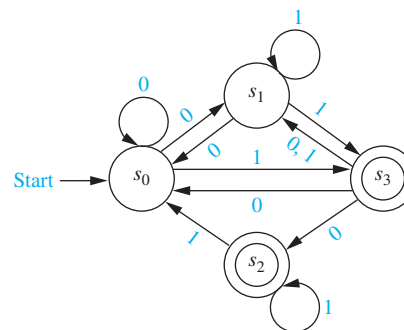


FIGURE 6 The nondeterministic finite-state automaton with state table given in Table 2.

EXAMPLE 10 Find the state table for the nondeterministic finite-state automaton with the state diagram shown in Figure 7.

Solution: The state table is given as Table 3. ◀

What does it mean for a nondeterministic finite-state automaton to recognize a string $x = x_1x_2 \dots x_k$? The first input symbol x_1 takes the starting state s_0 to a set S_1 of states. The next input symbol x_2 takes each of the states in S_1 to a set of states. Let S_2 be the union of these sets. We continue this process, including, at each stage, all states obtained using a state obtained at the previous stage and the current input symbol. We **recognize**, or **accept**, the string x if there is a final state in the set of all states that can be obtained from s_0 using x . The **language recognized** by a nondeterministic finite-state automaton is the set of all strings recognized by this automaton.

EXAMPLE 11 Find the language recognized by the nondeterministic finite-state automaton shown in Figure 7.

Solution: Because s_0 is a final state, and there is a transition from s_0 to itself when 0 is the input, the machine recognizes all strings consisting of zero or more consecutive 0s. Furthermore, because s_4 is a final state, any string that has s_4 in the set of states that can be reached from s_0 with this input string is recognized. The only such strings are strings consisting of zero or more consecutive 0s followed by 01 or 11. Because s_0 and s_4 are the only final states, the language recognized by the machine is $\{0^n, 0^n01, 0^n11 \mid n \geq 0\}$. ◀

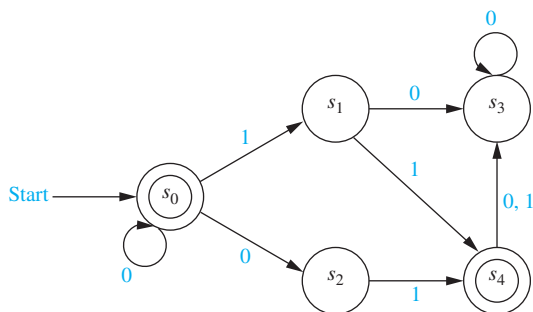


FIGURE 7 A nondeterministic finite-state automaton.

TABLE 3		
State	f	
	Input	
	0	1
s_0	s_0, s_2	s_1
s_1	s_3	s_4
s_2		s_4
s_3	s_3	
s_4	s_3	s_3

One important fact is that a language recognized by a nondeterministic finite-state automaton is also recognized by a deterministic finite-state automaton. We will take advantage of this fact in Section 13.4 when we will determine which languages are recognized by finite-state automata.

THEOREM 1

If the language L is recognized by a nondeterministic finite-state automaton M_0 , then L is also recognized by a deterministic finite-state automaton M_1 .



Proof: We will describe how to construct the deterministic finite-state automaton M_1 that recognizes L from M_0 , the nondeterministic finite-state automaton that recognizes this language. Each state in M_1 will be made up of a set of states in M_0 . The start symbol of M_1 is $\{s_0\}$, which is the set containing the start state of M_0 . The input set of M_1 is the same as the input set of M_0 .

Given a state $\{s_{i_1}, s_{i_2}, \dots, s_{i_k}\}$ of M_1 , the input symbol x takes this state to the union of the sets of next states for the elements of this set, that is, the union of the sets $f(s_{i_1}, x)$, $f(s_{i_2}, x)$, \dots , $f(s_{i_k}, x)$. The states of M_1 are all the subsets of S , the set of states of M_0 , that are obtained in this way starting with s_0 . (There are as many as 2^n states in the deterministic machine, where n is the number of states in the nondeterministic machine, because all subsets may occur as states, including the empty set, although usually far fewer states occur.) The final states of M_1 are those sets that contain a final state of M_0 .

Suppose that an input string is recognized by M_0 . Then one of the states that can be reached from s_0 using this input string is a final state (the reader should provide an inductive proof of this). This means that in M_1 , this input string leads from $\{s_0\}$ to a set of states of M_0 that contains a final state. This subset is a final state of M_1 , so this string is also recognized by M_1 . Also, an input string not recognized by M_0 does not lead to any final states in M_0 . (The reader should provide the details that prove this statement.) Consequently, this input string does not lead from $\{s_0\}$ to a final state in M_1 . ◀

EXAMPLE 12 Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Example 10.

Solution: The deterministic automaton shown in Figure 8 is constructed from the nondeterministic automaton in Example 10. The states of this deterministic automaton are subsets of the set of all states of the nondeterministic machine. The next state of a subset under an input symbol is the subset containing the next states in the nondeterministic machine of all elements in this subset. For instance, on input of 0, $\{s_0\}$ goes to $\{s_0, s_2\}$, because s_0 has transitions to itself and to s_2 in the nondeterministic machine; the set $\{s_0, s_2\}$ goes to $\{s_1, s_4\}$ on input of 1, because

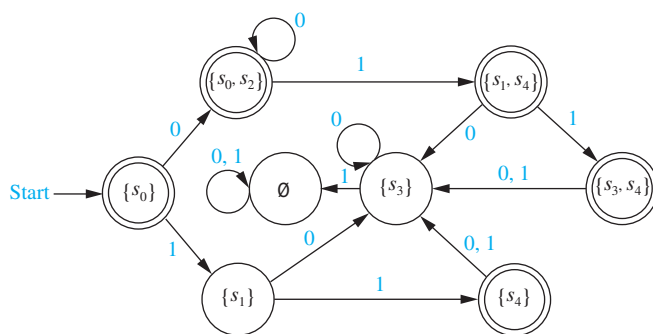


FIGURE 8 A deterministic automaton equivalent to the nondeterministic automaton in Example 10.

s_0 goes just to s_1 and s_2 goes just to s_4 on input of 1 in the nondeterministic machine; and the set $\{s_1, s_4\}$ goes to $\{s_3\}$ on input of 0, because s_1 and s_4 both go to just s_3 on input of 0 in the deterministic machine. All subsets that are obtained in this way are included in the deterministic finite-state machine. Note that the empty set is one of the states of this machine, because it is the subset containing all the next states of $\{s_3\}$ on input of 1. The start state is $\{s_0\}$, and the set of final states are all those that include s_0 or s_4 . ◀

Exercises

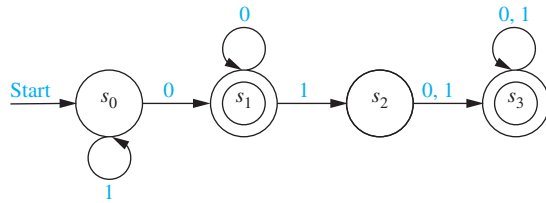
- Let $A = \{0, 11\}$ and $B = \{00, 01\}$. Find each of these sets.
a) AB b) BA c) A^2 d) B^3
- Show that if A is a set of strings, then $A\emptyset = \emptyset A = \emptyset$.
- Find all pairs of sets of strings A and B for which $AB = \{10, 111, 1010, 1000, 10111, 101000\}$.
- Show that these equalities hold.
a) $\{\lambda\}^* = \{\lambda\}$
b) $(A^*)^* = A^*$ for every set of strings A
- Describe the elements of the set A^* for these values of A .
a) $\{10\}$ b) $\{111\}$ c) $\{0, 01\}$ d) $\{1, 101\}$
- Let V be an alphabet, and let A and B be subsets of V^* . Show that $|AB| \leq |A||B|$.
- Let V be an alphabet, and let A and B be subsets of V^* with $A \subseteq B$. Show that $A^* \subseteq B^*$.
- Suppose that A is a subset of V^* , where V is an alphabet. Prove or disprove each of these statements.
a) $A \subseteq A^2$ b) if $A = A^2$, then $\lambda \in A$
c) $A\{\lambda\} = A$ d) $(A^*)^* = A^*$
e) $A^*A = A^*$ f) $|A^n| = |A|^n$
- Determine whether the string 11101 is in each of these sets.
a) $\{0, 1\}^*$ b) $\{1\}^*\{0\}^*\{1\}^*$
c) $\{11\}\{0\}^*\{01\}$ d) $\{11\}^*\{01\}^*$
e) $\{111\}^*\{0\}^*\{1\}$ f) $\{11, 0\}\{00, 101\}$
- Determine whether the string 01001 is in each of these sets.
a) $\{0, 1\}^*$ b) $\{0\}^*\{10\}\{1\}^*$
c) $\{010\}^*\{0\}^*\{1\}$ d) $\{010, 011\}\{00, 01\}$
e) $\{00\}\{0\}^*\{01\}$ f) $\{01\}^*\{01\}^*$
- Determine whether each of these strings is recognized by the deterministic finite-state automaton in Figure 1.
a) 111 b) 0011 c) 1010111 d) 011011011
- Determine whether each of these strings is recognized by the deterministic finite-state automaton in Figure 1.
a) 010 b) 1101 c) 1111110 d) 010101010
- Determine whether all the strings in each of these sets are recognized by the deterministic finite-state automaton in Figure 1.
a) $\{0\}^*$ b) $\{0\}\{0\}^*$ c) $\{1\}\{0\}^*$
d) $\{01\}^*$ e) $\{0\}^*\{1\}^*$ f) $\{1\}\{0, 1\}^*$

- Show that if $M = (S, I, f, s_0, F)$ is a deterministic finite-state automaton and $f(s, x) = s$ for the state $s \in S$ and the input string $x \in I^*$, then $f(s, x^n) = s$ for every nonnegative integer n . (Here x^n is the concatenation of n copies of the string x , defined recursively in Exercise 37 in Section 5.3.)
- Given a deterministic finite-state automaton $M = (S, I, f, s_0, F)$, use structural induction and the recursive definition of the extended transition function f to prove that $f(s, xy) = f(f(s, x), y)$ for all states $s \in S$ and all strings $x \in I^*$ and $y \in I^*$.

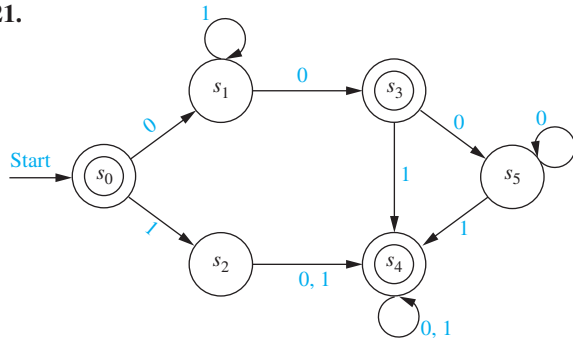
In Exercises 16–22 find the language recognized by the given deterministic finite-state automaton.

-
-
-
-

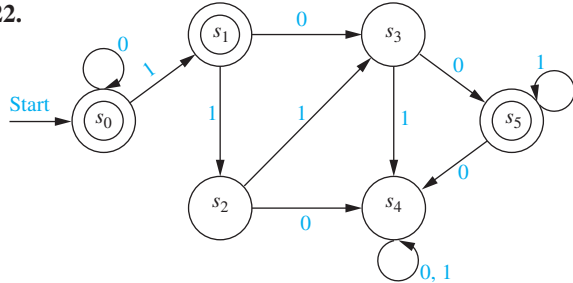
20.



21.



22.



23. Construct a deterministic finite-state automaton that recognizes the set of all bit strings beginning with 01.
24. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that end with 10.
25. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain the string 101.
26. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that do not contain three consecutive 0s.
27. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain exactly three 0s.
28. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain at least three 0s.
29. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain three consecutive 1s.
30. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that begin with 0 or with 11.
31. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that begin and end with 11.

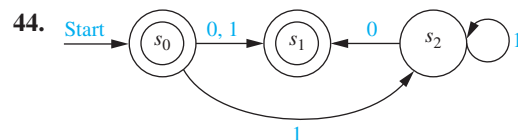
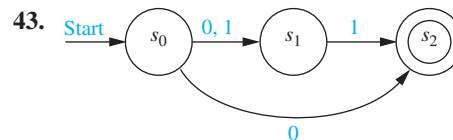
32. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain an even number of 1s.
33. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain an odd number of 0s.
34. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain an even number of 0s and an odd number of 1s.
35. Construct a finite-state automaton that recognizes the set of bit strings consisting of a 0 followed by a string with an odd number of 1s.
36. Construct a finite-state automaton with four states that recognizes the set of bit strings containing an even number of 1s and an odd number of 0s.
37. Show that there is no finite-state automaton with two states that recognizes the set of all bit strings that have one or more 1 bits and end with a 0.
38. Show that there is no finite-state automaton with three states that recognizes the set of bit strings containing an even number of 1s and an even number of 0s.
39. Explain how you can change the deterministic finite-state automaton M so that the changed automaton recognizes the set $I^* - L(M)$.

40. Use Exercise 39 and finite-state automata constructed in Example 6 to find deterministic finite-state automata that recognize each of these sets.

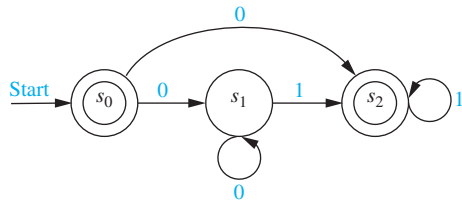
- a) the set of bit strings that do not begin with two 0s
- b) the set of bit strings that do not end with two 0s
- c) the set of bit strings that contain at most one 0 (that is, that do not contain at least two 0s)

41. Use the procedure you described in Exercise 39 and the finite-state automata you constructed in Exercise 25 to find a deterministic finite-state automaton that recognizes the set of all bit strings that do not contain the string 101.
42. Use the procedure you described in Exercise 39 and the finite-state automaton you constructed in Exercise 29 to find a deterministic finite-state automaton that recognizes the set of all bit strings that do not contain three consecutive 1s.

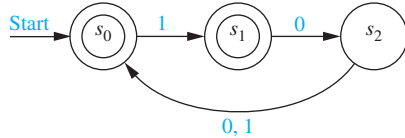
In Exercises 43–49 find the language recognized by the given nondeterministic finite-state automaton.



