second procedure we will describe, the Quine–McCluskey method, was invented in the 1960s. It automates the process of minimizing combinatorial circuits and can be implemented as a computer program.

COMPLEXITY OF BOOLEAN FUNCTION MINIMIZATION Unfortunately, minimizing Boolean functions with many variables is a computationally intensive problem. It has been shown that this problem is an NP-complete problem (see Section 3.3 and [Ka93]), so the existence of a polynomial-time algorithm for minimizing Boolean circuits is unlikely. The Quine–McCluskey method has exponential complexity. In practice, it can be used only when the number of literals does not exceed ten. Since the 1970s a number of newer algorithms have been developed for minimizing combinatorial circuits (see [Ha93] and [KaBe04]). However, with the best algorithms yet devised, only circuits with no more than 25 variables can be minimized. Also, heuristic (or rule-of-thumb) methods can be used to substantially simplify, but not necessarily minimize, Boolean expressions with a larger number of literals.

Karnaugh Maps

Links

To reduce the number of terms in a Boolean expression representing a circuit, it is necessary to find terms to combine. There is a graphical method, called a **Karnaugh map** or **K-map**, for finding terms to combine for Boolean functions involving a relatively small number of variables. The method we will describe was introduced by Maurice Karnaugh in 1953. His method is based on earlier work by E. W. Veitch. (This method is usually applied only when the function involves six or fewer variables.) K-maps give us a visual method for simplifying sum-of-products expansions; they are not suited for mechanizing this process. We will first illustrate how K-maps are used to simplify expansions of Boolean functions in two variables. We will continue by showing how K-maps can be used to minimize Boolean functions in three variables and then in four variables. Then we will describe the concepts that can be used to extend K-maps to minimize Boolean functions in more than four variables.

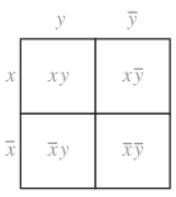


FIGURE 2 K-maps in Two Variables.

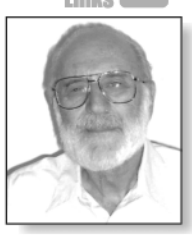
There are four possible minterms in the sum-of-products expansion of a Boolean function in the two variables x and y. A K-map for a Boolean function in these two variables consists of four cells, where a 1 is placed in the cell representing a minterm if this minterm is present in the expansion. Cells are said to be **adjacent** if the minterms that they represent differ in exactly one literal. For instance, the cell representing $\bar{x}y$ is adjacent to the cells representing xy and $\bar{x}\,\bar{y}$. The four cells and the terms that they represent are shown in Figure 2.

EXAMPLE 1 Find the K-maps for (a) $xy + \bar{x}y$, (b) $x\bar{y} + \bar{x}y$, and (c) $x\bar{y} + \bar{x}y + \bar{x}\,\bar{y}$.

Solution: We include a 1 in a cell when the minterm represented by this cell is present in the sum-of-products expansion. The three K-maps are shown in Figure 3. ◀

We can identify minterms that can be combined from the K-map. Whenever there are 1s in two adjacent cells in the K-map, the minterms represented by these cells can be combined into a product involving just one of the variables. For instance, $x\bar{y}$ and $\bar{x}\,\bar{y}$ are represented by adjacent cells and can be combined into $\bar{y}$, because $x\bar{y} + \bar{x}\,\bar{y} = (x + \bar{x})\bar{y} = \bar{y}$. Moreover, if 1s

Links



MAURICE KARNAUGH (BORN 1924) Maurice Karnaugh, born in New York City, received his B.S. from the City College of New York and his M.S. and Ph.D. from Yale University. He was a member of the technical staff at Bell Laboratories from 1952 until 1966 and Manager of Research and Development at the Federal Systems Division of AT&T from 1966 to 1970. In 1970 he joined IBM as a member of the research staff. Karnaugh has made fundamental contributions to the application of digital techniques in both computing and telecommunications. His current interests include knowledge-based systems in computers and heuristic search methods.

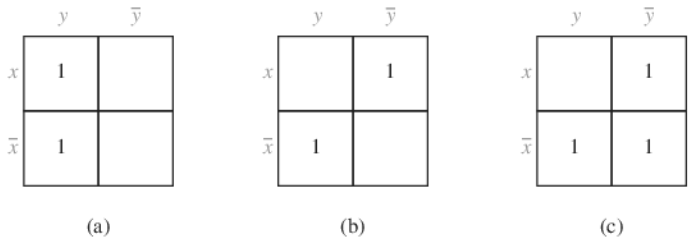


FIGURE 3 K-maps for the Sum-of-Products Expansions in Example 1.

are in all four cells, the four minterms can be combined into one term, namely, the Boolean expression 1 that involves none of the variables. We circle blocks of cells in the K-map that represent minterms that can be combined and then find the corresponding sum of products. The goal is to identify the largest possible blocks, and to cover all the 1s with the fewest blocks using the largest blocks first and always using the largest possible blocks.

EXAMPLE 2 Simplify the sum-of-products expansions given in Example 1.

Solution: The grouping of minterms is shown in Figure 4 using the K-maps for these expansions. Minimal expansions for these sums-of-products are (a) y, (b) $x\overline{y} + \overline{x}y$, and (c) $\overline{x} + \overline{y}$. ◀

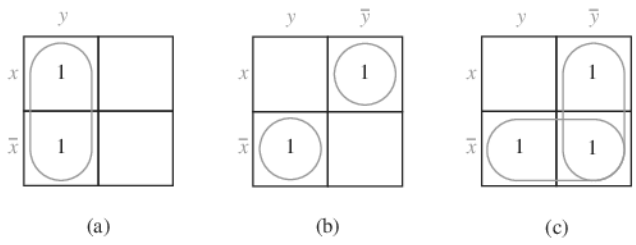


FIGURE 4 Simplifying the Sum-of-Products Expansions from Example 2.

A K-map in three variables is a rectangle divided into eight cells. The cells represent the eight possible minterms in three variables. Two cells are said to be adjacent if the minterms that they represent differ in exactly one literal. One of the ways to form a K-map in three variables is shown in Figure 5(a). This K-map can be thought of as lying on a cylinder, as shown in Figure 5(b). On the cylinder, two cells have a common border if and only if they are adjacent.

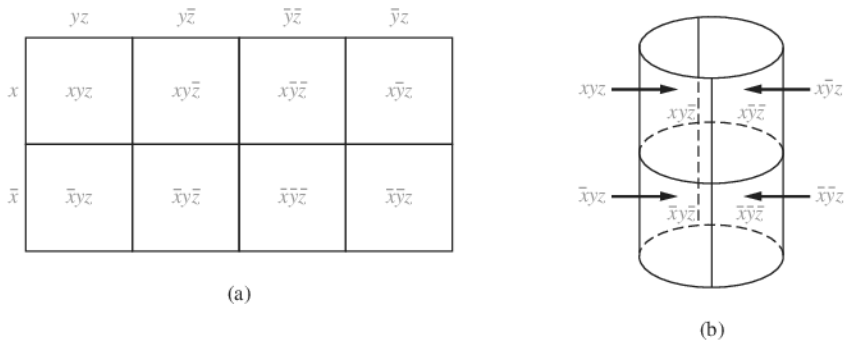


FIGURE 5 K-maps in Three Variables.

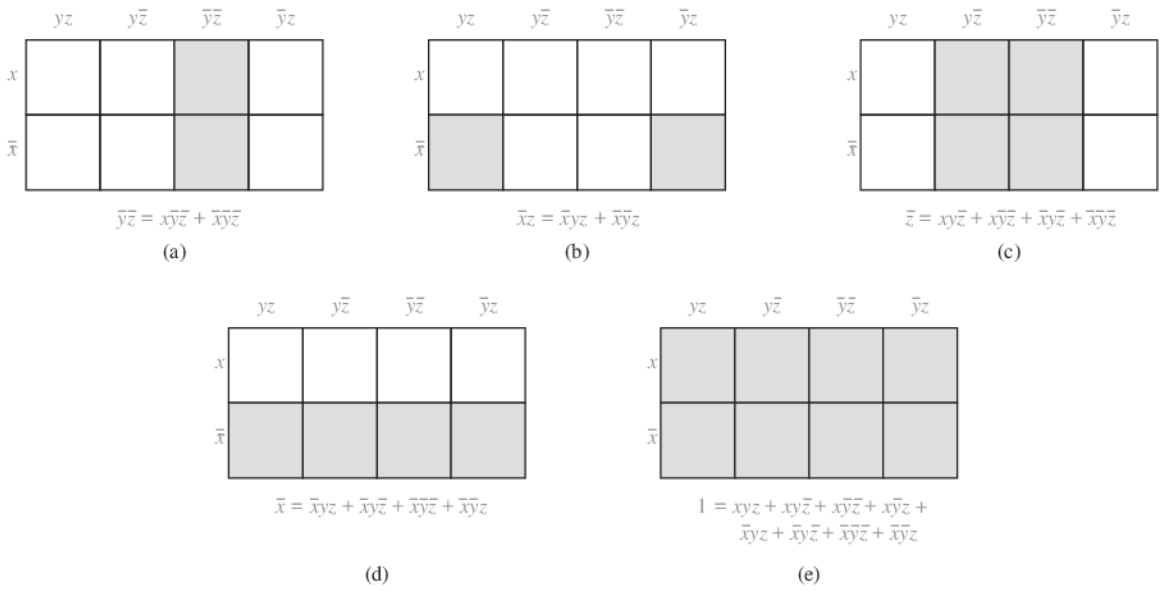


FIGURE 6 **Blocks in K-maps in Three Variables.**

To simplify a sum-of-products expansion in three variables, we use the K-map to identify blocks of minterms that can be combined. Blocks of two adjacent cells represent pairs of minterms that can be combined into a product of two literals; 2×2 and 4×1 blocks of cells represent minterms that can be combined into a single literal; and the block of all eight cells represents a product of no literals, namely, the function 1. In Figure 6, 1×2, 2×1, 2×2, 4×1, and 4×2 blocks and the products they represent are shown.