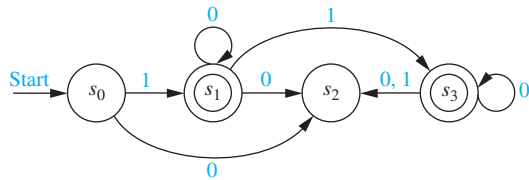
45.



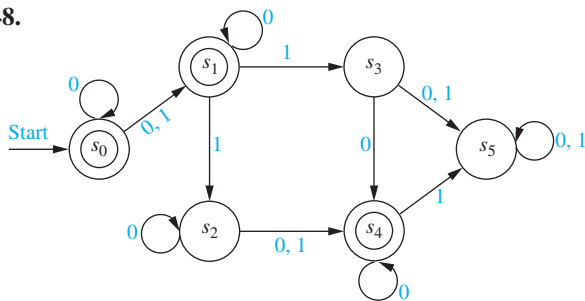
46.



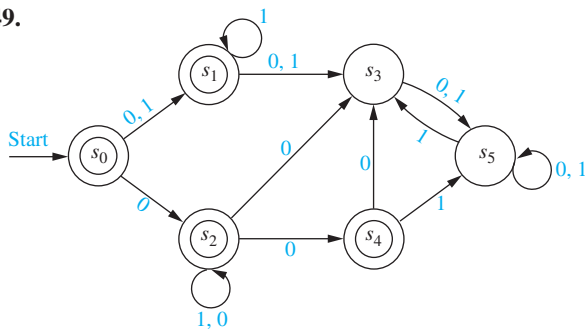
47.



48.



49.



50. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 43.

51. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 44.

52. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 45.

53. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 46.

54. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 47.

55. Find a deterministic finite-state automaton that recognizes each of these sets.

- a) $\{0\}$ b) $\{1, 00\}$
 c) $\{1^n \mid n = 2, 3, 4, \dots\}$

56. Find a nondeterministic finite-state automaton that recognizes each of the languages in Exercise 55, and has fewer states, if possible, than the deterministic automaton you found in that exercise.

*57. Show that there is no finite-state automaton that recognizes the set of bit strings containing an equal number of 0s and 1s.

In Exercises 58–62 we introduce a technique for constructing a deterministic finite-state machine equivalent to a given deterministic finite-state machine with the least number of states possible. Suppose that $M = (S, I, f, s_0, F)$ is a finite-state automaton and that k is a nonnegative integer. Let R_k be the relation on the set S of states of M such that $sR_k t$ if and only if for every input string x with $l(x) \leq k$ [where $l(x)$ is the length of x , as usual], $f(s, x)$ and $f(t, x)$ are both final states or both not final states. Furthermore, let R_* be the relation on the set of states of M such that $sR_* t$ if and only if for every input string x , regardless of length, $f(s, x)$ and $f(t, x)$ are both final states or both not final states.

- *58. a) Show that for every nonnegative integer k , R_k is an equivalence relation on S . We say that two states s and t are **k -equivalent** if $sR_k t$.
 b) Show that R_* is an equivalence relation on S . We say that two states s and t are ***-equivalent** if $sR_* t$.
 c) Show that if s and t are two k -equivalent states of M , where k is a positive integer, then s and t are also $(k-1)$ -equivalent.
 d) Show that the equivalence classes of R_k are a refinement of the equivalence classes of R_{k-1} if k is a positive integer. (The refinement of a partition of a set is defined in the preamble to Exercise 49 in Section 9.5.)
 e) Show that if s and t are k -equivalent for every nonnegative integer k , then they are *-equivalent.
 f) Show that all states in a given R_* -equivalence class are final states or all are not final states.
 g) Show that if s and t are R_* -equivalent, then $f(s, a)$ and $f(t, a)$ are also R_* -equivalent for all $a \in I$.

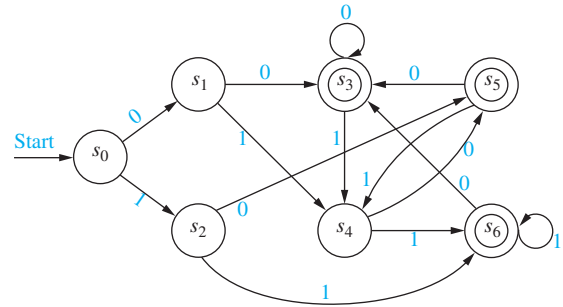
*59. Show that there is a nonnegative integer n such that the set of n -equivalence classes of states of M is the same as the set of $(n+1)$ -equivalence classes of states of M . Then show for this integer n , the set of n -equivalence classes of states of M equals the set of *-equivalence classes of states of M .

The **quotient automaton** \bar{M} of the deterministic finite-state automaton $M = (S, I, f, s_0, F)$ is the finite-state automaton $(\bar{S}, I, \bar{f}, [s_0]_{R_*}, \bar{F})$, where the set of states \bar{S} is the set of *-equivalence classes of S , the transition function \bar{f} is defined by $\bar{f}([s]_{R_*}, a) = [f(s, a)]_{R_*}$ for all states $[s]_{R_*}$ of \bar{M} and input symbols $a \in I$, and \bar{F} is the set consisting of R_* -equivalence classes of final states of M .

- *60. a) Show that s and t are 0-equivalent if and only if either both s and t are final states or neither s nor t is a final state. Conclude that each final state of \bar{M} , which is an R_* -equivalence class, contains only final states of M .
- b) Show that if k is a positive integer, then s and t are k -equivalent if and only if s and t are $(k-1)$ -equivalent and for every input symbol $a \in I$, $f(s, a)$ and $f(t, a)$ are $(k-1)$ -equivalent. Conclude that the transition function \bar{f} is well-defined.
- c) Describe a procedure that can be used to construct the quotient automaton of a finite-automaton M .
- **61. a) Show that if M is a finite-state automaton, then the quotient automaton \bar{M} recognizes the same language as M .
- b) Show that if M is a finite-state automaton with the property that for every state s of M there is a string $x \in I^*$ such that $f(s_0, x) = s$, then the quotient

automaton \bar{M} has the minimum number of states of any finite-state automaton equivalent to M .

62. Answer these questions about the finite-state automaton M shown here.



- a) Find the k -equivalence classes of M for $k = 0, 1, 2$, and 3. Also, find the $*$ -equivalence classes of M .
- b) Construct the quotient automaton \bar{M} of M .

13.4 Language Recognition

13.4.1 Introduction

We have seen that finite-state automata can be used as language recognizers. What sets can be recognized by these machines? Although this seems like an extremely difficult problem, there is a simple characterization of the sets that can be recognized by finite state automata. This problem was first solved in 1956 by the American mathematician Stephen Kleene. He showed that there is a finite-state automaton that recognizes a set if and only if this set can be built up from the null set, the empty string, and singleton strings by taking concatenations, unions, and Kleene closures, in arbitrary order. Sets that can be built up in this way are called **regular sets**. Regular grammars were defined in Section 13.1. Because of the terminology used, it is not surprising that there is a connection between regular sets, which are the sets recognized by finite-state automata, and regular grammars. In particular, a set is regular if and only if it is generated by a regular grammar.

Finally, there are sets that cannot be recognized by any finite-state automata. We will give an example of such a set. We will briefly discuss more powerful models of computation, such as pushdown automata and Turing machines, at the end of this section. The regular sets are those that can be formed using the operations of concatenation, union, and Kleene closure in arbitrary order, starting with the empty set, the set consisting of the empty string, and singleton sets. We will see that the regular sets are those that can be recognized using a finite-state automaton. To define regular sets we first need to define regular expressions.

Definition 1

The *regular expressions* over a set I are defined recursively by:

- the symbol \emptyset is a regular expression;
- the symbol λ is a regular expression;
- the symbol x is a regular expression whenever $x \in I$;
- the symbols (\mathbf{AB}) , $(\mathbf{A} \cup \mathbf{B})$, and \mathbf{A}^* are regular expressions whenever \mathbf{A} and \mathbf{B} are regular expressions.


Each regular expression represents a set specified by these rules:

- \emptyset represents the empty set, that is, the set with no strings;
- λ represents the set $\{\lambda\}$, which is the set containing the empty string;
- x represents the set $\{x\}$ containing the string with one symbol x ;
- (\mathbf{AB}) represents the concatenation of the sets represented by \mathbf{A} and by \mathbf{B} ;
- $(\mathbf{A} \cup \mathbf{B})$ represents the union of the sets represented by \mathbf{A} and by \mathbf{B} ;
- \mathbf{A}^* represents the Kleene closure of the set represented by \mathbf{A} .

Sets represented by regular expressions are called **regular sets**. Henceforth regular expressions will be used to describe regular sets, so when we refer to the regular set \mathbf{A} , we will mean the regular set represented by the regular expression \mathbf{A} . Note that we will leave out outer parentheses from regular expressions when they are not needed.

Example 1 shows how regular expressions are used to specify regular sets.

EXAMPLE 1 What are the strings in the regular sets specified by the regular expressions 10^* , $(10)^*$, $0 \cup 01$, $0(0 \cup 1)^*$, and $(0^*1)^*$?

Solution: The regular sets represented by these expressions are given in Table 1, as the reader should verify. 

Finding a regular expression that specifies a given set can be quite tricky, as Example 2 illustrates.

EXAMPLE 2 Find a regular expression that specifies each of these sets:

- (a) the set of bit strings with even length
- (b) the set of bit strings ending with a 0 and not containing 11
- (c) the set of bit strings containing an odd number of 0s

Solution: (a) To construct a regular expression for the set of bit strings with even length, we use the fact that such a string can be obtained by concatenating zero or more strings each consisting of two bits. The set of strings of two bits is specified by the regular expression $(00 \cup 01 \cup 10 \cup 11)$. Consequently, the set of strings with even length is specified by $(00 \cup 01 \cup 10 \cup 11)^*$.

(b) A bit string ending with a 0 and not containing 11 must be the concatenation of one or more strings where each string is either a 0 or a 10. (To see this, note that such a bit string must consist of 0 bits or 1 bits each followed by a 0; the string cannot end with a single 1 because we know it ends with a 0.) It follows that the regular expression $(0 \cup 10)^*(0 \cup 10)$ specifies the set of bit strings that do not contain 11 and end with a 0. [Note that the set specified by $(0 \cup 10)^*$ includes the empty string, which is not in this set, because the empty string does not end with a 0.]

TABLE 1

<i>Expression</i>	<i>Strings</i>
10^*	a 1 followed by any number of 0s (including no zeros)
$(10)^*$	any number of copies of 10 (including the null string)
$0 \cup 01$	the string 0 or the string 01
$0(0 \cup 1)^*$	any string beginning with 0
$(0^*1)^*$	any string not ending with 0

(c) A bit string containing an odd number of 0s must contain at least one 0, which tells us that it starts with zero or more 1s, followed by a 0, followed by zero or more 1s. That is, each such bit string begins with a string of the form $1^j 0 1^k$ for nonnegative integers j and k . Because the bit string contains an odd number of 0s, additional bits after this initial block can be split into blocks each starting with a 0 and containing one more 0. Each such block is of the form $0 1^p 0 1^q$, where p and q are nonnegative integers. Consequently, the regular expression $1^* 0 1^* (0 1^* 0 1^*)^*$ specifies the set of bit strings with an odd number of 0s. ◀

13.4.2 Kleene's Theorem

In 1956 Kleene proved that regular sets are the sets that are recognized by a finite-state automaton. Consequently, this important result is called Kleene's theorem.

THEOREM 1

KLEENE'S THEOREM A set is regular if and only if it is recognized by a finite-state automaton.

Links

Kleene's theorem is one of the central results in automata theory. We will prove the *only if* part of this theorem, namely, that every regular set is recognized by a finite-state automaton. The proof of the *if* part, that a set recognized by a finite-state automaton is regular, is left as an exercise for the reader.

Proof: Recall that a regular set is defined in terms of regular expressions, which are defined recursively. We can prove that every regular set is recognized by a finite-state automaton if we can do the following things.



1. Show that \emptyset is recognized by a finite-state automaton.
2. Show that $\{\lambda\}$ is recognized by a finite-state automaton.
3. Show that $\{a\}$ is recognized by a finite-state automaton whenever a is a symbol in I .
4. Show that AB is recognized by a finite-state automaton whenever both A and B are.
5. Show that $A \cup B$ is recognized by a finite-state automaton whenever both A and B are.
6. Show that A^* is recognized by a finite-state automaton whenever A is.

We now consider each of these tasks. First, we show that \emptyset is recognized by a nondeterministic finite-state automaton. To do this, all we need is an automaton with no final states. Such an automaton is shown in Figure 1(a).

Second, we show that $\{\lambda\}$ is recognized by a finite-state automaton. To do this, all we need is an automaton that recognizes λ , the null string, but not any other string. This can be done by making the start state s_0 a final state and having no transitions, so that no other string takes s_0 to a final state. The nondeterministic automaton in Figure 1(b) shows such a machine.

Third, we show that $\{a\}$ is recognized by a nondeterministic finite-state automaton. To do this, we can use a machine with a starting state s_0 and a final state s_1 . We have a transition from s_0 to s_1 when the input is a , and no other transitions. The only string recognized by this machine is a . This machine is shown in Figure 1(c).

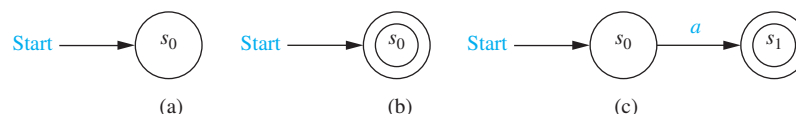


FIGURE 1 Nondeterministic finite-state automata that recognize some basic sets.

Next, we show that AB and $A \cup B$ can be recognized by finite-state automata if A and B are languages recognized by finite-state automata. Suppose that A is recognized by $M_A = (S_A, I, f_A, s_A, F_A)$ and B is recognized by $M_B = (S_B, I, f_B, s_B, F_B)$.