The product of literals corresponding to a block of all 1s in the K-map is called an **implicant** of the function being minimized. It is called a **prime implicant** if this block of 1s is not contained in a larger block of 1s representing the product of fewer literals than in this product.

The goal is to identify the largest possible blocks in the map and cover all the 1s in the map with the least number of blocks, using the largest blocks first. The largest possible blocks are always chosen, but we must always choose a block if it is the only block of 1s covering a 1 in the K-map. Such a block represents an **essential prime implicant**. By covering all the 1s in the map with blocks corresponding to prime implicants we can express the sum of products as a sum of prime implicants. Note that there may be more than one way to cover all the 1s using the least number of blocks.

Example 3 illustrates how K-maps in three variables are used.

EXAMPLE 3 Use K-maps to minimize these sum-of-products expansions.

(a) $xy\overline{z} + x\overline{y}\,\overline{z} + \overline{x}yz + \overline{x}\,\overline{y}\,\overline{z}$
(b) $x\overline{y}z + x\overline{y}\,\overline{z} + \overline{x}yz + \overline{x}\,\overline{y}z + \overline{x}\,\overline{y}\,\overline{z}$
(c) $xyz + xy\overline{z} + x\overline{y}z + x\overline{y}\,\overline{z} + \overline{x}yz + \overline{x}\,\overline{y}z + \overline{x}\,\overline{y}\,\overline{z}$
(d) $xy\overline{z} + x\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + \overline{x}\,\overline{y}\,\overline{z}$

Solution: The K-maps for these sum-of-products expansions are shown in Figure 7. The grouping of blocks shows that minimal expansions into Boolean sums of Boolean products are (a) $x\overline{z} + \overline{y}\,\overline{z} + \overline{x}yz$, (b) $\overline{y} + \overline{x}z$, (c) $x + \overline{y} + z$, and (d) $x\overline{z} + \overline{x}\,\overline{y}$. In part (d) note that the prime implicants $x\overline{z}$ and $\overline{x}\,\overline{y}$ are essential prime implicants, but the prime implicant $\overline{y}\,\overline{z}$ is a prime implicant that is not essential, because the cells it covers are covered by the other two prime implicants. ◀

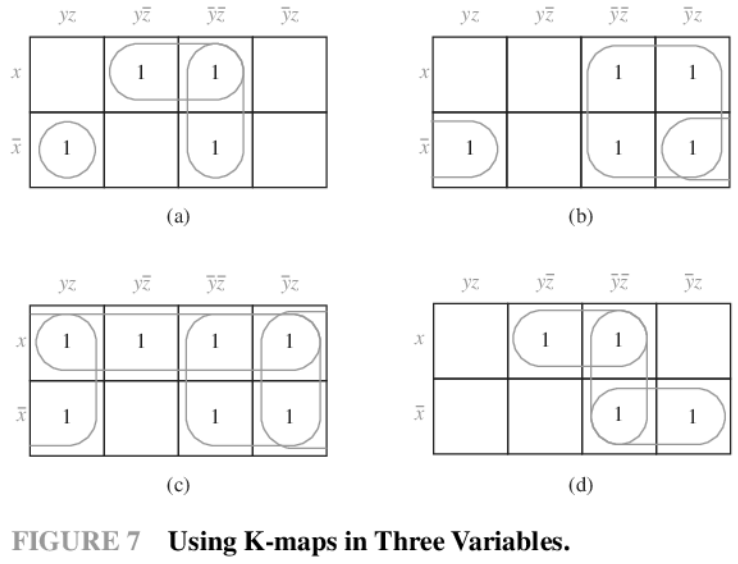


FIGURE 7 Using K-maps in Three Variables.

A K-map in four variables is a square that is divided into 16 cells. The cells represent the 16 possible minterms in four variables. One of the ways to form a K-map in four variables is shown in Figure 8.

Two cells are adjacent if and only if the minterms they represent differ in one literal. Consequently, each cell is adjacent to four other cells. The K-map of a sum-of-products expansion in four variables can be thought of as lying on a torus, so that adjacent cells have a common boundary (see Exercise 28). The simplification of a sum-of-products expansion in four variables is carried out by identifying those blocks of 2, 4, 8, or 16 cells that represent minterms that can be combined. Each cell representing a minterm must either be used to form a product using fewer literals, or be included in the expansion. In Figure 9 some examples of blocks that represent products of three literals, products of two literals, and a single literal are illustrated.

As is the case in K-maps in two and three variables, the goal is to identify the largest blocks of 1s in the map that correspond to the prime implicants and to cover all the 1s using the fewest blocks needed, using the largest blocks first. The largest possible blocks are always used. Example 4 illustrates how K-maps in four variables are used.

	yz	$yz̄$	$ȳz̄$	$ȳz$
wx	$wxyz$	$wxyz̄$	$wxȳz̄$	$wxȳz$
$wx̄$	$wx̄yz$	$wx̄yz̄$	$wx̄ȳz̄$	$wx̄ȳz$
$w̄x̄$	$w̄x̄yz$	$w̄x̄yz̄$	$w̄x̄ȳz̄$	$w̄x̄ȳz$
$w̄x$	$w̄xyz$	$w̄xyz̄$	$w̄xȳz̄$	$w̄xȳz$

FIGURE 8 K-maps in Four Variables.

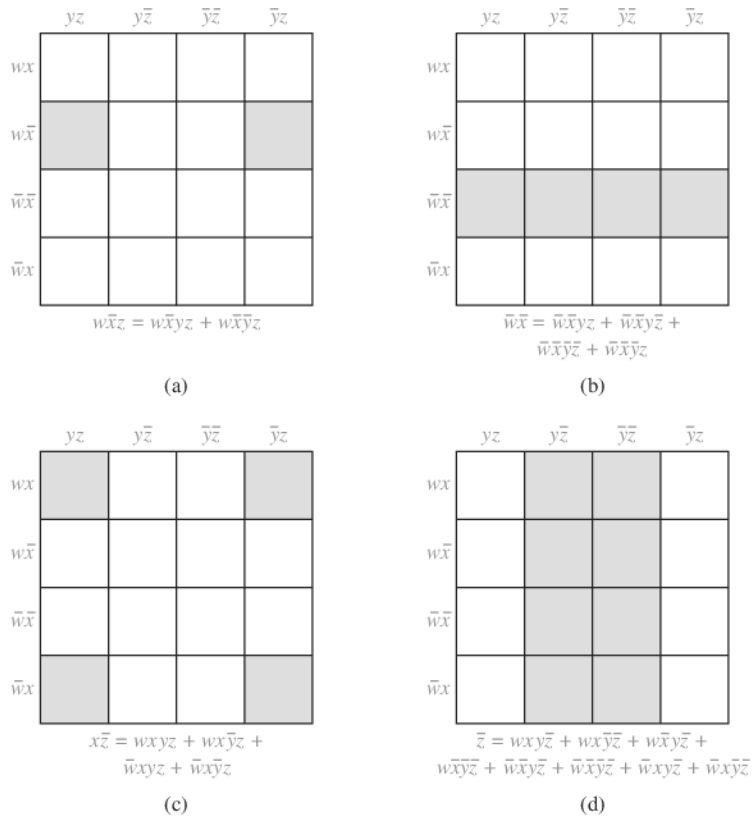


FIGURE 9 **Blocks in K-maps in Four Variables.**

EXAMPLE 4 Use K-maps to simplify these sum-of-products expansions.

(a) $wxyz + wxy\bar{z} + wx\bar{y}\,\bar{z} + w\bar{x}yz + w\bar{x}\,\bar{y}z + w\bar{x}\,\bar{y}\,\bar{z} + \bar{w}x\bar{y}z + \bar{w}\,\bar{x}yz + \bar{w}\,\bar{x}y\bar{z}$

(b) $wx\bar{y}\,\bar{z} + w\bar{x}yz + w\bar{x}y\bar{z} + w\bar{x}\,\bar{y}\,\bar{z} + \bar{w}x\bar{y}\,\bar{z} + \bar{w}\,\bar{x}y\bar{z} + \bar{w}\,\bar{x}\,\bar{y}\,\bar{z}$

(c) $wxy\bar{z} + wx\bar{y}\,\bar{z} + w\bar{x}yz + w\bar{x}y\bar{z} + w\bar{x}\,\bar{y}\,\bar{z} + \bar{w}xyz + \bar{w}xy\bar{z} + \bar{w}x\bar{y}\,\bar{z} + \bar{w}x\bar{y}z + \bar{w}\,\bar{x}y\bar{z} + \bar{w}\,\bar{x}\,\bar{y}\,\bar{z}$

Solution: The K-maps for these expansions are shown in Figure 10. Using the blocks shown leads to the sum of products (a) $wyz + wx\bar{z} + w\bar{x}\,\bar{y} + \bar{w}\bar{x}y + \bar{w}x\bar{y}z$, (b) $\bar{y}\,\bar{z} + w\bar{x}y + \bar{x}\,\bar{z}$, and (c) $\bar{z} + \bar{w}x + w\bar{x}y$. The reader should determine whether there are other choices of blocks in each part that lead to different sums of products representing these Boolean functions. ◀

K-maps can realistically be used to minimize Boolean functions with five or six variables, but beyond that, they are rarely used because they become extremely complicated. However, the concepts used in K-maps play an important role in newer algorithms. Furthermore, mastering these concepts helps you understand these newer algorithms and the computer-aided design (CAD) programs that implement them. As we develop these concepts, we will be able to illustrate them by referring back to our discussion of minimization of Boolean functions in three and in four variables.