We begin by constructing a finite-state machine $M_{AB} = (S_{AB}, I, f_{AB}, s_{AB}, F_{AB})$ that recognizes AB , the concatenation of A and B . We build such a machine by combining the machines for A and B in series, so a string in A takes the combined machine from s_A , the start state of M_A , to s_B , the start state of M_B . A string in B should take the combined machine from s_B to a final state of the combined machine. Consequently, we make the following construction. Let S_{AB} be $S_A \cup S_B$. [Note that we can assume that S_A and S_B are disjoint.] The starting state s_{AB} is the same as s_A . The set of final states, F_{AB} , is the set of final states of M_B with s_{AB} included if and only if $\lambda \in A \cap B$. The transitions in M_{AB} include all transitions in M_A and in M_B , as well as some new transitions. For every transition in M_A that leads to a final state, we form a transition in M_{AB} from the same state to s_B , on the same input. In this way, a string in A takes M_{AB} from s_{AB} to s_B , and then a string in B takes s_B to a final state of M_{AB} . Moreover, for every transition from s_B we form a transition in M_{AB} from s_{AB} to the same state. Figure 2(a) contains an illustration of this construction.

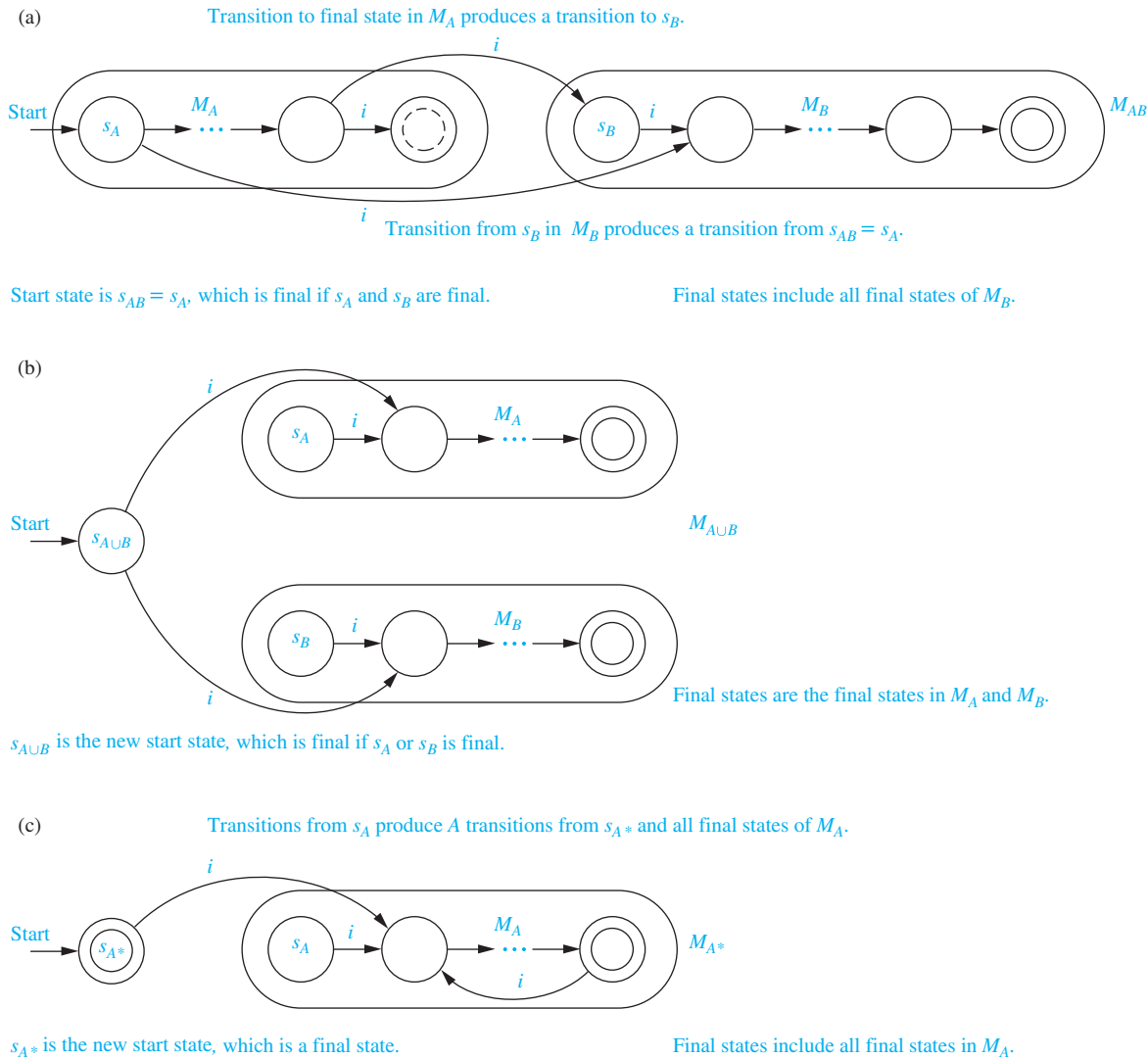


FIGURE 2 Building automata to recognize concatenations, unions, and Kleene closures.

We now construct a machine $M_{A \cup B} = (S_{A \cup B}, I, f_{A \cup B}, s_{A \cup B}, F_{A \cup B})$ that recognizes $A \cup B$. This automaton can be constructed by combining M_A and M_B in parallel, using a new start state that has the transitions that both s_A and s_B have. Let $S_{A \cup B} = S_A \cup S_B \cup \{s_{A \cup B}\}$, where $s_{A \cup B}$ is a new state that is the start state of $M_{A \cup B}$. Let the set of final states $F_{A \cup B}$ be $F_A \cup F_B \cup \{s_{A \cup B}\}$ if $\lambda \in A \cup B$, and $F_A \cup F_B$ otherwise. The transitions in $M_{A \cup B}$ include all those in M_A and in M_B . Also, for each transition from s_A to a state s on input i we include a transition from $s_{A \cup B}$ to s on input i , and for each transition from s_B to a state s on input i we include a transition from $s_{A \cup B}$ to s on input i . In this way, a string in A leads from $s_{A \cup B}$ to a final state in the new machine, and a string in B leads from $s_{A \cup B}$ to a final state in the new machine. Figure 2(b) illustrates the construction of $M_{A \cup B}$.

Finally, we construct $M_{A^*} = (S_{A^*}, I, f_{A^*}, s_{A^*}, F_{A^*})$, a machine that recognizes A^* , the Kleene closure of A . Let S_{A^*} include all states in S_A and one additional state s_{A^*} , which is the starting state for the new machine. The set of final states F_{A^*} includes all states in F_A as well as the start state s_{A^*} , because λ must be recognized. To recognize concatenations of arbitrarily many strings from A , we include all the transitions in M_A , as well as transitions from s_{A^*} that match the transitions from s_A , and transitions from each final state that match the transitions from s_A . With this set of transitions, a string made up of concatenations of strings from A will take s_{A^*} to a final state when the first string in A has been read, returning to a final state when the second string in A has been read, and so on. Figure 2(c) illustrates the construction we used. \triangleleft

A nondeterministic finite-state automaton can be constructed for any regular set using the procedure described in this proof. We illustrate how this is done with Example 3.

EXAMPLE 3 Construct a nondeterministic finite-state automaton that recognizes the regular set $1^* \cup 01$.

Solution: We begin by building a machine that recognizes 1^* . This is done using the machine that recognizes 1 and then using the construction for M_{A^*} described in the proof. Next, we build a machine that recognizes 01 , using machines that recognize 0 and 1 and the construction in the proof for M_{AB} . Finally, using the construction in the proof for $M_{A \cup B}$, we construct the machine for $1^* \cup 01$. The finite-state automata used in this construction are shown in Figure 3. The states in the successive machines have been labeled using different subscripts, even when a state is formed from one previously used in another machine. Note that the construction given here does not produce the simplest machine that recognizes $1^* \cup 01$. A much simpler machine that recognizes this set is shown in Figure 3(b). \triangleleft

13.4.3 Regular Sets and Regular Grammars

In Section 13.1 we introduced phrase-structure grammars and defined different types of grammars. In particular we defined regular, or type 3, grammars, which are grammars of the form $G = (V, T, S, P)$, where each production is of the form $S \rightarrow \lambda$, $A \rightarrow a$, or $A \rightarrow aB$, where a is a terminal symbol, and A and B are nonterminal symbols. As the terminology suggests, there is a close connection between regular grammars and regular sets.

THEOREM 2 A set is generated by a regular grammar if and only if it is a regular set.

Proof: First we show that a set generated by a regular grammar is a regular set. Suppose that $G = (V, T, S, P)$ is a regular grammar generating the set $L(G)$. To show that $L(G)$ is regular we will build a nondeterministic finite-state machine $M = (S, I, f, s_0, F)$ that recognizes $L(G)$. Let S , the set of states, contain a state s_A for each nonterminal symbol A of G and an additional state s_F , which is a final state. The start state s_0 is the state formed from the start symbol S . The transitions of M are formed from the productions of G in the following way. A transition

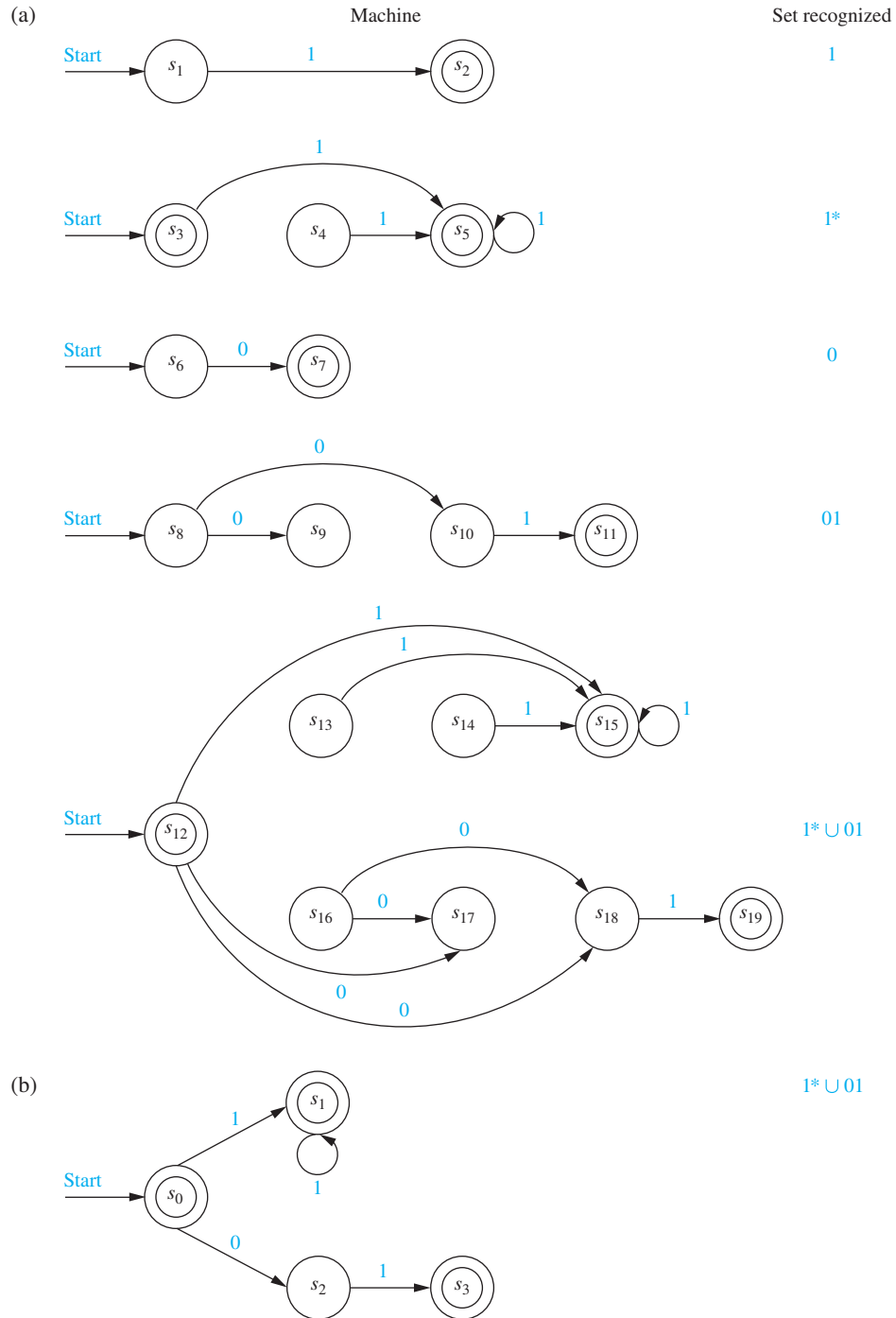


FIGURE 3 Nondeterministic finite-state automata recognizing $1^* \cup 01$.

from s_A to s_F on input of a is included if $A \rightarrow a$ is a production, and a transition from s_A to s_B on input of a is included if $A \rightarrow aB$ is a production. The set of final states includes s_F and also includes s_0 if $S \rightarrow \lambda$ is a production in G . It is not hard to show that the language recognized by M equals the language generated by the grammar G , that is, $L(M) = L(G)$. This can be done by determining the words that lead to a final state. The details are left as an exercise for the reader. ◀

Before giving the proof of the converse, we illustrate how a nondeterministic machine is constructed that recognizes the same set as a regular grammar.

EXAMPLE 4 Construct a nondeterministic finite-state automaton that recognizes the language generated by the regular grammar $G = (V, T, S, P)$, where $V = \{0, 1, A, S\}$, $T = \{0, 1\}$, and the productions in P are $S \rightarrow 1A$, $S \rightarrow 0$, $S \rightarrow \lambda$, $A \rightarrow 0A$, $A \rightarrow 1A$, and $A \rightarrow 1$.

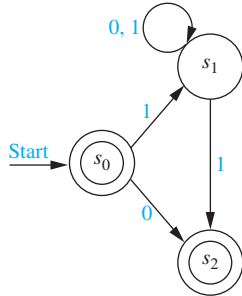


FIGURE 4 A nondeterministic finite-state automaton recognizing $L(G)$.

Solution: The state diagram for a nondeterministic finite-state automaton that recognizes $L(G)$ is shown in Figure 4. This automaton is constructed following the procedure described in the proof. In this automaton, s_0 is the state corresponding to S , s_1 is the state corresponding to A , and s_2 is the final state.

We now complete the proof of Theorem 2.