The K-maps we used to minimize Boolean functions in two, three, and four variables are built using 2×2, 2×4, and 4×4 rectangles, respectively. Furthermore, corresponding cells in the top row and bottom row and in the leftmost column and rightmost column in each of these

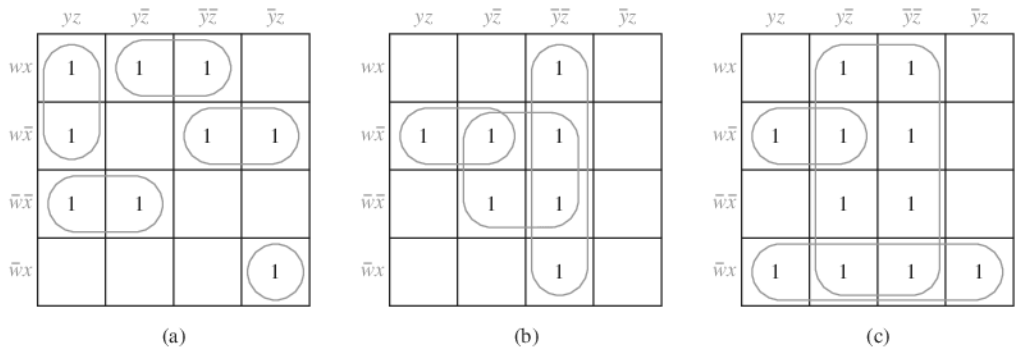


FIGURE 10 **Using K-maps in Four Variables.**

cases are considered adjacent because they represent minterms differing in only one literal. We can build K-maps for minimizing Boolean functions in more than four variables in a similar way. We use a rectangle containing $2^{\lfloor n/2 \rfloor}$ rows and $2^{\lceil n/2 \rceil}$ columns. (These K-maps contain 2^n cells because $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$.) The rows and columns need to be positioned so that the cells representing minterms differing in just one literal are adjacent or are considered adjacent by specifying additional adjacencies of rows and columns. To help (but not entirely) achieve this, the rows and columns of a K-map are arranged using Gray codes (see Section 10.5), where we associate bit strings and products by specifying that a 1 corresponds to the appearance of a variable and a 0 with the appearance of its complement. For example, in a 10-dimensional K-map, the Gray code 01110 used to label a row corresponds to the product $\overline{x}_1 x_2 x_3 x_4 \overline{x}_5$.

EXAMPLE 5 The K-maps we used to minimize Boolean functions with four variables have four rows and four columns. Both the rows and the columns are arranged using the Gray code 11,10,00,01. The rows represent products wx, $w\overline{x}$, $\overline{w}\,\overline{x}$, and $\overline{w}x$, respectively, and the columns correspond to the products yz, $y\overline{z}$, $\overline{y}\,\overline{z}$, and $\overline{y}z$, respectively. Using Gray codes and considering cells adjacent in the first and last rows and in the first and last columns, we ensured that minterms that differ in only one variable are always adjacent. ◀

EXAMPLE 6 To minimize Boolean functions in five variables we use K-maps with $2^3 = 8$ columns and $2^2 = 4$ rows. We label the four rows using the Gray code 11,10,00,01, corresponding to x_1x_2, $x_1\overline{x}_2$, $\overline{x}_1\overline{x}_2$, and $\overline{x}_1x_2$, respectively. We label the eight columns using the Gray code 111,110,100,101,001,000,010,011 corresponding to the terms $x_3x_4x_5$, $x_3x_4\overline{x}_5$, $x_3\overline{x}_4\overline{x}_5$, $x_3\overline{x}_4x_5$, $\overline{x}_3\overline{x}_4x_5$, $\overline{x}_3\overline{x}_4\overline{x}_5$, $\overline{x}_3x_4\overline{x}_5$, and $\overline{x}_3x_4x_5$, respectively. Using Gray codes to label columns and rows ensures that the minterms represented by adjacent cells differ in only one variable. However, to make sure all cells representing products that differ in only one variable are considered adjacent, we consider cells in the top and bottom rows to be adjacent, as well as cells in the first and eighth columns, the first and fourth columns, the second and seventh columns, the third and sixth columns, and the fifth and eighth columns (as the reader should verify). ◀

To use a K-map to minimize a Boolean function in n variables, we first draw a K-map of the appropriate size. We place 1s in all cells corresponding to minterms in the sum-of-products expansion of this function. We then identify all prime implicants of the Boolean function. To do this we look for the blocks consisting of 2^k clustered cells all containing a 1, where $1 \leq k \leq n$. These blocks correspond to the product of $n - k$ literals. (Exercise 33 asks the reader to verify this.) Furthermore, a block of 2^k cells each containing a 1 not contained in a block of 2^{k+1} cells each containing a 1 represents a prime implicant. The reason that this implicant is a prime implicant is that no product obtained by deleting a literal is also represented by a block of cells all containing 1s.

EXAMPLE 7 A block of eight cells representing a product of two literals in a K-map for minimizing Boolean functions in five variables all containing 1s is a prime implicant if it is not contained in a larger block of 16 cells all containing 1s representing a single literal. ◀

Once all prime implicants have been identified, the goal is to find the smallest possible subset of these prime implicants with the property that the cells representing these prime implicants cover all the cells containing a 1 in the K-map. We begin by selecting the essential prime implicants because each of these is represented by a block that covers a cell containing a 1 that is not covered by any other prime implicant. We add additional prime implicants to ensure that all 1s in the K-map are covered. When the number of variables is large, this last step can become exceedingly complicated.

Don't Care Conditions

Links

In some circuits we care only about the output for some combinations of input values, because other combinations of input values are not possible or never occur. This gives us freedom in producing a simple circuit with the desired output because the output values for all those combinations that never occur can be arbitrarily chosen. The values of the function for these combinations are called ***don't care* conditions**. A d is used in a K-map to mark those combinations of values of the variables for which the function can be arbitrarily assigned. In the minimization process we can assign 1s as values to those combinations of the input values that lead to the largest blocks in the K-map. This is illustrated in Example 8.

EXAMPLE 8 One way to code decimal expansions using bits is to use the four bits of the binary expansion of each digit in the decimal expansion. For instance, 873 is encoded as 100001110011. This encoding of a decimal expansion is called a **binary coded decimal expansion**. Because there are 16 blocks of four bits and only 10 decimal digits, there are six combinations of four bits that are not used to encode digits. Suppose that a circuit is to be built that produces an output of 1 if the decimal digit is 5 or greater and an output of 0 if the decimal digit is less than 5. How can this circuit be simply built using OR gates, AND gates, and inverters?

Solution: Let $F(w, x, y, z)$ denote the output of the circuit, where $wxyz$ is a binary expansion of a decimal digit. The values of F are shown in Table 1. The K-map for F, with ds in the *don't care* positions, is shown in Figure 11(a). We can either include or exclude squares with ds from blocks. This gives us many possible choices for the blocks. For example, excluding all squares with ds and forming blocks, as shown in Figure 11(b), produces the expression $w\overline{x}\,\overline{y} + \overline{w}xy + \overline{w}xz$. Including some of the ds and excluding others and forming blocks, as

TABLE 1

Digit	*w*	*x*	*y*	*z*	*F*
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1

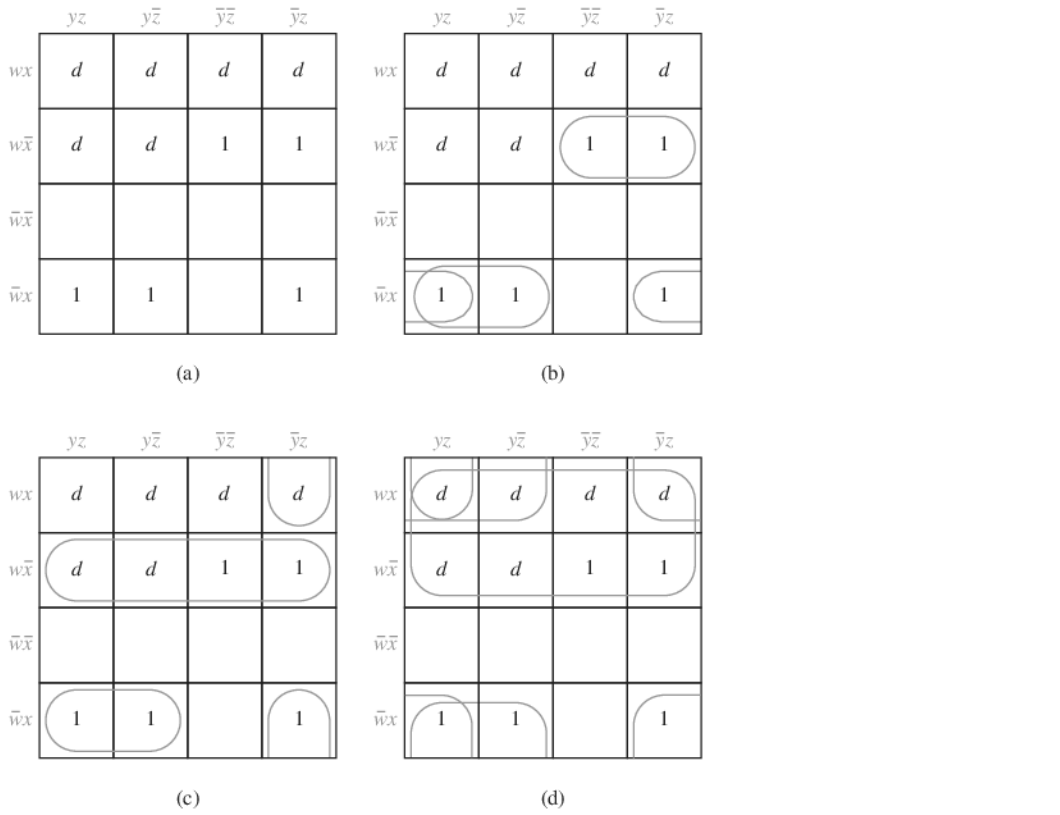


FIGURE 11 **The K-map for F Showing Its *Don't Care* Positions.**

shown in Figure 11(c), produces the expression $w\overline{x} + \overline{w}xy + x\overline{y}z$. Finally, including all the ds and using the blocks shown in Figure 11(d) produces the simplest sum-of-products expansion possible, namely, $F(x, y, z) = w + xy + xz$. ◀