Proof: We now show that if a set is regular, then there is a regular grammar that generates it. Suppose that M is a finite-state machine that recognizes this set with the property that s_0 , the starting state of M , is never the next state for a transition. (We can find such a machine by Exercise 20.) The grammar $G = (V, T, S, P)$ is defined as follows. The set V of symbols of G is formed by assigning a symbol to each state of S and each input symbol in I . The set T of terminal symbols of G is the set I . The start symbol S is the symbol formed from the start state s_0 . The set P of productions in G is formed from the transitions in M . In particular, if the state s goes to a final state under input a , then the production $A_s \rightarrow a$ is included in P , where A_s is the nonterminal symbol formed from the state s . If the state s goes to the state t on input a , then the production $A_s \rightarrow aA_t$ is included in P . The production $S \rightarrow \lambda$ is included in P if and only if $\lambda \in L(M)$. Because the productions of G correspond to the transitions of M and the productions leading to terminals correspond to transitions to final states, it is not hard to show that $L(G) = L(M)$. We leave the details as an exercise for the reader.

Example 5 illustrates the construction used to produce a grammar from an automaton that generates the language recognized by this automaton.

EXAMPLE 5 Find a regular grammar that generates the regular set recognized by the finite-state automaton shown in Figure 5.

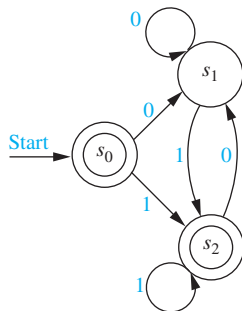


FIGURE 5 A finite-state automaton.

Solution: The grammar $G = (V, T, S, P)$ generates the set recognized by this automaton where $V = \{S, A, B, 0, 1\}$, the symbols S, A , and B correspond to the states s_0, s_1 , and s_2 , respectively, $T = \{0, 1\}$, S is the start symbol; and the productions are $S \rightarrow 0A$, $S \rightarrow 1B$, $S \rightarrow \lambda$, $A \rightarrow 0A$, $A \rightarrow 1B$, $A \rightarrow 1$, $B \rightarrow 0A$, $B \rightarrow 1B$, and $B \rightarrow 1$.

13.4.4 A Set Not Recognized by a Finite-State Automaton

We have seen that a set is recognized by a finite state automaton if and only if it is regular. We will now show that there are sets that are not regular by describing one such set. The technique used to show that this set is not regular illustrates an important method for showing that certain sets are not regular.

EXAMPLE 6 Show that the set $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$, made up of all strings consisting of a block of 0s followed by a block of an equal number of 1s, is not regular.

Solution: Suppose that this set were regular. Then there would be a nondeterministic finite-state automaton $M = (S, I, f, s_0, F)$ recognizing it. Let N be the number of states in this machine,



13.4.5 More Powerful Types of Machines

Links

Finite-state automata are unable to carry out many computations. The main limitation of these machines is their finite amount of memory. This prevents them from recognizing languages that are not regular, such as $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$. Because a set is regular if and only if it is the language generated by a regular grammar, Example 6 shows that there is no regular grammar that generates the set $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$. However, there is a context-free grammar that generates this set. Such a grammar was given in Example 5 in Section 13.1.

Because of the limitations of finite-state machines, it is necessary to use other, more powerful, models of computation. One such model is the **pushdown automaton**. A pushdown automaton includes everything in a finite-state automaton, as well as a stack, which provides unlimited memory. Symbols can be placed on the top or taken off the top of the stack. A set is recognized in one of two ways by a pushdown automaton. First, a set is recognized if the set consists of all the strings that produce an empty stack when they are used as input. Second, a set is recognized if it consists of all the strings that lead to a final state when used as input. It can be shown that a set is recognized by a pushdown automaton if and only if it is the language generated by a context-free grammar.

However, there are sets that cannot be expressed as the language generated by a context-free grammar. One such set is $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$. We will indicate why this set cannot be recognized by a pushdown automaton, but we will not give a proof, because we have not developed the machinery needed. (However, one method of proof is given in Exercise 28 of the supplementary exercises at the end of this chapter.) The stack can be used to show that a string begins with a sequence of 0s followed by an equal number of 1s by placing a symbol on the stack for each 0 (as long as only 0s are read), and removing one of these symbols for each 1 (as long as only 1s following the 0s are read). But once this is done, the stack is empty, and there is no way to determine that there are the same number of 2s in the string as 0s.

There are other machines called **linear bounded automata**, more powerful than pushdown automata, that can recognize sets such as $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$. In particular, linear bounded automata can recognize context-sensitive languages. However, these machines cannot recognize all the languages generated by phrase-structure grammars. To avoid the limitations of special types of machines, the model known as a **Turing machine**, named after the British mathematician Alan Turing, is used. A Turing machine is made up of everything included in a finite-state machine together with a tape, which is infinite in both directions. A Turing machine has read and write capabilities on the tape, and it can move back and forth along this tape. Turing machines can recognize all languages generated by phrase-structure grammars. In addition, Turing machines can model all the computations that can be performed on a computing machine. Because of their power, Turing machines are extensively studied in theoretical computer science. We will briefly study them in Section 13.5.

Alan Turing invented Turing machines before modern computers existed!

Exercises

- Describe in words the strings in each of these regular sets.

a) 1^*0	b) 1^*00^*	e) $(01)^*(11)^*$	f) $0^*(10 \cup 11)^*$
c) $111 \cup 001$	d) $(1 \cup 00)^*$	g) $0^*(10)^*11$	h) $01(01 \cup 0)1^*$
e) $(00^*1)^*$	f) $(0 \cup 1)(0 \cup 1)^*00$		
- Describe in words the strings in each of these regular sets.

a) 001^*	b) $(01)^*$	a) 10^*1^*	b) $0^*(10 \cup 11)^*$
c) $01 \cup 001^*$	d) $0(11 \cup 0)^*$	c) $1(01)^*1^*$	d) $1^*01(0 \cup 1)$
e) $(101^*)^*$	f) $(0^* \cup 1)11$	e) $(10)^*(11)^*$	f) $1(00)^*(11)^*$
		g) $(10)^*1011$	h) $(1 \cup 00)(01 \cup 0)1^*$
- Determine whether 0101 belongs to each of these regular sets.

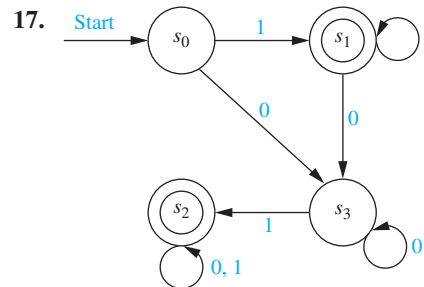
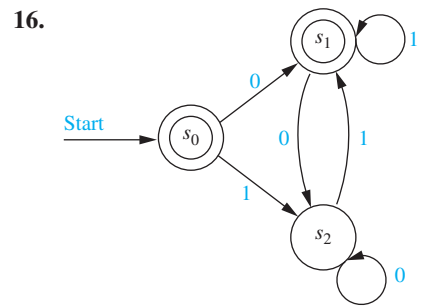
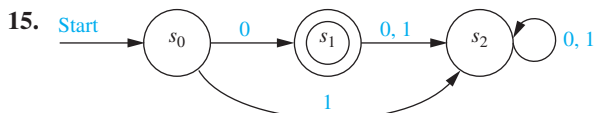
a) 01^*0^*	b) $0(11)^*(01)^*$
c) $0(10)^*1^*$	d) $0^*10(0 \cup 1)$
- Determine whether 1011 belongs to each of these regular sets.

a) 10^*1^*	b) $0^*(10 \cup 11)^*$
c) $1(01)^*1^*$	d) $1^*01(0 \cup 1)$
e) $(10)^*(11)^*$	f) $1(00)^*(11)^*$
g) $(10)^*1011$	h) $(1 \cup 00)(01 \cup 0)1^*$
- Express each of these sets using a regular expression.

a) the set consisting of the strings 0, 11, and 010
b) the set of strings of three 0s followed by two or more 0s
c) the set of strings of odd length

- d) the set of strings that contain exactly one 1
 e) the set of strings ending in 1 and not containing 000
6. Express each of these sets using a regular expression.
- the set containing all strings with zero, one, or two bits
 - the set of strings of two 0s, followed by zero or more 1s, and ending with a 0
 - the set of strings with every 1 followed by two 0s
 - the set of strings ending in 00 and not containing 11
 - the set of strings containing an even number of 1s
7. Express each of these sets using a regular expression.
- the set of strings of one or more 0s followed by a 1
 - the set of strings of two or more symbols followed by three or more 0s
 - the set of strings with either no 1 preceding a 0 or no 0 preceding a 1
 - the set of strings containing a string of 1s such that the number of 1s equals 2 modulo 3, followed by an even number of 0s
8. Construct deterministic finite-state automata that recognize each of these sets from I^* , where I is an alphabet.
- \emptyset
 - $\{\lambda\}$
 - $\{a\}$, where $a \in I$
9. Construct nondeterministic finite-state automata that recognize each of the sets in Exercise 8.
10. Construct nondeterministic finite-state automata that recognize each of these sets.
- $\{\lambda, 0\}$
 - $\{0, 11\}$
 - $\{0, 11, 000\}$
- *11. Show that if A is a regular set, then A^R , the set of all reversals of strings in A , is also regular.
12. Using the constructions described in the proof of Kleene's theorem, find nondeterministic finite-state automata that recognize each of these sets.
- 01^*
 - $(0 \cup 1)1^*$
 - $00(1^* \cup 10)$
13. Using the constructions described in the proof of Kleene's theorem, find nondeterministic finite-state automata that recognize each of these sets.
- 0^*1^*
 - $(0 \cup 11)^*$
 - $01^* \cup 00^*1$
14. Construct a nondeterministic finite-state automaton that recognizes the language generated by the regular grammar $G = (V, T, S, P)$, where $V = \{0, 1, S, A, B\}$, $T = \{0, 1\}$, S is the start symbol, and the set of productions is
- $S \rightarrow 0A, S \rightarrow 1B, A \rightarrow 0, B \rightarrow 0.$
 - $S \rightarrow 1A, S \rightarrow 0, S \rightarrow \lambda, A \rightarrow 0B, B \rightarrow 1B, B \rightarrow 1.$
 - $S \rightarrow 1B, S \rightarrow 0, A \rightarrow 1A, A \rightarrow 0B, A \rightarrow 1, A \rightarrow 0, B \rightarrow 1.$

In Exercises 15–17 construct a regular grammar $G = (V, T, S, P)$ that generates the language recognized by the given finite-state machine.



18. Show that the finite-state automaton constructed from a regular grammar in the proof of Theorem 2 recognizes the set generated by this grammar.
19. Show that the regular grammar constructed from a finite-state automaton in the proof of Theorem 2 generates the set recognized by this automaton.
20. Show that every nondeterministic finite-state automaton is equivalent to another such automaton that has the property that its starting state is never revisited.
- *21. Let $M = (S, I, f, s_0, F)$ be a deterministic finite-state automaton. Show that the language recognized by M , $L(M)$, is infinite if and only if there is a word x recognized by M with $l(x) \geq |S|$.
- *22. One important technique used to prove that certain sets are not regular is the **pumping lemma**. The pumping lemma states that if $M = (S, I, f, s_0, F)$ is a deterministic finite-state automaton and if x is a string in $L(M)$, the language recognized by M , with $l(x) \geq |S|$, then there are strings u, v , and w in I^* such that $x = uvw$, $l(uv) \leq |S|$ and $l(v) \geq 1$, and $uv^i w \in L(M)$ for $i = 0, 1, 2, \dots$. Prove the pumping lemma. [Hint: Use the same idea as was used in Example 5.]
- *23. Show that the set $\{0^{2^n}1^n \mid n = 0, 1, 2, \dots\}$ is not regular using the pumping lemma given in Exercise 22.
- *24. Show that the set $\{1^{n^2} \mid n = 0, 1, 2, \dots\}$ is not regular using the pumping lemma from Exercise 22.
- *25. Show that the set of palindromes over $\{0, 1\}$ is not regular using the pumping lemma given in Exercise 22. [Hint: Consider strings of the form $0^N 10^N$.]
- **26. Show that a set recognized by a finite-state automaton is regular. (This is the *if* part of Kleene's theorem.)



Suppose that L is a subset of I^* , where I is a nonempty set of symbols. If $x \in I^*$, we let $L/x = \{z \in I^* \mid xz \in L\}$. We say that the strings $x \in I^*$ and $y \in I^*$ are **distinguishable with respect to L** if $L/x \neq L/y$. A string z for which $xz \in L$ but $yz \notin L$, or $xz \notin L$, but $yz \in L$ is said to **distinguish** x and y with respect to L . When $L/x = L/y$, we say that x and y are **indistinguishable** with respect to L .

27. Let L be the set of all bit strings that end with 01. Show that 11 and 10 are distinguishable with respect to L and that the strings 1 and 11 are indistinguishable with respect to L .
28. Suppose that $M = (S, I, f, s_0, F)$ is a deterministic finite-state machine. Show that if x and y are two strings in I^* that are distinguishable with respect to $L(M)$, then $f(s_0, x) \neq f(s_0, y)$.
- *29. Suppose that L is a subset of I^* and for some positive integer n there are n strings in I^* such that every two of these strings are distinguishable with respect to L . Prove that every deterministic finite-state automaton recognizing L has at least n states.
- *30. Let L_n be the set of strings with at least n bits in which the n th symbol from the end is a 0. Use Exercise 29 to show that a deterministic finite-state machine recognizing L_n must have at least 2^n states.
- *31. Use Exercise 29 to show that the language consisting of all bit strings that are palindromes (that is, strings that equal their own reversals) is not regular.

13.5 Turing Machines

13.5.1 Introduction



The finite-state automata studied earlier in this chapter cannot be used as general models of computation. They are limited in what they can do. For example, finite-state automata are able to recognize regular sets, but are not able to recognize many easy-to-describe sets, including $\{0^n 1^n \mid n \geq 0\}$, which computers recognize using memory. We can use finite-state automata to compute relatively simple functions such as the sum of two numbers, but we cannot use them to compute functions that computers can, such as the product of two numbers. To overcome these deficiencies we can use a more powerful type of machine known as a Turing machine, after Alan Turing, the illustrious mathematician and computer scientist who invented them in the 1930s.

Basically, a Turing machine consists of a control unit, which at any step is in one of finitely many different states, together with a tape divided into cells, which is infinite in both directions. Turing machines have read and write capabilities on the tape as the control unit moves back and forth along this tape, changing states depending on the tape symbol read. Turing machines are more powerful than finite-state machines because they include memory capabilities that finite-state machines lack. We will show how to use Turing machines to recognize sets, including sets that cannot be recognized by finite-state machines. We will also show how to compute functions using Turing machines. Turing machines are the most general models of computation; essentially, they can do whatever a computer can do. Note that Turing machines are much more powerful than real computers, which have finite memory capabilities.

“Machines take me by surprise with great frequency.” – Alan Turing

13.5.2 Definition of Turing Machines

We now give the formal definition of a Turing machine. Afterward we will explain how this formal definition can be interpreted in terms of a control head that can read and write symbols on a tape and move either right or left along the tape.

Definition 1

A Turing machine $T = (S, I, f, s_0)$ consists of a finite set S of states, an alphabet I containing the blank symbol B , a partial function f from $S \times I$ to $S \times I \times \{R, L\}$, and a starting state s_0 .

Recall from Section 2.3 that a partial function is defined only for those elements in its domain of definition. This means that for some (state, symbol) pairs the partial function f may be

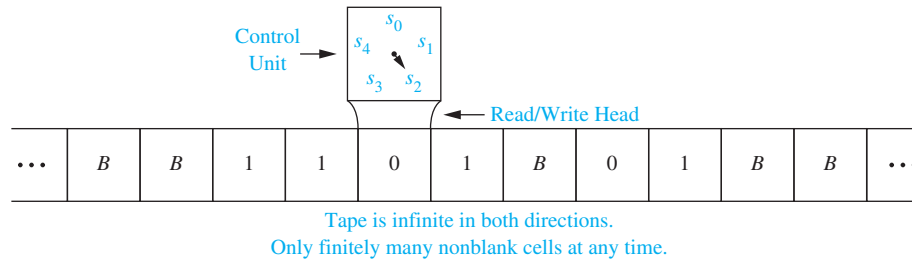


FIGURE 1 A representation of a Turing machine.

undefined, but for a pair for which it is defined, there is a unique (state, symbol, direction) triple associated to this pair. We call the five-tuples corresponding to the partial function in the definition of a Turing machine the **transition rules** of the machine.

To interpret this definition in terms of a machine, consider a control unit and a tape divided into cells, infinite in both directions, having only a finite number of nonblank symbols on it at any given time, as pictured in Figure 1. The action of the Turing machine at each step of its operation depends on the value of the partial function f for the current state and tape symbol.

At each step, the control unit reads the current tape symbol x . If the control unit is in state s and if the partial function f is defined for the pair (s, x) with $f(s, x) = (s', x', d)$, the control unit

1. enters the state s' ,
2. writes the symbol x' in the current cell, erasing x , and
3. moves right one cell if $d = R$ or moves left one cell if $d = L$.

We write this step as the five-tuple (s, x, s', x', d) . If the partial function f is undefined for the pair (s, x) , then the Turing machine T will *halt*.

A common way to define a Turing machine is to specify a set of five-tuples of the form (s, x, s', x', d) . The set of states and input alphabet is implicitly defined when such a definition is used.

At the beginning of its operation a Turing machine is assumed to be in the initial state s_0 and to be positioned over the leftmost nonblank symbol on the tape. If the tape is all blank, the control head can be positioned over any cell. We will call the positioning of the control head over the leftmost nonblank tape symbol the *initial position* of the machine.

Example 1 illustrates how a Turing machine works.

EXAMPLE 1 What is the final tape when the Turing machine T defined by the seven five-tuples $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, (s_0, B, s_3, B, R) , $(s_1, 0, s_0, 0, R)$, $(s_1, 1, s_2, 0, L)$, (s_1, B, s_3, B, R) , and $(s_2, 1, s_3, 0, R)$ is run on the tape shown in Figure 2(a)?

Extra Examples

Solution: We start the operation with T in state s_0 and with T positioned over the leftmost nonblank symbol on the tape. The first step, using the five-tuple $(s_0, 0, s_0, 0, R)$, reads the 0 in the leftmost nonblank cell, stays in state s_0 , writes a 0 in this cell, and moves one cell right. The second step, using the five-tuple $(s_0, 1, s_1, 1, R)$, reads the 1 in the current cell, enters state s_1 , writes a 1 in this cell, and moves one cell right. The third step, using the five-tuple $(s_1, 0, s_0, 0, R)$, reads the 0 in the current cell, enters state s_0 , writes a 0 in this cell, and moves one cell right. The fourth step, using the five-tuple $(s_0, 1, s_1, 1, R)$, reads the 1 in the current cell, enters state s_1 , writes a 1 in this cell, and moves right one cell. The fifth step, using the five-tuple $(s_1, 1, s_2, 0, L)$, reads the 1 in the current cell, enters state s_2 , writes a 0 in this cell, and moves left one cell. The sixth step, using the five-tuple $(s_2, 1, s_3, 0, R)$, reads the 1 in the current cell, enters the state s_3 , writes a 0 in this cell, and moves right one cell. Finally, in the seventh step, the machine halts because there is no five-tuple beginning with the pair $(s_3, 0)$ in the description of the machine. The steps are shown in Figure 2.

Note that T changes the first pair of consecutive 1s on the tape to 0s and then halts. ◀

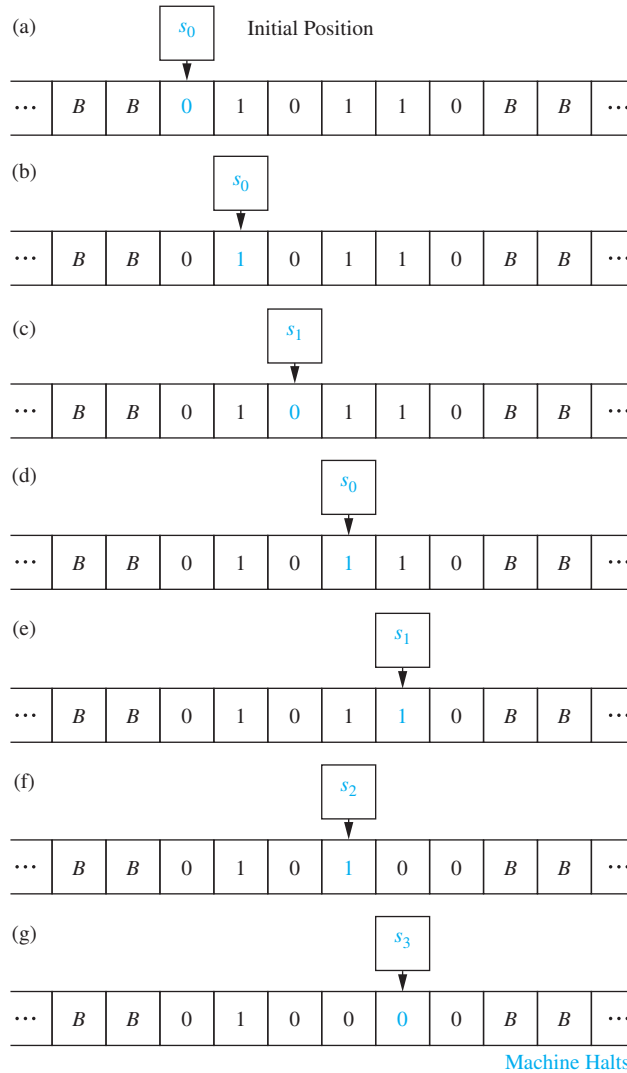


FIGURE 2 The steps produced by running T on the tape in Figure 1.

13.5.3 Using Turing Machines to Recognize Sets

Turing machines can be used to recognize sets. To do so requires that we define the concept of a final state as follows. A *final state* of a Turing machine T is a state that is not the first state in any five-tuple in the description of T using five-tuples (for example, state s_3 in Example 1).

We can now define what it means for a Turing machine to recognize a string. Given a string, we write consecutive symbols in this string in consecutive cells.

Definition 2

Let V be a subset of an alphabet I . A Turing machine $T = (S, I, f, s_0)$ recognizes a string x in V^* if and only if T , starting in the initial position when x is written on the tape, halts in a final state. T is said to recognize a subset A of V^* if x is recognized by T if and only if x belongs to A .

Note that to recognize a subset A of V^* we can use symbols not in V . This means that the input alphabet I may include symbols not in V . These extra symbols are often used as markers (see Example 3).


When does a Turing machine T *not* recognize a string x in V^* ? The answer is that x is not recognized if T does not halt or halts in a state that is not final when it operates on a tape containing the symbols of x in consecutive cells, starting in the initial position. (The reader should understand that this is one of many possible ways to define how to recognize sets using Turing machines.)

We illustrate this concept with Example 2.

EXAMPLE 2 Find a Turing machine that recognizes the set of bit strings that have a 1 as their second bit, that is, the regular set $(0 \cup 1)1(0 \cup 1)^*$.

Solution: We want a Turing machine that, starting at the leftmost nonblank tape cell, moves right, and determines whether the second symbol is a 1. If the second symbol is 1, the machine should move into a final state. If the second symbol is not a 1, the machine should not halt or it should halt in a nonfinal state.

To construct such a machine, we include the five-tuples $(s_0, 0, s_1, 0, R)$ and $(s_0, 1, s_1, 1, R)$ to read in the first symbol and put the Turing machine in state s_1 . Next, we include the five-tuples $(s_1, 0, s_2, 0, R)$ and $(s_1, 1, s_3, 1, R)$ to read in the second symbol and either move to state s_2 if this symbol is a 0, or to state s_3 if this symbol is a 1. We do not want to recognize strings that have a 0 as their second bit, so s_2 should not be a final state. We want s_3 to be a final state. So, we can include the five-tuple $(s_2, 0, s_2, 0, R)$. Because we do not want to recognize the empty string or a string with one bit, we also include the five-tuples $(s_0, B, s_2, 0, R)$ and $(s_1, B, s_2, 0, R)$.

The Turing machine T consisting of the seven five-tuples listed here will terminate in the final state s_3 if and only if the bit string has at least two bits and the second bit of the input string is a 1. If the bit string contains fewer than two bits or if the second bit is not a 1, the machine will terminate in the nonfinal state s_2 . 

Given a regular set, a Turing machine that always moves to the right can be built to recognize this set (as in Example 2). To build the Turing machine, first find a finite-state automaton that recognizes the set and then construct a Turing machine using the transition function of the finite-state machine, always moving to the right.


We will now show how to build a Turing machine that recognizes a nonregular set.

EXAMPLE 3 Find a Turing machine that recognizes the set $\{0^n 1^n \mid n \geq 1\}$.



Solution: To build such a machine, we will use an auxiliary tape symbol M as a marker. We have $V = \{0, 1\}$ and $I = \{0, 1, M\}$. We wish to recognize only a subset of strings in V^* . We will have one final state, s_6 . The Turing machine successively replaces a 0 at the leftmost position of the string with an M and a 1 at the rightmost position of the string with an M , sweeping back and forth, terminating in a final state if and only if the string consists of a block of 0s followed by a block of the same number of 1s.

Although this is easy to describe and is easily carried out by a Turing machine, the machine we need to use is somewhat complicated. We use the marker M to keep track of the leftmost and rightmost symbols we have already examined. The five-tuples we use are $(s_0, 0, s_1, M, R)$, $(s_1, 0, s_1, 0, R)$, $(s_1, 1, s_1, 1, R)$, (s_1, M, s_2, M, L) , (s_1, B, s_2, B, L) , $(s_2, 1, s_3, M, L)$, $(s_3, 1, s_3, 1, L)$, $(s_3, 0, s_4, 0, L)$, (s_3, M, s_5, M, R) , $(s_4, 0, s_4, 0, L)$, (s_4, M, s_0, M, R) , and (s_5, M, s_6, M, R) . For example, the string 000111 would successively become $M00111$, $M0011M$, $MM011M$, $MM01MM$, $MMM1MM$, $MMMMMM$ as the machine operates until it halts. Only the changes are shown, as most steps leave the string unaltered.

We leave it to the reader (Exercise 13) to explain the actions of this Turing machine and to explain why it recognizes the set $\{0^n 1^n \mid n \geq 1\}$. 

It can be shown that a set can be recognized by a Turing machine if and only if it can be generated by a type 0 grammar, or in other words, if the set is generated by a phrase-structure grammar. The proof will not be presented here.

13.5.4 Computing Functions with Turing Machines

A Turing machine can be thought of as a computer that finds the values of a partial function. To see this, suppose that the Turing machine T , when given the string x as input, halts with the string y on its tape. We can then define $T(x) = y$. The domain of T is the set of strings for which T halts; $T(x)$ is undefined if T does not halt when given x as input. Thinking of a Turing machine as a machine that computes the values of a function on strings is useful, but how can we use Turing machines to compute functions defined on integers, on pairs of integers, on triples of integers, and so on?

To consider a Turing machine as a computer of functions from the set of k -tuples of nonnegative integers to the set of nonnegative integers (such functions are called **number-theoretic functions**), we need a way to represent k -tuples of integers on a tape. To do so, we use **unary representations** of integers. We represent the nonnegative integer n by a string of $n + 1$ 1s so that, for instance, 0 is represented by the string 1 and 5 is represented by the string 11111. To represent the k -tuple (n_1, n_2, \dots, n_k) , we use a string of $n_1 + 1$ 1s, followed by an asterisk, followed by a string of $n_2 + 1$ 1s, followed by an asterisk, and so on, ending with a string of $n_k + 1$ 1s. For example, to represent the four-tuple $(2, 0, 1, 3)$ we use the string 111 * 1 * 11 * 1111.

We can now consider a Turing machine T as computing a sequence of number-theoretic functions $T, T^2, \dots, T^k, \dots$. The function T^k is defined by the action of T on k -tuples of integers represented by unary representations of integers separated by asterisks.

EXAMPLE 4 Construct a Turing machine for adding two nonnegative integers.