The Quine–McCluskey Method

Links

We have seen that K-maps can be used to produce minimal expansions of Boolean functions as Boolean sums of Boolean products. However, K-maps are awkward to use when there are more than four variables. Furthermore, the use of K-maps relies on visual inspection to identify terms to group. For these reasons there is a need for a procedure for simplifying sum-of-products expansions that can be mechanized. The Quine–McCluskey method is such a procedure. It can be used for Boolean functions in any number of variables. It was developed in the 1950s by W. V. Quine and E. J. McCluskey, Jr. Basically, the Quine–McCluskey method consists of two

Links

EDWARD J. McCLUSKEY (BORN 1929) Edward McCluskey attended Bowdoin College and M.I.T., where he received his doctorate in electrical engineering in 1956. He joined Bell Telephone Laboratories in 1955, remaining there until 1959. McCluskey was professor of electrical engineering at Princeton University from 1959 until 1966, also serving as director of the Computer Center at Princeton from 1961 to 1966. In 1967 he took a position as professor of computer science and electrical engineering at Stanford University, where he also served as director of the Digital Systems Laboratory from 1969 to 1978. McCluskey has worked in a variety of areas in computer science, including fault-tolerant computing, computer architecture, testing, and logic design. He is the director of the Center for Reliable Computing at Stanford University where he is now an emeritus professor. McCluskey is also an ACM Fellow.

TABLE 2

Minterm	*Bit String*	*Number of 1s*
xyz	111	3
$x\overline{y}z$	101	2
$\overline{x}yz$	011	2
$\overline{x}\,\overline{y}z$	001	1
$\overline{x}\,\overline{y}\,\overline{z}$	000	0

parts. The first part finds those terms that are candidates for inclusion in a minimal expansion as a Boolean sum of Boolean products. The second part determines which of these terms to actually use. We will use Example 9 to illustrate how, by successively combining implicants into implicants with one fewer literal, this procedure works.

EXAMPLE 9 We will show how the Quine–McCluskey method can be used to find a minimal expansion equivalent to

$$xyz + x\overline{y}z + \overline{x}yz + \overline{x}\,\overline{y}z + \overline{x}\,\overline{y}\,\overline{z}.$$

We will represent the minterms in this expansion by bit strings. The first bit will be 1 if x occurs and 0 if $\overline{x}$ occurs. The second bit will be 1 if y occurs and 0 if $\overline{y}$ occurs. The third bit will be 1 if z occurs and 0 if $\overline{z}$ occurs. We then group these terms according to the number of 1s in the corresponding bit strings. This information is shown in Table 2.

Minterms that can be combined are those that differ in exactly one literal. Hence, two terms that can be combined differ by exactly one in the number of 1s in the bit strings that represent them. When two minterms are combined into a product, this product contains two literals. A product in two literals is represented using a dash to denote the variable that does not occur. For instance, the minterms $x\overline{y}z$ and $\overline{x}\,\overline{y}z$, represented by bit strings 101 and 001, can be combined into $\overline{y}z$, represented by the string –01. All pairs of minterms that can be combined and the product formed from these combinations are shown in Table 3.

Next, all pairs of products of two literals that can be combined are combined into one literal. Two such products can be combined if they contain literals for the same two variables, and literals for only one of the two variables differ. In terms of the strings representing the products, these strings must have a dash in the same position and must differ in exactly one of the other two slots. We can combine the products yz and $\overline{y}z$, represented by the strings –11 and –01, into z, represented by the string – –1. We show all the combinations of terms that can be formed in this way in Table 3.

TABLE 3

			Step 1			*Step 2*		
	Term	*Bit String*		*Term*	*String*		*Term*	*String*
1	xyz	111	(1,2)	xz	1–1	(1,2,3,4)	z	– –1
2	$x\overline{y}z$	101	(1,3)	yz	–11			
3	$\overline{x}yz$	011	(2,4)	$\overline{y}z$	–01			
4	$\overline{x}\,\overline{y}z$	001	(3,4)	$\overline{x}z$	0–1			
5	$\overline{x}\,\overline{y}\,\overline{z}$	000	(4,5)	$\overline{x}\,\overline{y}$	00–			

TABLE 4					
	xyz	$x\overline{y}z$	$\overline{x}yz$	$\overline{x}\,\overline{y}z$	$\overline{x}\,\overline{y}\,\overline{z}$
z	X	X	X	X	
$\overline{x}\,\overline{y}$				X	X

In Table 3 we also indicate which terms have been used to form products with fewer literals; these terms will not be needed in a minimal expansion. The next step is to identify a minimal set of products needed to represent the Boolean function. We begin with all those products that were not used to construct products with fewer literals. Next, we form Table 4, which has a row for each candidate product formed by combining original terms, and a column for each original term; and we put an X in a position if the original term in the sum-of-products expansion was used to form this candidate product. In this case, we say that the candidate product **covers** the original minterm. We need to include at least one product that covers each of the original minterms. Consequently, whenever there is only one X in a column in the table, the product corresponding to the row this X is in must be used. From Table 4 we see that both z and $\overline{x}\,\overline{y}$ are needed. Hence, the final answer is $z + \overline{x}\,\overline{y}$. ◀

As was illustrated in Example 9, the Quine–McCluskey method uses this sequence of steps to simplify a sum-of-products expression.

1. Express each minterm in n variables by a bit string of length n with a 1 in the ith position if x_i occurs and a 0 in this position if $\overline{x}_i$ occurs.

2. Group the bit strings according to the number of 1s in them.

3. Determine all products in $n - 1$ variables that can be formed by taking the Boolean sum of minterms in the expansion. Minterms that can be combined are represented by bit strings that differ in exactly one position. Represent these products in $n - 1$ variables with strings that have a 1 in the ith position if x_i occurs in the product, a 0 in this position if $\overline{x}_i$ occurs, and a dash in this position if there is no literal involving x_i in the product.

Links

WILLARD VAN ORMAN QUINE (1908–2000) Willard Quine, born in Akron, Ohio, attended Oberlin College and later Harvard University, where he received his Ph.D. in philosophy in 1932. He became a Junior Fellow at Harvard in 1933 and was appointed to a position on the faculty there in 1936. He remained at Harvard his entire professional life, except for World War II, when he worked for the U.S. Navy decrypting messages from German submarines. Quine was always interested in algorithms, but not in hardware. He arrived at his discovery of what is now called the Quine–McCluskey method as a device for teaching mathematical logic, rather than as a method for simplifying switching circuits. Quine was one of the most famous philosophers of the twentieth century. He made fundamental contributions to the theory of knowledge, mathematical logic and set theory, and the philosophies of logic and language. His books, including *New Foundations of Mathematical Logic* published in 1937 and *Word and Object* published in 1960, have had a profound impact. Quine retired from Harvard in 1978 but continued to commute from his home in Beacon Hill to his office there. He used the 1927 Remington typewriter on which he prepared his doctoral thesis for his entire life. He even had an operation performed on this machine to add a few special symbols, removing the second period, the second comma, and the question mark. When asked whether he missed the question mark, he replied, "Well, you see, I deal in certainties." There is even a word *quine*, defined in the *New Hacker's Dictionary* as a program that generates a copy of its own source code as its complete output. Producing the shortest possible quine in a given programming language is a popular puzzle for hackers.

4. Determine all products in $n-2$ variables that can be formed by taking the Boolean sum of the products in $n-1$ variables found in the previous step. Products in $n-1$ variables that can be combined are represented by bit strings that have a dash in the same position and differ in exactly one position.
5. Continue combining Boolean products into products in fewer variables as long as possible.
6. Find all the Boolean products that arose that were not used to form a Boolean product in one fewer literal.
7. Find the smallest set of these Boolean products such that the sum of these products represents the Boolean function. This is done by forming a table showing which minterms are covered by which products. Every minterm must be covered by at least one product. The first step in using this table is to find all essential prime implicants. Each essential prime implicant must be included because it is the only prime implicant that covers one of the minterms. Once we have found essential prime implicants, we can simplify the table by eliminating the rows for minterms covered by these prime implicants. Furthermore, we can eliminate any prime implicants that cover a subset of minterms covered by another prime implicant (as the reader should verify). Moreover, we can eliminate from the table the row for a minterm if there is another minterm that is covered by a subset of the prime implicants that cover this minterm. This process of identifying essential prime implicants that must be included, followed by eliminating redundant prime implicants and identifying minterms that can be ignored, is iterated until the table does not change. At this point we use a backtracking procedure to find the optimal solution where we add prime implicants to the cover to find possible solutions, which we compare to the best solution found so far at each step.

A final example will illustrate how this procedure is used to simplify a sum-of-products expansion in four variables.

EXAMPLE 10 Use the Quine–McCluskey method to simplify the sum-of-products expansion $wxy\overline{z} + w\overline{x}yz + w\overline{x}y\overline{z} + \overline{w}xyz + \overline{w}x\overline{y}z + \overline{w}\,\overline{x}yz + \overline{w}\,\overline{x}\,\overline{y}z$.

Solution: We first represent the minterms by bit strings and then group these terms together according to the number of 1s in the bit strings. This is shown in Table 5. All the Boolean products that can be formed by taking Boolean sums of these products are shown in Table 6.

The only products that were not used to form products in fewer variables are $\overline{w}z$, $wy\overline{z}$, $w\overline{x}y$, and $\overline{x}yz$. In Table 7 we show the minterms covered by each of these products. To cover these minterms we must include $\overline{w}z$ and $wy\overline{z}$, because these products are the only products that cover $\overline{w}xyz$ and $wxy\overline{z}$, respectively. Once these two products are included, we see that only one of the two products left is needed. Consequently, we can take either $\overline{w}z + wy\overline{z} + w\overline{x}y$ or $\overline{w}z + wy\overline{z} + \overline{x}yz$ as the final answer. ◀

TABLE 5

Term	*Bit String*	*Number of 1s*
$wxy\overline{z}$	1110	3
$w\overline{x}yz$	1011	3
$\overline{w}xyz$	0111	3
$w\overline{x}y\overline{z}$	1010	2
$\overline{w}x\overline{y}z$	0101	2
$\overline{w}\,\overline{x}yz$	0011	2
$\overline{w}\,\overline{x}\,\overline{y}z$	0001	1

TABLE 6

			Step 1			*Step 2*		
	Term	***Bit String***		***Term***	***String***		***Term***	***String***
1	$wxy\overline{z}$	1110	(1,4)	$wy\overline{z}$	1–10	(3,5,6,7)	$\overline{w}z$	0 – –1
2	$w\overline{x}yz$	1011	(2,4)	$w\overline{x}y$	101–			
3	$\overline{w}xyz$	0111	(2,6)	$\overline{x}yz$	–011			
4	$w\overline{x}y\overline{z}$	1010	(3,5)	$\overline{w}xz$	01–1			
5	$\overline{w}x\overline{y}z$	0101	(3,6)	$\overline{w}yz$	0–11			
6	$\overline{w}\,\overline{x}yz$	0011	(5,7)	$\overline{w}\,\overline{y}z$	0–01			
7	$\overline{w}\,\overline{x}\,\overline{y}z$	0001	(6,7)	$\overline{w}\,\overline{x}z$	00–1			

TABLE 7

	$wxy\overline{z}$	$w\overline{x}yz$	$\overline{w}xyz$	$w\overline{x}y\overline{z}$	$\overline{w}x\overline{y}z$	$\overline{w}\,\overline{x}yz$	$\overline{w}\,\overline{x}\,\overline{y}z$
$\overline{w}z$			X		X	X	X
$wy\overline{z}$	X			X			
$w\overline{x}y$		X		X			
$\overline{x}yz$		X				X	

Exercises

1. **a)** Draw a K-map for a function in two variables and put a 1 in the cell representing $\overline{x}y$.

b) What are the minterms represented by cells adjacent to this cell?

2. Find the sum-of-products expansions represented by each of these K-maps.

a)

	y	$\overline{y}$
x	1	
$\overline{x}$	1	1

b)

	y	$\overline{y}$
x	1	1
$\overline{x}$		

c)

	y	$\overline{y}$
x	1	1
$\overline{x}$	1	1

3. Draw the K-maps of these sum-of-products expansions in two variables.

a) $x\overline{y}$ **b)** $xy + \overline{x}\,\overline{y}$

c) $xy + x\overline{y} + \overline{x}y + \overline{x}\,\overline{y}$

4. Use a K-map to find a minimal expansion as a Boolean sum of Boolean products of each of these functions of the Boolean variables x and y.

a) $\overline{x}y + \overline{x}\,\overline{y}$

b) $xy + x\overline{y}$

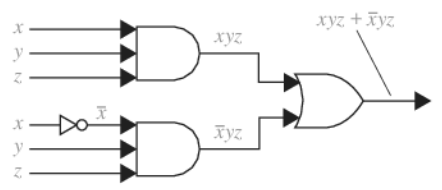
c) $xy + x\overline{y} + \overline{x}y + \overline{x}\,\overline{y}$

5. **a)** Draw a K-map for a function in three variables. Put a 1 in the cell that represents $\overline{x}y\overline{z}$.

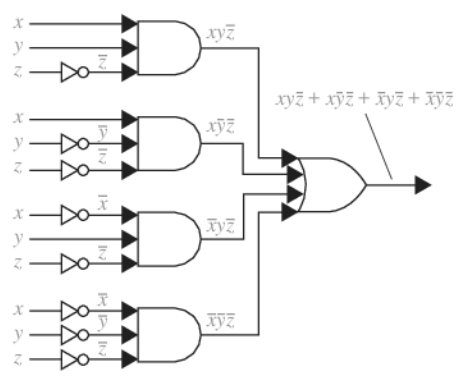
b) Which minterms are represented by cells adjacent to this cell?

6. Use K-maps to find simpler circuits with the same output as each of the circuits shown.

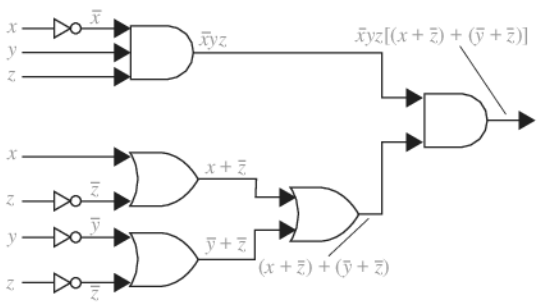
a)



b)



c)



7. Draw the K-maps of these sum-of-products expansions in three variables.

a) $x\overline{y}\,\overline{z}$ **b)** $\overline{x}yz + \overline{x}\,\overline{y}\,\overline{z}$

c) $xyz + xy\overline{z} + \overline{x}yz + \overline{x}\,\overline{y}z$

8. Construct a K-map for $F(x, y, z) = xz + yz + xy\overline{z}$. Use this K-map to find the implicants, prime implicants, and essential prime implicants of $F(x, y, z)$.

9. Construct a K-map for $F(x, y, z) = x\overline{z} + xyz + y\overline{z}$. Use this K-map to find the implicants, prime implicants, and essential prime implicants of $F(x, y, z)$.

10. Draw the 3-cube Q_3 and label each vertex with the minterm in the Boolean variables x, y, and z associated with the bit string represented by this vertex. For each literal in these variables indicate the 2-cube Q_2 that is a subgraph of Q_3 and represents this literal.

11. Draw the 4-cube Q_4 and label each vertex with the minterm in the Boolean variables w, x, y, and z associated with the bit string represented by this vertex. For each literal in these variables, indicate which 3-cube Q_3 that is a subgraph of Q_4 represents this literal. Indicate which 2-cube Q_2 that is a subgraph of Q_4 represents the products wz, $\overline{x}y$, and $\overline{y}\,\overline{z}$.

12. Use a K-map to find a minimal expansion as a Boolean sum of Boolean products of each of these functions in the variables x, y, and z.

a) $\overline{x}yz + \overline{x}\,\overline{y}z$

b) $xyz + xy\overline{z} + \overline{x}yz + \overline{x}y\overline{z}$

c) $xy\overline{z} + x\overline{y}z + x\overline{y}\,\overline{z} + \overline{x}yz + \overline{x}\,\overline{y}z$

d) $xyz + x\overline{y}z + x\overline{y}\,\overline{z} + \overline{x}yz + \overline{x}y\overline{z} + \overline{x}\,\overline{y}\,\overline{z}$

13. a) Draw a K-map for a function in four variables. Put a 1 in the cell that represents $\overline{w}xy\overline{z}$.

b) Which minterms are represented by cells adjacent to this cell?

14. Use a K-map to find a minimal expansion as a Boolean sum of Boolean products of each of these functions in the variables w, x, y, and z.

a) $wxyz + wx\overline{y}z + wx\overline{y}\,\overline{z} + w\overline{x}y\overline{z} + w\overline{x}\,\overline{y}z$

b) $wxy\overline{z} + wx\overline{y}z + w\overline{x}yz + \overline{w}xy\overline{z} + \overline{w}\,\overline{x}\,\overline{y}z$

c) $wxyz + wxy\overline{z} + wx\overline{y}z + w\overline{x}\,\overline{y}z + w\overline{x}\,\overline{y}\,\overline{z} + \overline{w}x\overline{y}z + \overline{w}\,\overline{x}y\overline{z} + \overline{w}\,\overline{x}\,\overline{y}z$

d) $wxyz + wxy\overline{z} + wx\overline{y}z + w\overline{x}yz + w\overline{x}y\overline{z} + \overline{w}xyz + \overline{w}\,\overline{x}yz + \overline{w}\,\overline{x}y\overline{z} + \overline{w}\,\overline{x}\,\overline{y}z$

15. Find the cells in a K-map for Boolean functions with five variables that correspond to each of these products.

a) $x_1x_2x_3x_4$ **b)** $\overline{x}_1x_3x_5$ **c)** x_2x_4

d) $\overline{x}_3\overline{x}_4$ **e)** x_3 **f)** $\overline{x}_5$

16. How many cells in a K-map for Boolean functions with six variables are needed to represent x_1, $\overline{x}_1x_6$, $\overline{x}_1x_2\overline{x}_6$, $x_2x_3x_4x_5$, and $x_1\overline{x}_2x_4\overline{x}_5$, respectively?