Solution: We need to build a Turing machine T that computes the function $f(n_1, n_2) = n_1 + n_2$. The pair (n_1, n_2) is represented by a string of $n_1 + 1$ 1s followed by an asterisk followed by $n_2 + 1$ 1s. The machine T should take this as input and produce as output a tape with $n_1 + n_2 + 1$ 1s. One way to do this is as follows. The machine starts at the leftmost 1 of the input string, and carries out steps to erase this 1, halting if $n_1 = 0$ so that there are no more 1s before the asterisk, replaces the asterisk with the leftmost remaining 1, and then halts. We can use these five-tuples to do this: $(s_0, 1, s_1, B, R)$, $(s_1, *, s_2, B, R)$, $(s_1, 1, s_2, B, R)$, $(s_2, 1, s_2, 1, R)$, and $(s_2, *, s_3, 1, R)$. ◀

Unfortunately, constructing Turing machines to compute relatively simple functions can be extremely demanding. For example, one Turing machine for multiplying two nonnegative integers found in many books has 31 five-tuples and 11 states. If it is challenging to construct Turing machines to compute even relatively simple functions, what hope do we have of building Turing machines for more complicated functions? One way to simplify this problem is to use a multitape Turing machine that uses more than one tape simultaneously and to build up multitape Turing machines for the composition of functions. It can be shown that for any multitape Turing machine there is a one-tape Turing machine that can do the same thing.

13.5.5 Different Types of Turing Machines

There are many variations on the definition of a Turing machine. We can expand the capabilities of a Turing machine in a wide variety of ways. For example, we can allow a Turing machine to move right, left, or not at all at each step. We can allow a Turing machine to operate on multiple tapes, using $(2 + 3n)$ -tuples to describe the Turing machine when n tapes are used. We can allow the tape to be two-dimensional, where at each step we move up, down, right, or left, not just right or left as we do on a one-dimensional tape. We can allow multiple tape heads that read different cells simultaneously. Furthermore, we can allow a Turing machine to be **nondeterministic**, by

allowing a (state, tape symbol) pair to possibly appear as the first elements in more than one five-tuple of the Turing machine. So, unlike a deterministic Turing machine, which has at most one prescribed action for a given (state, tape symbol) pair, a nondeterministic Turing machine has a set of prescribed actions for such a pair. Hence, a nondeterministic Turing machine branches into multiple copies as it runs, each of which follows a sequence of possible transitions. We can also reduce the capabilities of a Turing machine in different ways. For example, we can restrict the tape to be infinite in only one direction or we can restrict the tape alphabet to have only two symbols. All these variations of Turing machines have been studied in detail.

The crucial point is that no matter which of these variations we use, or even which combination of variations we use, we never increase or decrease the power of the machine. Anything that one of these variations can do can be done by the Turing machine defined in this section, and vice versa. The reason that these variations are useful is that sometimes they make doing some particular job much easier than if the Turing machine defined in Definition 1 were used. They never extend the capability of the machine. Sometimes it is useful to have a wide variety of Turing machines with which to work. For example, one way to show that for every nondeterministic Turing machine, there is a deterministic Turing machine that recognizes the same language is to use a deterministic Turing machine with three tapes. (For details on variations of Turing machines and demonstrations of their equivalence, see [HoMoUI01].)

Besides introducing the notion of a Turing machine, Turing also showed that it is possible to construct a single Turing machine that can simulate the computations of every Turing machine when given an encoding of this target Turing machine and its input. Such a machine is called a **universal Turing machine**. (See a book on the theory of computation, such as [Si06], for more about universal Turing machines.)

13.5.6 The Church–Turing Thesis

Links

Turing machines are relatively simple. They can have only finitely many states and they can read and write only one symbol at a time on a one-dimensional tape. But it turns out that Turing machines are extremely powerful. We have seen that Turing machines can be built to add numbers and to multiply numbers. Although it may be difficult to actually construct a Turing machine to compute a particular function that can be computed with an algorithm, such a Turing machine can always be found. This was the original goal of Turing when he invented his machines. Furthermore, there is a tremendous amount of evidence for the **Church–Turing thesis**, which states that given any problem that can be solved with an effective algorithm, there is a Turing machine that can solve this problem. The reason this is called a *thesis* rather than a theorem is that the concept of solvability by an effective algorithm is informal and imprecise, as opposed to the notion of solvability by a Turing machine, which is formal and precise. Certainly, though, any problem that can be solved using a computer with a program written in any language, perhaps using an unlimited amount of memory, should be considered effectively solvable. (Note that Turing machines have unlimited memory, unlike computers in the real world, which have only a finite amount of memory.)

Many different formal theories have been developed to capture the notion of effective computability. These include Turing’s theory and Church’s lambda-calculus, as well as theories proposed by Stephen Kleene and by E. L. Post. These theories seem quite different on the surface. The surprising thing is that they can be shown to be equivalent by demonstrating that they define exactly the same class of functions. With this evidence, it seems that Turing’s original ideas, formulated before the invention of modern computers, describe the ultimate capabilities of these machines. The interested reader should consult books on the theory of computation, such as [HoMoUI01] and [Si96], for a discussion of these different theories and their equivalence.

For the remainder of this section we will briefly explore some of the consequences of the Church–Turing thesis and we will describe the importance of Turing machines in the study of the complexity of algorithms. Our goal will be to introduce some important ideas from theoretical computer science to entice the interested student to further study. We will cover a lot of ground

quickly without providing explicit details. Our discussion will also tie together some of the concepts discussed in previous parts of the book with the theory of computation.

13.5.7 Computational Complexity, Computability, and Decidability

Throughout this book we have discussed the computational complexity of a wide variety of problems. We described the complexity of these problems in terms of the number of operations used by the most efficient algorithms that solve them. The basic operations used by algorithms differ considerably; we have measured the complexity of different algorithms in terms of bit operations, comparisons of integers, arithmetic operations, and so on. In Section 3.3, we defined various classes of problems in terms of their computational complexity. However, these definitions were not precise, because the types of operations used to measure their complexity vary so drastically. Turing machines provide a way to make the concept of computational complexity precise. If the Church–Turing thesis is true, it would then follow that if a problem can be solved using an effective algorithm, then there is a Turing machine that can solve this problem. When a Turing machine is used to solve a problem, the input to the problem is encoded as a string of symbols that is written on the tape of this Turing machine. How we encode input depends on the domain of this input. For example, as we have seen, we can encode a positive integer using a string of 1s. We can also devise ways to express pairs of integers, negative integers, and so on. Similarly, for graph algorithms, we need a way to encode graphs as strings of symbols. This can be done in many ways and can be based on adjacency lists or adjacency matrices. (We omit the details of how this is done.) However, the way input is encoded does not matter as long as it is relatively efficient, as a Turing machine can always change one encoding into another encoding. We will now use this model to make precise some of the notions concerning computational complexity that were informally introduced in Section 3.3.

The kind of problems that are most easily studied by using Turing machines are those problems that can be answered either by a “yes” or by a “no.”

Definition 3

A decision problem asks whether statements from a particular class of statements are true. Decision problems are also known as yes-or-no problems.

Given a decision problem, we would like to know whether there is an algorithm that can determine whether statements from the class of statements it addresses are true. For example, consider the class of statements each of which asks whether a particular integer n is prime. This is a decision problem because the answer to the question “Is n prime?” is either yes or no. Given this decision problem, we can ask whether there is an algorithm that can decide whether each of the statements in the decision problem is true, that is, given an integer n , deciding whether n is prime. The answer is that there is such an algorithm. In particular, in Section 3.5 we discussed the algorithm that determines whether a positive integer n is prime by checking whether it is divisible by primes not exceeding its square root. (There are many other algorithms for determining whether a positive integer is prime.)

The set of inputs for which the answer to the yes–no problem is “yes” is a subset of the set of possible inputs, that is, it is a subset of the set of strings of the input alphabet. In other words, solving a yes–no problem is the same as recognizing the language consisting of all bit strings that represent input values to the problem leading to the answer “yes.” Consequently, solving a yes–no problem is the same as recognizing the language corresponding to the input values for which the answer to the problem is “yes.”

DECIDABILITY When there is an effective algorithm that decides whether instances of a decision problem are true, we say that this problem is **solvable** or **decidable**. For instance, the problem of determining whether a positive integer is prime is a solvable problem. However, if no

effective algorithm exists for solving a problem, then we say the problem is **unsolvable** or **undecidable**. To show that a decision problem is solvable we need only construct an algorithm that can determine whether statements of the particular class are true. On the other hand, to show that a decision problem is unsolvable we need to prove that no such algorithm exists. (The fact that we tried to find such an algorithm but failed, does not prove the problem is unsolvable.)

By studying only decision problems, it may seem that we are studying only a small set of problems of interest. However, most problems can be recast as decision problems. Recasting the types of problems we have studied in this book as decision problems can be quite complicated, so we will not go into the details of this process here. The interested reader can consult references on the theory of computation, such as [Wo87], which, for example, explains how to recast the traveling salesperson problem (described in Section 9.6) as a decision problem. (To recast the traveling salesman problem as a decision problem, we first consider the decision problem that asks whether there is a Hamilton circuit of weight not exceeding k , where k is a positive integer. With some additional effort it is possible to use answers to this question for different values of k to find the smallest possible weight of a Hamilton circuit.)

In Section 3.1 we introduced the halting problem and proved that it is an unsolvable problem. That discussion was somewhat informal because the notion of a procedure was not precisely defined. A precise definition of the halting problem can be made in terms of Turing machines.

Definition 4

The *halting problem* is the decision problem that asks whether a Turing machine T eventually halts when given an input string x .

With this definition of the halting problem, we have Theorem 1.

THEOREM 1

The halting problem is an unsolvable decision problem. That is, no Turing machine exists that, when given an encoding of a Turing machine T and its input string x as input, can determine whether T eventually halts when started with x written on its tape.

The proof of Theorem 1 given in Section 3.1 for the informal definition of the halting problem still applies here.

Other examples of unsolvable problems include:

- (i) the problem of determining whether two context-free grammars generate the same set of strings;
- (ii) the problem of determining whether a given set of tiles can be used with repetition allowed to cover the entire plane without overlap; and
- (iii) *Hilbert's Tenth Problem*, which asks whether there are integer solutions to a given polynomial equation with integer coefficients. (This question occurs tenth on the influential list of 23 unsolved problems Hilbert posed in 1900. Hilbert envisioned that the work done to solve these problems would help further the progress of mathematics in the twentieth century. The unsolvability of Hilbert's Tenth Problem was established in 1970 by Yuri Matiyasevich.)



COMPUTABILITY A function that can be computed by a Turing machine is called **computable** and a function that cannot be computed by a Turing machine is called **uncomputable**. It is fairly straightforward, using a countability argument, to show that there are number-theoretic functions that are not computable (see Exercise 39 in Section 2.5). However, it is not so easy to actually produce such a function. The **busy beaver function** defined in the preamble to Exercise 31 is an example of an uncomputable function. One way to show that the busy beaver function is not computable is to show that it grows faster than any computable function. (See Exercise 32.)

Note that every decision problem can be reformulated as the problem of computing a function, namely, the function that has the value 1 when the answer to the problem is “yes” and that has the value 0 when the answer to the problem is “no.” A decision problem is solvable if and only if the corresponding function constructed in this way is computable.

THE CLASSES P AND NP In Section 3.3 we informally defined the classes of problems called P and NP. We are now able to define these concepts precisely using the notions of deterministic and nondeterministic Turing machines.

We first elaborate on the difference between a deterministic Turing machine and a nondeterministic Turing machine. The Turing machines we have studied in this section have all been deterministic. In a deterministic Turing machine $T = (S, I, f, s_0)$, transition rules are defined by the partial function f from $S \times I$ to $S \times I \times \{R, L\}$. Consequently, when transition rules of the machine are represented as five-tuples of the form (s, x, s', x', d) , where s is the current state, x is the current tape symbol, s' is the next state, x' is the symbol that replaces x on the tape, and d is the direction the machine moves on the tape, no two transition rules begin with the same pair (s, x) .

In a nondeterministic Turing machine, allowed steps are defined using a relation consisting of five-tuples rather than using a partial function. The restriction that no two transition rules begin with the same pair (s, x) is eliminated; that is, there may be more than one transition rule beginning with each (state, tape symbol) pair. Consequently, in a nondeterministic Turing machine, there is a choice of transitions for some pairs of the current state and the tape symbol being read. At each step of the operation of a nondeterministic Turing machine, the machine picks one of the different choices of the transition rules that begin with the current state and tape symbol pair. This choice can be considered to be a “guess” of which step to use. Just as for deterministic Turing machines, a nondeterministic Turing machine halts when there is no transition rule in its definition that begins with the current state and tape symbol. Given a nondeterministic Turing machine T , we say that a string x is recognized by T if and only if there exists some sequence of transitions of T that ends in a final state when the machine starts in the initial position with x written on the tape. The nondeterministic Turing machine T recognizes the set A if x is recognized by T if and only if $x \in A$. The nondeterministic Turing machine T is said to solve a decision problem if it recognizes the set consisting of all input values for which the answer to the decision problem is yes.

Definition 5

A decision problem is in P, the *class of polynomial-time problems*, if it can be solved by a deterministic Turing machine in polynomial time in terms of the size of its input. That is, a decision problem is in P if there is a deterministic Turing machine T that solves the decision problem and a polynomial $p(n)$ such that for all integers n , T halts in a final state after no more than $p(n)$ transitions whenever the input to T is a string of length n . A decision problem is in NP, the *class of nondeterministic polynomial-time problems*, if it can be solved by a nondeterministic Turing machine in polynomial time in terms of the size of its input. That is, a decision problem is in NP if there is a nondeterministic Turing machine T that solves the problem and a polynomial $p(n)$ such that for all integers n , T halts for every choice of transitions after no more than $p(n)$ transitions whenever the input to T is a string of length n .

Links



©University Archives.
Department of Rare Books
and Special Collections.
Princeton University Library

ALONZO CHURCH (1903–1995) Alonzo Church was born in Washington, D.C. He studied at Göttingen under Hilbert and later in Amsterdam. He was a member of the faculty at Princeton University from 1927 until 1967, when he moved to UCLA. Church was one of the founding members of the Association for Symbolic Logic. He made many substantial contributions to the theory of computability, including his solution to the decision problem, his invention of the lambda-calculus, and his statement of what is now known as the Church–Turing thesis. Among Church’s students were Stephen Kleene and Alan Turing. He published articles past his 90th birthday.