17. a) How many cells does a K-map in six variables have?

b) How many cells are adjacent to a given cell in a K-map in six variables?

18. Show that cells in a K-map for Boolean functions in five variables represent minterms that differ in exactly one literal if and only if they are adjacent or are in cells that become adjacent when the top and bottom rows and cells in the first and eighth columns, the first and fourth columns, the second and seventh columns, the third and sixth columns, and the fifth and eighth columns are considered adjacent.

19. Which rows and which columns of a 4×16 map for Boolean functions in six variables using the Gray codes 1111, 1110, 1010, 1011, 1001, 1000, 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101 to label the columns and 11, 10, 00, 01 to label the rows need to be considered adjacent so that cells that represent minterms that differ in exactly one literal are considered adjacent?

***20.** Use K-maps to find a minimal expansion as a Boolean sum of Boolean products of Boolean functions that have as input the binary code for each decimal digit and produce as output a 1 if and only if the digit corresponding to the input is

a) odd. **b)** not divisible by 3.

c) not 4, 5, or 6.

***21.** Suppose that there are five members on a committee, but that Smith and Jones always vote the opposite of Marcus. Design a circuit that implements majority voting of the committee using this relationship between votes.

22. Use the Quine–McCluskey method to simplify the sum-of-products expansions in Example 3.

23. Use the Quine–McCluskey method to simplify the sum-of-products expansions in Exercise 12.

24. Use the Quine–McCluskey method to simplify the sum-of-products expansions in Example 4.

25. Use the Quine–McCluskey method to simplify the sum-of-products expansions in Exercise 14.

***26.** Explain how K-maps can be used to simplify product-of-sums expansions in three variables. [*Hint:* Mark with a 0 all the maxterms in an expansion and combine blocks of maxterms.]

27. Use the method from Exercise 26 to simplify the product-of-sums expansion $(x + y + z)(x + y + \overline{z})(x + \overline{y} + \overline{z})$ $(x + \overline{y} + z)(\overline{x} + y + z)$.

***28.** Draw a K-map for the 16 minterms in four Boolean variables on the surface of a torus.

29. Build a circuit using OR gates, AND gates, and inverters that produces an output of 1 if a decimal digit, encoded using a binary coded decimal expansion, is divisible by 3, and an output of 0 otherwise.

In Exercises 30–32 find a minimal sum-of-products expansion, given the K-map shown with *don't care* conditions indicated with ds.

30.

	yz	$y\bar{z}$	$\bar{y}\bar{z}$	$\bar{y}z$
wx	d	1	d	1
$w\bar{x}$		d	d	
$\bar{w}\bar{x}$		d	1	
$\bar{w}x$		1	d	

31.

	yz	$y\bar{z}$	$\bar{y}\bar{z}$	$\bar{y}z$
wx	1			1
$w\bar{x}$		d	1	
$\bar{w}\bar{x}$		1	d	
$\bar{w}x$	d			d

32.

	yz	$y\bar{z}$	$\bar{y}\bar{z}$	$\bar{y}z$
wx		d	d	1
$w\bar{x}$	d	d	1	d
$\bar{w}\bar{x}$				
$\bar{w}x$	1	1	1	d

33. Show that products of k literals correspond to 2^{n-k}-dimensional subcubes of the n-cube Q_n, where the vertices of the cube correspond to the minterms represented by the bit strings labeling the vertices, as described in Example 8 of Section 10.2.

Key Terms and Results

TERMS

Boolean variable: a variable that assumes only the values 0 and 1

$\bar{x}$ (complement of x): an expression with the value 1 when x has the value 0 and the value 0 when x has the value 1

$x \cdot y$ (or xy) (Boolean product or conjunction of x and y): an expression with the value 1 when both x and y have the value 1 and the value 0 otherwise

$x + y$ (Boolean sum or disjunction of x and y): an expression with the value 1 when either x or y, or both, has the value 1, and 0 otherwise

Boolean expressions: the expressions obtained recursively by specifying that $0, 1, x_1, \ldots, x_n$ are Boolean expressions and $\bar{E}_1$, $(E_1 + E_2)$, and $(E_1 E_2)$ are Boolean expressions if E_1 and E_2 are

dual of a Boolean expression: the expression obtained by interchanging $+$ signs and $\cdot$ signs and interchanging 0s and 1s

Boolean function of degree n: a function from B^n to B where $B = \{0, 1\}$

Boolean algebra: a set B with two binary operations $\vee$ and $\wedge$, elements 0 and 1, and a complementation operator $^-$ that satisfies the identity, complement, associative, commutative, and distributive laws

literal of the Boolean variable x: either x or $\bar{x}$

minterm of $x_1, x_2, \ldots, x_n$: a Boolean product $y_1 y_2 \cdots y_n$, where each y_i is either x_i or $\bar{x}_i$

sum-of-products expansion (or disjunctive normal form): the representation of a Boolean function as a disjunction of minterms

functionally complete: a set of Boolean operators is called functionally complete if every Boolean function can be represented using these operators

$x \mid y$ (or x *NAND* y): the expression that has the value 0 when both x and y have the value 1 and the value 1 otherwise

$x \downarrow y$ (or x *NOR* y): the expression that has the value 0 when either x or y or both have the value 1 and the value 0 otherwise

inverter: a device that accepts the value of a Boolean variable as input and produces the complement of the input

OR gate: a device that accepts the values of two or more Boolean variables as input and produces their Boolean sum as output

AND gate: a device that accepts the values of two or more Boolean variables as input and produces their Boolean product as output

half adder: a circuit that adds two bits, producing a sum bit and a carry bit

full adder: a circuit that adds two bits and a carry, producing a sum bit and a carry bit

K-map for n variables: a rectangle divided into 2^n cells where each cell represents a minterm in the variables

minimization of a Boolean function: representing a Boolean function as the sum of the fewest products of literals such that these products contain the fewest literals possible among all sums of products that represent this Boolean function

implicant of a Boolean function: a product of literals with the property that if this product has the value 1, then the value of this Boolean function is 1

prime implicant of a Boolean function: a product of literals that is an implicant of the Boolean function and no product obtained by deleting a literal is also an implicant of this function

essential prime implicant of a Boolean function: a prime implicant of the Boolean function that must be included in a minimization of this function

don't care condition: a combination of input values for a circuit that is not possible or never occurs

RESULTS

The identities for Boolean algebra (see Table 5 in Section 12.1).

An identity between Boolean functions represented by Boolean expressions remains valid when the duals of both sides of the identity are taken.
Every Boolean function can be represented by a sum-of-products expansion.
Each of the sets $\{+, \bar{}\}$ and $\{\cdot, \bar{}\}$ is functionally complete.
Each of the sets $\{\downarrow\}$ and $\{\mid\}$ is functionally complete.
The use of K-maps to minimize Boolean expressions.
The Quine–McCluskey method for minimizing Boolean expressions.

Review Questions

1. Define a Boolean function of degree n.

2. How many Boolean functions of degree two are there?

3. Give a recursive definition of the set of Boolean expressions.

4. **a)** What is the dual of a Boolean expression?
b) What is the duality principle? How can it be used to find new identities involving Boolean expressions?

5. Explain how to construct the sum-of-products expansion of a Boolean function.

6. **a)** What does it mean for a set of operators to be functionally complete?
b) Is the set $\{+, \cdot\}$ functionally complete?
c) Are there sets of a single operator that are functionally complete?

7. Explain how to build a circuit for a light controlled by two switches using OR gates, AND gates, and inverters.

8. Construct a half adder using OR gates, AND gates, and inverters.

9. Is there a single type of logic gate that can be used to build all circuits that can be built using OR gates, AND gates, and inverters?

10. **a)** Explain how K-maps can be used to simplify sum-of-products expansions in three Boolean variables.
b) Use a K-map to simplify the sum-of-products expansion $xyz + x\bar{y}z + x\bar{y}\,\bar{z} + \bar{x}yz + \bar{x}\,\bar{y}\,\bar{z}$.

11. **a)** Explain how K-maps can be used to simplify sum-of-products expansions in four Boolean variables.
b) Use a K-map to simplify the sum-of-products expansion $wxyz + wxy\bar{z} + wx\bar{y}z + wx\bar{y}\,\bar{z} + w\bar{x}yz + w\bar{x}\,\bar{y}z + \bar{w}xyz + \bar{w}\,\bar{x}yz + \bar{w}\,\bar{x}y\bar{z}$.

12. **a)** What is a *don't care* condition?
b) Explain how *don't care* conditions can be used to build a circuit using OR gates, AND gates, and inverters that produces an output of 1 if a decimal digit is 6 or greater, and an output of 0 if this digit is less than 6.

13. **a)** Explain how to use the Quine–McCluskey method to simplify sum-of-products expansions.
b) Use this method to simplify $xy\bar{z} + x\bar{y}\,\bar{z} + \bar{x}y\bar{z} + \bar{x}\,\bar{y}z + \bar{x}\,\bar{y}\,\bar{z}$.