Problems in P are called **tractable**, whereas problems not in P are called **intractable**. For a problem to be in P , a deterministic Turing machine must exist that can decide in polynomial time whether a particular statement of the class addressed by the decision problem is true. For example, determining whether an item is in a list of n elements is a tractable problem. (We will not provide details on how this fact can be shown; the basic ideas used in the analyses of algorithms earlier in the text can be adapted when Turing machines are employed.) For a problem to be in NP , it is necessary only that there be a nondeterministic Turing machine that, when given a true statement from the set of statements addressed by the problem, can verify its truth in polynomial time by making the correct guess at each step from the set of allowable steps corresponding to the current state and tape symbol. The problem of determining whether a given graph has a Hamilton circuit is an NP problem, because a nondeterministic Turing machine can easily verify that a simple circuit in a graph passes through each vertex exactly once. It can do this by making a series of correct guesses corresponding to successively adding edges to form the circuit. Because every deterministic Turing machine can also be considered to be a nondeterministic Turing machine where each (state, tape symbol) pair occurs in exactly one transition rule defining the machine, every problem in P is also in NP . In symbols, $P \subseteq NP$.

One of the most perplexing open questions in theoretical computer science is whether every problem in NP is also in P , that is, whether $P = NP$. As mentioned in Section 3.3, there is an important class of problems, the class of NP -complete problems, such that a problem is in this class if it is in the class NP and if it can be shown that if it is also in the class P , then *every* problem in the class NP must also be in the class P . That is, a problem is NP -complete if the existence of a polynomial-time algorithm for solving it implies the existence of a polynomial-time algorithm for every problem in NP . In this book we have discussed several different NP -complete problems, such as determining whether a simple graph has a Hamilton circuit and determining whether a proposition in n -variables is a tautology.

Exercises

1. Let T be the Turing machine defined by the five-tuples: $(s_0, 0, s_1, 1, R)$, $(s_0, 1, s_1, 0, R)$, $(s_0, B, s_1, 0, R)$, $(s_1, 0, s_2, 1, L)$, $(s_1, 1, s_1, 0, R)$, and $(s_1, B, s_2, 0, L)$. For each of these initial tapes, determine the final tape when T halts, assuming that T begins in initial position.

a)	...	B	B	0	0	1	1	B	B	...
b)	...	B	B	1	0	1	B	B	B	...
c)	...	B	B	1	1	B	0	1	B	...
d)	...	B	B	B	B	B	B	B	B	...

2. Let T be the Turing machine defined by the five-tuples: $(s_0, 0, s_1, 0, R)$, $(s_0, 1, s_1, 0, L)$, $(s_0, B, s_1, 1, R)$, $(s_1, 0, s_2, 1, R)$, $(s_1, 1, s_1, 1, R)$, $(s_1, B, s_2, 0, R)$, and $(s_2, B, s_3, 0, R)$. For each of these initial tapes, determine the final tape when T halts, assuming that T begins in initial position.

a)	...	B	B	0	1	0	1	B	B	...
b)	...	B	B	1	1	1	B	B	B	...

c)	...	B	B	0	0	B	0	0	B	...
d)	...	B	B	B	B	B	B	B	B	...

3. What does the Turing machine described by the five-tuples $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 0, R)$, (s_0, B, s_2, B, R) , $(s_1, 0, s_1, 0, R)$, $(s_1, 1, s_0, 1, R)$, and (s_1, B, s_2, B, R) do when given
- 11 as input?
 - an arbitrary bit string as input?
4. What does the Turing machine described by the five-tuples $(s_0, 0, s_0, 1, R)$, $(s_0, 1, s_0, 1, R)$, (s_0, B, s_1, B, L) , $(s_1, 1, s_2, 1, R)$, do when given
- 101 as input?
 - an arbitrary bit string as input?
5. What does the Turing machine described by the five-tuples $(s_0, 1, s_1, 0, R)$, $(s_1, 1, s_1, 1, R)$, $(s_1, 0, s_2, 0, R)$, $(s_2, 0, s_3, 1, L)$, $(s_2, 1, s_2, 1, R)$, $(s_3, 1, s_3, 1, L)$, $(s_3, 0, s_4, 0, L)$, $(s_4, 1, s_4, 1, L)$, and $(s_4, 0, s_0, 1, R)$ do when given
- 11 as input?
 - a bit string consisting entirely of 1s as input?
6. Construct a Turing machine with tape symbols 0, 1, and B that, when given a bit string as input, adds a 1 to

the end of the bit string and does not change any of the other symbols on the tape.

7. Construct a Turing machine with tape symbols 0, 1, and B that, given a bit string as input, replaces the first 0 with a 1 and does not change any of the other symbols on the tape.
8. Construct a Turing machine with tape symbols 0, 1, and B that, given a bit string as input, replaces all 0s on the tape with 1s and does not change any of the 1s on the tape.
9. Construct a Turing machine with tape symbols 0, 1, and B that, given a bit string as input, replaces all but the leftmost 1 on the tape with 0s and does not change any of the other symbols on the tape.
10. Construct a Turing machine with tape symbols 0, 1, and B that, given a bit string as input, replaces the first two consecutive 1s on the tape with 0s and does not change any of the other symbols on the tape.
11. Construct a Turing machine that recognizes the set of all bit strings that end with a 0.
12. Construct a Turing machine that recognizes the set of all bit strings that contain at least two 1s.
13. Construct a Turing machine that recognizes the set of all bit strings that contain an even number of 1s.
14. Show at each step the contents of the tape of the Turing machine in Example 3 starting with each of these strings.
a) 0011 b) 00011 c) 101100 d) 000111
15. Explain why the Turing machine in Example 3 recognizes a bit string if and only if this string is of the form $0^n 1^n$ for some positive integer n .
- *16. Construct a Turing machine that recognizes the set $\{0^{2n} 1^n \mid n \geq 0\}$.
- *17. Construct a Turing machine that recognizes the set $\{0^n 1^n 2^n \mid n \geq 0\}$.
18. Construct a Turing machine that computes the function $f(n) = n + 2$ for all nonnegative integers n .
19. Construct a Turing machine that computes the function $f(n) = n - 3$ if $n \geq 3$ and $f(n) = 0$ for $n = 0, 1, 2$ for all nonnegative integers n .
20. Construct a Turing machine that computes the function $f(n) = n \bmod 3$ for every nonnegative integer n .
21. Construct a Turing machine that computes the function $f(n) = 3$ if $n \geq 5$ and $f(n) = 0$ if $n = 0, 1, 2, 3$, or 4.
22. Construct a Turing machine that computes the function $f(n) = 2n$ for all nonnegative integers n .
23. Construct a Turing machine that computes the function $f(n) = 3n$ for all nonnegative integers n .
24. Construct a Turing machine that computes the function $f(n_1, n_2) = n_2 + 2$ for all pairs of nonnegative integers n_1 and n_2 .
- *25. Construct a Turing machine that computes the function $f(n_1, n_2) = \min(n_1, n_2)$ for all nonnegative integers n_1 and n_2 .

26. Construct a Turing machine that computes the function $f(n_1, n_2) = n_1 + n_2 + 1$ for all nonnegative integers n_1 and n_2 .

Suppose that T_1 and T_2 are Turing machines with disjoint sets of states S_1 and S_2 and with transition functions f_1 and f_2 , respectively. We can define the Turing machine $T_1 T_2$, the **composite** of T_1 and T_2 , as follows. The set of states of $T_1 T_2$ is $S_1 \cup S_2$. $T_1 T_2$ begins in the start state of T_1 . It first executes the transitions of T_1 using f_1 up to, but not including, the step at which T_1 would halt. Then, for all moves for which T_1 halts, it executes the same transitions of T_1 except that it moves to the start state of T_2 . From this point on, the moves of $T_1 T_2$ are the same as the moves of T_2 .

27. By finding the composite of the Turing machines you constructed in Exercises 18 and 22, construct a Turing machine that computes the function $f(n) = 2n + 2$.
28. By finding the composite of the Turing machines you constructed in Exercises 18 and 23, construct a Turing machine that computes the function $f(n) = 3(n + 2) = 3n + 6$.
29. Which of the following problems is a decision problem?
 - a) What is the smallest prime greater than n ?
 - b) Is a graph G bipartite?
 - c) Given a set of strings, is there a finite-state automaton that recognizes this set of strings?
 - d) Given a checkerboard and a particular type of polyomino (see Section 1.8), can this checkerboard be tiled using polyominoes of this type?
30. Which of the following problems is a decision problem?
 - a) Is the sequence a_1, a_2, \dots, a_n of positive integers in increasing order?
 - b) Can the vertices of a simple graph G be colored using three colors so that no two adjacent vertices are the same color?
 - c) What is the vertex of highest degree in a graph G ?
 - d) Given two finite-state machines, do these machines recognize the same language?

Links

Let $B(n)$ be the maximum number of 1s that a Turing machine with n states with the alphabet $\{1, B\}$ may print on a tape that is initially blank. The problem of determining $B(n)$ for particular values of n is known as the **busy beaver problem**. This problem was first studied by Tibor Rado in 1962. Currently it is known that $B(2) = 4$, $B(3) = 6$, and $B(4) = 13$, but $B(n)$ is not known for $n \geq 5$. $B(n)$ grows rapidly; it is known that $B(5) \geq 4098$ and $B(6) \geq 3.5 \times 10^{18267}$.

- *31. Show that $B(2)$ is at least 4 by finding a Turing machine with two states and alphabet $\{1, B\}$ that halts with four consecutive 1s on the tape.
- **32. Show that the function $B(n)$ cannot be computed by any Turing machine. [Hint: Assume that there is a Turing machine that computes $B(n)$ in binary. Build a Turing machine T that, starting with a blank tape, writes n down in binary, computes $B(n)$ in binary, and converts $B(n)$ from binary to unary. Show that for sufficiently large n , the number of states of T is less than $B(n)$, leading to a contradiction.]

Key Terms and Results

TERMS

alphabet (or vocabulary): a set that contains elements used to form strings

language: a subset of the set of all strings over an alphabet

phrase-structure grammar (V, T, S, P) : a description of a language containing an alphabet V , a set of terminal symbols T , a start symbol S , and a set of productions P

the production $w \rightarrow w_1$: w can be replaced by w_1 whenever it occurs in a string in the language

$w_1 \Rightarrow w_2$ (w_2 is directly derivable from w_1): w_2 can be obtained from w_1 using a production to replace a string in w_1 with another string

$w_1 \stackrel{*}{\Rightarrow} w_2$ (w_2 is derivable from w_1): w_2 can be obtained from w_1 using a sequence of productions to replace strings by other strings

type 0 grammar: any phrase-structure grammar

type 1 grammar: a phrase-structure grammar in which every production is of the form $w_1 \rightarrow w_2$, where $w_1 = lAr$ and $w_2 = lwr$, where $A \in N$, $l, r, w \in (N \cup T)^*$ and $w \neq \lambda$, or $w_1 = S$ and $w_2 = \lambda$ as long as S is not on the right-hand side of another production

type 2, or context-free, grammar: a phrase-structure grammar in which every production is of the form $A \rightarrow w_1$, where A is a nonterminal symbol

type 3, or regular, grammar: a phrase-structure grammar where every production is of the form $A \rightarrow aB$, $A \rightarrow a$, or $S \rightarrow \lambda$, where A and B are nonterminal symbols, S is the start symbol, and a is a terminal symbol

derivation (or parse) tree: an ordered rooted tree where the root represents the starting symbol of a type 2 grammar, internal vertices represent nonterminals, leaves represent terminals, and the children of a vertex are the symbols on the right side of a production, in order from left to right, where the symbol represented by the parent is on the left-hand side

Backus–Naur form: a description of a context-free grammar in which all productions having the same nonterminal as their left-hand side are combined with the different right-hand sides of these productions, each separated by a bar, with nonterminal symbols enclosed in angular brackets and the symbol \rightarrow replaced by $::=$

finite-state machine (S, I, O, f, g, s_0) (or a Mealy machine): a six-tuple containing a set S of states, an input alphabet I , an output alphabet O , a transition function f that assigns a next state to every pair of a state and an input, an output function g that assigns an output to every pair of a state and an input, and a starting state s_0

AB (concatenation of A and B): the set of all strings formed by concatenating a string in A and a string in B in that order

A^* (Kleene closure of A): the set of all strings made up by concatenating arbitrarily many strings from A

deterministic finite-state automaton (S, I, f, s_0, F) : a five-tuple containing a set S of states, an input alphabet I , a transition function f that assigns a next state to every pair of a state and an input, a starting state s_0 , and a set of final states F

nondeterministic finite-state automaton (S, I, f, s_0, F) : a five-tuple containing a set S of states, an input alphabet I , a transition function f that assigns a set of possible next states to every pair of a state and an input, a starting state s_0 , and a set of final states F

language recognized by an automaton: the set of input strings that take the start state to a final state of the automaton

regular expression: an expression defined recursively by specifying that \emptyset , λ , and x , for all x in the input alphabet, are regular expressions, and that (AB) , $(A \cup B)$, and A^* are regular expressions when A and B are regular expressions

regular set: a set defined by a regular expression (see page 820)

Turing machine $T = (S, I, f, s_0)$: a four-tuple consisting of a finite set S of states, an alphabet I containing the blank symbol B , a partial function f from $S \times I$ to $S \times I \times \{R, L\}$, and a starting state s_0

nondeterministic Turing machine: a Turing machine that may have more than one transition rule corresponding to each (state, tape symbol) pair

decision problem: a problem that asks whether statements from a particular class of statements are true

solvable problem: a problem with the property that there is an effective algorithm that can solve all instances of the problem

unsolvable problem: a problem with the property that no effective algorithm exists that can solve all instances of the problem

computable function: a function whose values can be computed using a Turing machine

uncomputable function: a function whose values cannot be computed using a Turing machine

P, the class of polynomial-time problems: the class of problems that can be solved by a deterministic Turing machine in polynomial time in terms of the size of the input

NP, the class of nondeterministic polynomial-time problems: the class of problems that can be solved by a nondeterministic Turing machine in polynomial time in terms of the size of the input

NP-complete: a subset of the class of NP problems with the property that if any one of them is in the class P, then all problems in NP are in the class P

RESULTS

For every nondeterministic finite-state automaton there is a deterministic finite-state automaton that recognizes the same set.