Supplementary Exercises

1. For which values of the Boolean variables x, y, and z does
a) $x + y + z = xyz$?
b) $x(y + z) = x + yz$?
c) $\bar{x}\,\bar{y}\,\bar{z} = x + y + z$?

2. Let x and y belong to $\{0, 1\}$. Does it necessarily follow that $x = y$ if there exists a value z in $\{0, 1\}$ such that
a) $xz = yz$? **b)** $x + z = y + z$?
c) $x \oplus z = y \oplus z$? **d)** $x \downarrow z = y \downarrow z$?
e) $x \mid z = y \mid z$?

A Boolean function F is called **self-dual** if and only if $F(x_1, \ldots, x_n) = \overline{F(\bar{x}_1, \ldots, \bar{x}_n)}$.

3. Which of these functions are self-dual?
a) $F(x, y) = x$ **b)** $F(x, y) = xy + \bar{x}\,\bar{y}$
c) $F(x, y) = x + y$ **d)** $F(x, y) = xy + \bar{x}y$

4. Give an example of a self-dual Boolean function of three variables.

***5.** How many Boolean functions of degree n are self-dual?

We define the relation $\leq$ on the set of Boolean functions of degree n so that $F \leq G$, where F and G are Boolean functions if and only if $G(x_1, x_2, \ldots, x_n) = 1$ whenever $F(x_1, x_2, \ldots, x_n) = 1$.

6. Determine whether $F \leq G$ or $G \leq F$ for the following pairs of functions.
a) $F(x, y) = x$, $G(x, y) = x + y$
b) $F(x, y) = x + y$, $G(x, y) = xy$
c) $F(x, y) = \bar{x}$, $G(x, y) = x + y$

7. Show that if F and G are Boolean functions of degree n, then
a) $F \leq F + G$. **b)** $FG \leq F$.

8. Show that if F, G, and H are Boolean functions of degree n, then $F + G \leq H$ if and only if $F \leq H$ and $G \leq H$.

***9.** Show that the relation $\leq$ is a partial ordering on the set of Boolean functions of degree n.

***10.** Draw the Hasse diagram for the poset consisting of the set of the 16 Boolean functions of degree two (shown in Table 3 of Section 12.1) with the partial ordering $\leq$.

***11.** For each of these equalities either prove it is an identity or find a set of values of the variables for which it does not hold.
a) $x \mid (y \mid z) = (x \mid y) \mid z$
b) $x \downarrow (y \downarrow z) = (x \downarrow y) \downarrow (x \downarrow z)$
c) $x \downarrow (y \mid z) = (x \downarrow y) \mid (x \downarrow z)$

Define the Boolean operator $\odot$ as follows: $1 \odot 1 = 1, 1 \odot 0 = 0, 0 \odot 1 = 0$, and $0 \odot 0 = 1$.

12. Show that $x \odot y = xy + \overline{x}\,\overline{y}$.

13. Show that $x \odot y = \overline{(x \oplus y)}$.

14. Show that each of these identities holds.

a) $x \odot x = 1$ **b)** $x \odot \overline{x} = 0$

c) $x \odot y = y \odot x$

15. Is it always true that $(x \odot y) \odot z = x \odot (y \odot z)$?

***16.** Determine whether the set $\{\odot\}$ is functionally complete.

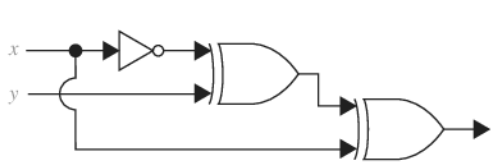
***17.** How many of the 16 Boolean functions in two variables x and y can be represented using only the given set of operators, variables x and y, and values 0 and 1?

a) $\{\overline{\ }\}$ **b)** $\{\cdot\}$ **c)** $\{+\}$ **d)** $\{\cdot, +\}$

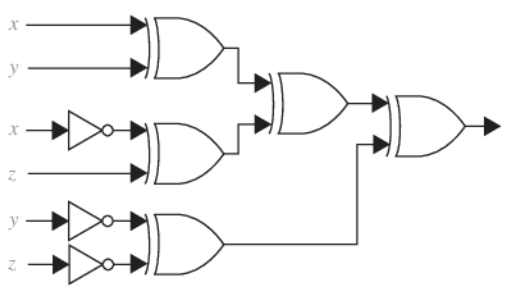
The notation for an **XOR gate**, which produces the output $x \oplus y$ from x and y, is as follows:

18. Determine the output of each of these circuits.

a)



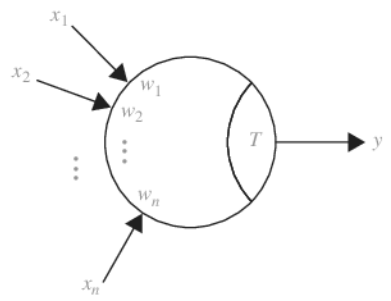
b)



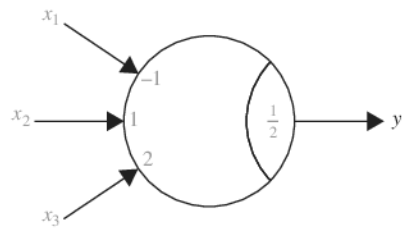
19. Show how a half adder can be constructed using fewer gates than are used in Figure 8 of Section 12.3 when XOR gates can be used in addition to OR gates, AND gates, and inverters.

20. Design a circuit that determines whether three or more of four individuals on a committee vote yes on an issue, where each individual uses a switch for the voting.

A **threshold gate** produces an output y that is either 0 or 1 given a set of input values for the Boolean variables $x_1, x_2, \ldots, x_n$. A threshold gate has a **threshold value** T, which is a real number, and **weights** $w_1, w_2, \ldots, w_n$, each of which is a real number. The output y of the threshold gate is 1 if and only if $w_1x_1 + w_2x_2 + \cdots + w_nx_n \geq T$. The threshold gate with threshold value T and weights $w_1, w_2, \ldots, w_n$ is represented by the following diagram. Threshold gates are useful in modeling in neurophysiology and in artificial intelligence.



21. A threshold gate represents a Boolean function. Find a Boolean expression for the Boolean function represented by this threshold gate.



22. A Boolean function that can be represented by a threshold gate is called a **threshold function**. Show that each of these functions is a threshold function.

a) $F(x) = \overline{x}$ **b)** $F(x, y) = x + y$

c) $F(x, y) = xy$ **d)** $F(x, y) = x \mid y$

e) $F(x, y) = x \downarrow y$ **f)** $F(x, y, z) = x + yz$

g) $F(w, x, y, z) = w + xy + z$

h) $F(w, x, y, z) = wxz + x\overline{y}z$

***23.** Show that $F(x, y) = x \oplus y$ is not a threshold function.

***24.** Show that $F(w, x, y, z) = wx + yz$ is not a threshold function.

Computer Projects

Write programs with these input and output.

1. Given the values of two Boolean variables x and y, find the values of $x + y$, xy, $x \oplus y$, $x \mid y$, and $x \downarrow y$.

2. Construct a table listing the set of values of all 256 Boolean functions of degree three.

3. Given the values of a Boolean function in n variables, where n is a positive integer, construct the sum-of-products expansion of this function.

4. Given the table of values of a Boolean function, express this function using only the operators $\cdot$ and $\overline{\ }$.

5. Given the table of values of a Boolean function, express this function using only the operators $+$ and $\overline{\ }$.

***6.** Given the table of values of a Boolean function, express this function using only the operator $\mid$.

***7.** Given the table of values of a Boolean function, express this function using only the operator $\downarrow$.

8. Given the table of values of a Boolean function of degree three, construct its K-map.

9. Given the table of values of a Boolean function of degree four, construct its K-map.

****10.** Given the table of values of a Boolean function, use the Quine–McCluskey method to find a minimal sum-of-products representation of this function.

11. Given a threshold value and a set of weights for a threshold gate and the values of the n Boolean variables in the input, determine the output of this gate.

12. Given a positive integer n, construct a random Boolean expression in n variables in disjunctive normal form.

Computations and Explorations

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

1. Compute the number of Boolean functions of degrees seven, eight, nine, and ten.

2. Construct a table of the Boolean functions of degree three.

3. Construct a table of the Boolean functions of degree four.

4. Express each of the different Boolean expressions in three variables in disjunctive normal form with just the *NAND* operator, using as few *NAND* operators as possible. What is the largest number of *NAND* operators required?

5. Express each of the different Boolean expressions in disjunctive normal form in four variables using just the *NOR* operator, with as few *NOR* operators as possible. What is the largest number of *NOR* operators required?

6. Randomly generate 10 different Boolean expressions in four variables and determine the average number of steps required to minimize them using the Quine–McCluskey method.

7. Randomly generate 10 different Boolean expressions in five variables and determine the average number of steps required to minimize them using the Quine–McCluskey method.

Writing Projects

Respond to these with essays using outside sources.

1. Describe some of the early machines devised to solve problems in logic, such as the Stanhope Demonstrator, Jevons's Logic Machine, and the Marquand Machine.

2. Explain the difference between combinational circuits and sequential circuits. Then explain how *flip-flops* are used to build sequential circuits.

3. Define a *shift register* and discuss how shift registers are used. Show how to build shift registers using flip-flops and logic gates.

4. Show how *multipliers* can be built using logic gates.

5. Find out how logic gates are physically constructed. Discuss whether NAND and NOR gates are used in building circuits.