Review Questions

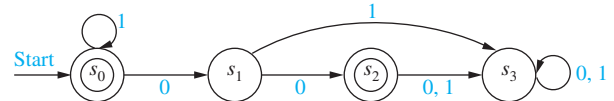
- Define a phrase-structure grammar.
 - What does it mean for a string to be derivable from a string w by a phrase-structure grammar G ?
- What is the language generated by a phrase-structure grammar G ?
 - What is the language generated by the grammar G with vocabulary $\{S, 0, 1\}$, set of terminals $T = \{0, 1\}$, starting symbol S , and productions $S \rightarrow 000S, S \rightarrow 1$?
 - Give a phrase-structure grammar that generates the set $\{01^n \mid n = 0, 1, 2, \dots\}$.
- Define a type 1 grammar.
 - Give an example of a grammar that is not a type 1 grammar.
 - Define a type 2 grammar.
 - Give an example of a grammar that is not a type 2 grammar but is a type 1 grammar.
 - Define a type 3 grammar.
 - Give an example of a grammar that is not a type 3 grammar but is a type 2 grammar.
- Define a regular grammar.
 - Define a regular language.
 - Show that the set $\{0^m 1^n \mid m, n = 0, 1, 2, \dots\}$ is a regular language.
- What is Backus–Naur form?
 - Give an example of the Backus–Naur form of the grammar for a subset of English of your choice.
- What is a finite-state machine?
 - Show how a vending machine that accepts only quarters and dispenses a soft drink after 75 cents has been deposited can be modeled using a finite-state machine.

Kleene's theorem: A set is regular if and only if there is a finite-state automaton that recognizes it.

A set is regular if and only if it is generated by a regular grammar.

The halting problem is unsolvable.

- Find the set of strings recognized by the deterministic finite-state automaton shown here.



- Construct a deterministic finite-state automaton that recognizes the set of bit strings that start with 1 and end with 1.
- What is the Kleene closure of a set of strings?
 - Find the Kleene closure of the set $\{11, 0\}$.
- Define a finite-state automaton.
 - What does it mean for a string to be recognized by a finite-state automaton?
- Define a nondeterministic finite-state automaton.
 - Show that given a nondeterministic finite-state automaton, there is a deterministic finite-state automaton that recognizes the same language.
- Define the set of regular expressions over a set I .
 - Explain how regular expressions are used to represent regular sets.
- State Kleene's theorem.
- Show that a set is generated by a regular grammar if and only if it is a regular set.
- Give an example of a set not recognized by a finite-state automaton. Show that no finite-state automaton recognizes it.
- Define a Turing machine.
- Describe how Turing machines are used to recognize sets.
- Describe how Turing machines are used to compute number-theoretic functions.
- What is an unsolvable decision problem? Give an example of such a problem.

Supplementary Exercises

- Find a phrase-structure grammar that generates each of these languages.
 - the set of bit strings of the form $0^{2n}1^{3n}$, where n is a nonnegative integer
 - the set of bit strings with twice as many 0s as 1s
 - the set of bit strings of the form w^2 , where w is a bit string
- Find a phrase-structure grammar that generates the set $\{0^{2^n} \mid n \geq 0\}$.

For Exercises 3 and 4, let $G = (V, T, S, P)$ be the context-free grammar with $V = \{(,), S, A, B\}$, $T = \{(,)\}$, starting symbol S , and productions $S \rightarrow A, A \rightarrow AB, A \rightarrow B, B \rightarrow (A)$, and $B \rightarrow (), S \rightarrow \lambda$.

- Construct the derivation trees of these strings.
 - $(())$
 - $((()))$
 - $(((())))$
- Show that $L(G)$ is the set of all balanced strings of parentheses, defined in the preamble to Supplementary Exercise 59 in Chapter 5.

A context-free grammar is **ambiguous** if there is a word in $L(G)$ with two derivations that produce different derivation trees, considered as ordered, rooted trees.

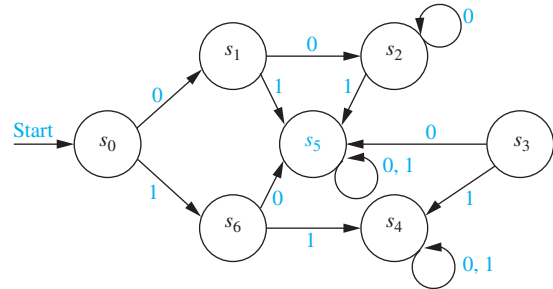
5. Show that the grammar $G = (V, T, S, P)$ with $V = \{0, S\}$, $T = \{0\}$, starting state S , and productions $S \rightarrow 0S$, $S \rightarrow S0$, and $S \rightarrow 0$ is ambiguous by constructing two different derivation trees for 0^3 .
6. Show that the grammar $G = (V, T, S, P)$ with $V = \{0, S\}$, $T = \{0\}$, starting state S , and productions $S \rightarrow 0S$ and $S \rightarrow 0$ is unambiguous.
7. Suppose that A and B are finite subsets of V^* , where V is an alphabet. Is it necessarily true that $|AB| = |BA|$?
8. Prove or disprove each of these statements for subsets A , B , and C of V^* , where V is an alphabet.
 - a) $A(B \cup C) = AB \cup AC$
 - b) $A(B \cap C) = AB \cap AC$
 - c) $(AB)C = A(BC)$
 - d) $(A \cup B)^* = A^* \cup B^*$
9. Suppose that A and B are subsets of V^* , where V is an alphabet. Does it follow that $A \subseteq B$ if $A^* \subseteq B^*$?
10. What set of strings with symbols in the set $\{0, 1, 2\}$ is represented by the regular expression $(2^*)(0 \cup (12^*))^*$?

The **star height** $h(E)$ of a regular expression over the set I is defined recursively by

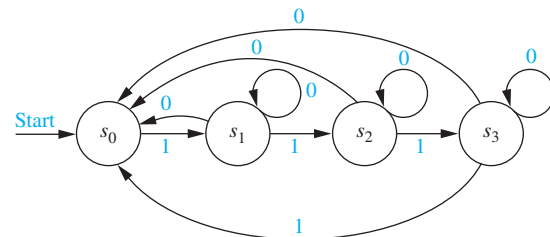
$$\begin{aligned} h(\emptyset) &= 0; \\ h(x) &= 0 \text{ if } x \in I; \\ h(E_1 \cup E_2) &= h(E_1 E_2) = \max(h(E_1), h(E_2)) \\ &\quad \text{if } E_1 \text{ and } E_2 \text{ are regular expressions;} \\ h(E^*) &= h(E) + 1 \text{ if } E \text{ is a regular expression.} \end{aligned}$$

11. Find the star height of each of these regular expressions.
 - a) 0^*1
 - b) 0^*1^*
 - c) $(0^*01)^*$
 - d) $((0^*1)^*)^*$
 - e) $(010^*)(1^*01^*)((01)^*(10)^*)^*$
 - f) $(((((0^*)1^*)0^*)1^*))^*)^*$
- *12. For each of these regular expressions find a regular expression that represents the same language with minimum star height.
 - a) $(0^*1^*)^*$
 - b) $(0(01^*0)^*)^*$
 - c) $(0^* \cup (01)^* \cup 1^*)^*$
13. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input contains four or more 1s. Then construct a deterministic finite-state automaton that recognizes this set.
14. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input contains four or more consecutive 1s. Then construct a deterministic finite-state automaton that recognizes this set.
15. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input ends with four or more consecutive 1s. Then construct a deterministic finite-state automaton that recognizes this set.

16. A state s' in a finite-state machine is said to be **reachable** from state s if there is an input string x such that $f(s, x) = s'$. A state s is called **transient** if there is no nonempty input string x with $f(s, x) = s$. A state s is called a **sink** if $f(s, x) = s$ for all input strings x . Answer these questions about the finite-state machine with the state diagram illustrated here.



- a) Which states are reachable from s_0 ?
- b) Which states are reachable from s_2 ?
- c) Which states are transient?
- d) Which states are sinks?
- *17. Suppose that S , I , and O are finite sets such that $|S| = n$, $|I| = k$, and $|O| = m$.
 - a) How many different finite-state machines (Mealy machines) $M = (S, I, O, f, g, s_0)$ can be constructed, where the starting state s_0 can be arbitrarily chosen?
 - b) How many different Moore machines $M = (S, I, O, f, g, s_0)$ can be constructed, where the starting state s_0 can be arbitrarily chosen?
- *18. Suppose that S and I are finite sets such that $|S| = n$ and $|I| = k$. How many different finite-state automata $M = (S, I, f, s_0, F)$ are there where the starting state s_0 and the subset F of S consisting of final states can be chosen arbitrarily?
 - a) if the automata are deterministic?
 - b) if the automata may be nondeterministic? (Note: This includes deterministic automata.)
19. Construct a deterministic finite-state automaton that is equivalent to the nondeterministic automaton with the state diagram shown here.



20. What is the language recognized by the automaton in Exercise 19?
21. Construct finite-state automata that recognize these sets.
 - a) $0^*(10)^*$
 - b) $(01 \cup 111)^*10^*(0 \cup 1)$
 - c) $(001 \cup (11)^*)^*$

- *22. Find regular expressions that represent the set of all strings of 0s and 1s
 - a) made up of blocks of even numbers of 1s interspersed with odd numbers of 0s.
 - b) with at least two consecutive 0s or three consecutive 1s.
 - c) with no three consecutive 0s or two consecutive 1s.
- *23. Show that if A is a regular set, then so is \bar{A} .
- *24. Show that if A and B are regular sets, then so is $A \cap B$.
- *25. Find finite-state automata that recognize these sets of strings of 0s and 1s.
 - a) the set of all strings that start with no more than three consecutive 0s and contain at least two consecutive 1s
 - b) the set of all strings with an even number of symbols that do not contain the pattern 101
 - c) the set of all strings with at least three blocks of two or more 1s and at least two 0s
- *26. Show that $\{0^{2^n} \mid n \in \mathbf{N}\}$ is not regular. You may use the pumping lemma given in Exercise 22 of Section 13.4.
- *27. Show that $\{1^p \mid p \text{ is prime}\}$ is not regular. You may use the pumping lemma given in Exercise 22 of Section 13.4.
- *28. There is a result for context-free languages analogous to the pumping lemma for regular sets. Suppose that $L(G)$ is the language recognized by a context-free language G . This result states that there is a constant N such that if z is a word in $L(G)$ with $l(z) \geq N$, then z can be written as $uvwxy$, where $l(vwx) \leq N$, $l(vx) \geq 1$, and uv^iwx^iy belongs to $L(G)$ for $i = 0, 1, 2, 3, \dots$. Use this result to show that there is no context-free grammar G with $L(G) = \{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$.
- *29. Construct a Turing machine that computes the function $f(n_1, n_2) = \max(n_1, n_2)$.
- *30. Construct a Turing machine that computes the function $f(n_1, n_2) = n_2 - n_1$ if $n_2 \geq n_1$ and $f(n_1, n_2) = 0$ if $n_2 < n_1$.

Computer Projects

Write programs with these input and output.

1. Given the productions in a phrase-structure grammar, determine which type of grammar this is in the Chomsky classification scheme.
2. Given the productions of a phrase-structure grammar, find all strings that are generated using twenty or fewer applications of its production rules.
3. Given the Backus–Naur form of a type 2 grammar, find all strings that are generated using twenty or fewer applications of the rules defining it.
- *4. Given the productions of a context-free grammar and a string, produce a derivation tree for this string if it is in the language generated by this grammar.
5. Given the state table of a Moore machine and an input string, produce the output string generated by the machine.
6. Given the state table of a Mealy machine and an input string, produce the output string generated by the machine.
7. Given the state table of a deterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.
8. Given the state table of a nondeterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.
- *9. Given the state table of a nondeterministic finite-state automaton, construct the state table of a deterministic finite-state automaton that recognizes the same language.
- **10. Given a regular expression, construct a nondeterministic finite-state automaton that recognizes the set that this expression represents.
11. Given a regular grammar, construct a finite-state automaton that recognizes the language generated by this grammar.
12. Given a finite-state automaton, construct a regular grammar that generates the language recognized by this automaton.
- *13. Given a Turing machine, find the output string produced by a given input string.

Computations and Explorations

Use a computational program or programs you have written to do these exercises.

1. Solve the busy beaver problem for two states by testing all possible Turing machines with two states and alphabet $\{1, B\}$.
- *2. Solve the busy beaver problem for three states by testing all possible Turing machines with three states and alphabet $\{1, B\}$.
- **3. Find a busy beaver machine with four states by testing all possible Turing machines with four states and alphabet $\{1, B\}$.
- **4. Make as much progress as you can toward finding a busy beaver machine with five states.
- **5. Make as much progress as you can toward finding a busy beaver machine with six states.

Writing Projects

Respond to these with essays using outside sources.

1. Describe how the growth of certain types of plants can be modeled using a Lindenmeyer system. Such a system uses a grammar with productions modeling the different ways plants can grow.
2. Describe the Backus–Naur form (and extended Backus–Naur form) rules used to specify the syntax of a programming language, such as Java, LISP, or Ada, or the database language SQL.
3. Explain how finite-state machines are used by spell-checkers.
4. Explain how finite-state machines are used in the study of network protocols.
5. Explain how finite-state machines are used in speech recognition programs.
6. Compare the use of Moore machines versus Mealy machines in the design of hardware systems and computer software.
7. Explain the concept of minimizing finite-state automata. Give an algorithm that carries out this minimization.
8. Give the definition of cellular automata. Explain their applications. Use the Game of Life as an example.
9. Define a pushdown automaton. Explain how pushdown automata are used to recognize sets. Which sets are recognized by pushdown automata? Provide an outline of a proof justifying your answer.
10. Define a linear-bounded automaton. Explain how linear-bounded automata are used to recognize sets. Which sets are recognized by linear-bounded automata? Provide an outline of a proof justifying your answer.
11. Look up Turing’s original definition of what we now call a Turing machine. What was his motivation for defining these machines?
12. Describe the concept of the universal Turing machine. Explain how such a machine can be built.
13. Explain the kinds of applications in which nondeterministic Turing machines are used instead of deterministic Turing machines.
14. Show that a Turing machine can simulate any action of a nondeterministic Turing machine.
15. Show that a set is recognized by a Turing machine if and only if it is generated by a phrase-structure grammar.
16. Describe the basic concepts of the lambda-calculus and explain how it is used to study computability of functions.
17. Show that a Turing machine as defined in this chapter can do anything a Turing machine with n tapes can do.
18. Show that a Turing machine with a tape infinite in one direction can do anything a Turing machine with a tape infinite in both directions can do.