6. Explain how *dependency notation* can be used to describe complicated switching circuits.

7. Describe how multiplexers are used to build switching circuits.

8. Explain the advantages of using threshold gates to construct switching circuits. Illustrate this by using threshold gates to construct half and full adders.

9. Describe the concept of *hazard-free switching circuits* and give some of the principles used in designing such circuits.

10. Explain how to use K-maps to minimize functions of six variables.

11. Discuss the ideas used by newer methods for minimizing Boolean functions, such as Espresso. Explain how these methods can help solve minimization problems in as many as 25 variables.

12. Describe what is meant by the *functional decomposition* of a Boolean function of n variables and discuss procedures for decomposing Boolean functions into a composition of Boolean functions with fewer variables.

CHAPTER

13 Modeling Computation

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

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.

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, 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, the, rabbit, mathematician, hops, eats, quickly, wildly*}, and the set of nonterminals is {**sentence, noun phrase, verb phrase, adjective, article, noun, verb, 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 $w_0, w_1, \ldots, w_n$ are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$, then we say that w_n is *derivable from* w_0, and we write $w_0 \stackrel{*}{\Rightarrow} w_n$. 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 $B \to ab$ 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 $B \to ab$, $A \to BB$, $B \to ab$, and $B \to ab$ 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,

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

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 $P = \{S \to aA, S \to b, A \to aa\}$. What is $L(G)$, the language of this grammar?

Solution: From the start state S we can derive aA using the production $S \to aA$. We can also use the production $S \to b$ to derive b. From aA the production $A \to aa$ 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 $P = \{S \to 11S, S \to 0\}$. What is $L(G)$, the language of this grammar?

Solution: From S we can derive 0 using $S \to 0$, or $11S$ using $S \to 11S$. 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 $L(G) = \{0, 110, 11110, 1111110, \ldots\}$, 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 $\{0^n 1^n \mid n = 0, 1, 2, \ldots\}$.

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

$$S \to 0S1$$
$$S \to \lambda.$$

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 grammars can generate the same language.

The grammar G_1 has alphabet $V = \{S, 0, 1\}$; terminals $T = \{0, 1\}$; and productions $S \to 0S$, $S \to S1$, and $S \to \lambda$. 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. The details of this verification are left to the reader.

The grammar G_2 has alphabet $V = \{S, A, 0, 1\}$; terminals $T = \{0, 1\}$; and productions $S \to 0S$, $S \to 1A$, $S \to 1$, $A \to 1A$, $A \to 1$, and $S \to \lambda$. 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 $\{0^n 1^n 2^n \mid n = 0, 1, 2, 3, \ldots\}$ is $G = (V, T, S, P)$ with $V = \{0, 1, 2, S, A, B, C\}$; $T = \{0, 1, 2\}$; starting state S; and productions $S \to C$, $C \to 0CAB$, $S \to \lambda$, $BA \to AB$, $0A \to 01$, $1A \to 11$, $1B \to 12$, and $2B \to 22$. 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. ◀

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 $w_1 \to w_2$, 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 $S \to \lambda$ 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 $w_1 \to w_2$, where w_1 is a single symbol that is not a terminal symbol. A **type 3** grammar can have productions only of the form $w_1 \to w_2$ 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 $lw_1r \to lw_2r$ (but not of the form $w_1 \to w_2$), 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 $w_1 \to w_2$ 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-senstive languages, every production in a type 1 grammar, other than the production $S \to \lambda$, if it is present, is noncontracting. It follows that the lengths of the strings in a derivation in a context-sensitive language are nondecreasing unless the production $S \to \lambda$ 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 $S \to \lambda$ 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 $S \to 0S1$ and $S \to \lambda$. 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.

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 arise in the derivation. The leaves of the tree represent the terminal symbols that arise. If the production $A \to w$ 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.

TABLE 1 Types of Grammars.

Type	*Restrictions on Productions* $w_1 \to w_2$
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 $w_2 = \lambda$

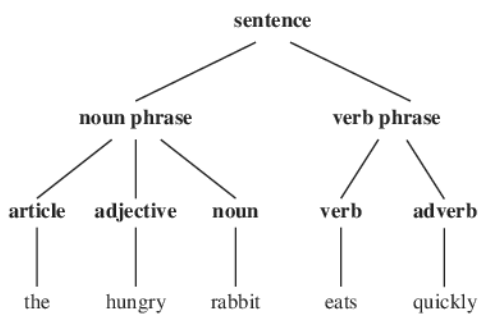


FIGURE 1 **A Derivation Tree.**

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. Two approaches to this problem are indicated in Example 12.

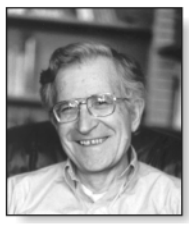
EXAMPLE 12 Determine whether the word *cbab* belongs to the language generated by the grammar $G = (V, T, S, P)$, where $V = \{a, b, c, A, B, C, S\}$, $T = \{a, b, c\}$, S is the starting symbol, and the productions are

$$S \to AB$$
$$A \to Ca$$
$$B \to Ba$$
$$B \to Cb$$
$$B \to b$$
$$C \to cb$$
$$C \to b.$$

Solution: One way to approach 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, namely, $A \to Ca$, to obtain $S \Rightarrow AB \Rightarrow CaB$. Because *cbab* begins with the symbols *cb*, we use the production $C \to cb$. This gives us $S \Rightarrow AB \Rightarrow CaB \Rightarrow cbaB$. We finish by using the production $B \to b$, to obtain $S \Rightarrow AB \Rightarrow CaB \Rightarrow cbaB \Rightarrow cbab$. The approach that we have used is called **top-down parsing**, because it begins with the starting symbol and proceeds by successively applying productions.

There is another approach to this problem, called **bottom-up parsing**. In this approach, we work backward. Because *cbab* is the string to be derived, we can use the production $C \to cb$, so

Links



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.

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 $S \Rightarrow AB \Rightarrow Ab \Rightarrow Cab \Rightarrow cbab$. ◀

Backus–Naur Form

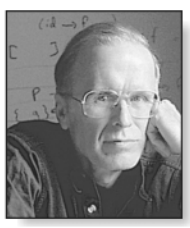
Links

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

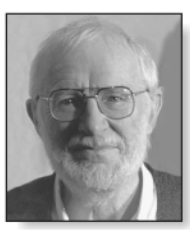
There is another notation that is sometimes used to specify a type 2 grammar, called the **Backus–Naur form (BNF)**, after John Backus, who invented it, and Peter Naur, who refined it for use in the specification of the programming language ALGOL. (Surprisingly, a notation quite similar to the Backus–Naur 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 nonterminal 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.

Links



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 neurosurgery 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. 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 of Computing Machinery in 1977.



PETER NAUR (BORN 1928) 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. In 1969 he became professor of computer science at Copenhagen University, where he has worked in the area of programming methodology. His research interests include the design, structure, and performance of computer programs. Naur has been a pioneer in both the areas of software architecture and software engineering. He rejects the view that computer programming is a branch of mathematics and prefers that computer science be called *datalogy*.

EXAMPLE 13 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:

Extra Examples

⟨*identifier*⟩ ::= ⟨*letter*⟩ | ⟨*identifier*⟩⟨*letter*⟩ | ⟨*identifier*⟩⟨*digit*⟩
⟨*letter*⟩ ::= *a* | *b* | ⋯ | *y* | *z* the ellipsis indicates that all 26 letters are included
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

For example, we can produce the valid identifier $x99a$ by using the first rule to replace ⟨*identifier*⟩ by ⟨*identifier*⟩⟨*letter*⟩, the second rule to obtain ⟨*identifier*⟩*a*, the first rule twice to obtain ⟨*identifier*⟩⟨*digit*⟩⟨*digit*⟩*a*, the third rule twice to obtain ⟨*identifier*⟩99*a*, the first rule to obtain ⟨*letter*⟩99*a*, 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

⟨*sentence*⟩ ::= ⟨*noun phrase*⟩⟨*verb phrase*⟩
⟨*noun phrase*⟩ ::= ⟨*article*⟩⟨*adjective*⟩⟨*noun*⟩ | ⟨*article*⟩⟨*noun*⟩
⟨*verb phrase*⟩ ::= ⟨*verb*⟩⟨*adverb*⟩ | ⟨*verb*⟩
⟨*article*⟩ ::= *a* | *the*
⟨*adjective*⟩ ::= *large* | *hungry*
⟨*noun*⟩ ::= *rabbit* | *mathematician*
⟨*verb*⟩ ::= *eats* | *hops*
⟨*adverb*⟩ ::= *quickly* | *wildly* ◀

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

⟨*signed integer*⟩ ::= ⟨*sign*⟩⟨*integer*⟩
⟨*sign*⟩ ::= + | −
⟨*integer*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩⟨*integer*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ◀

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 $N = \{$**noun phrase, transitive verb phrase, intransitive verb phrase, article, adjective, noun, verb, adverb**$\}$, and productions:

sentence → noun phrase transitive verb phrase
noun phrase → noun phrase
sentence → noun phrase intransitive verb phrase
noun phrase → article adjective noun
noun phrase → article noun
transitive verb phrase → transitive verb
intransitive verb phrase → intransitive verb adverb
intransitive verb phrase → intransitive verb
article → *the*

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

1. Use the set of productions to show that each of these sentences is a valid sentence.
a) *the happy hare runs*
b) *the sleepy tortoise runs quickly*
c) *the tortoise passes the hare*
d) *the sleepy hare passes the happy tortoise*

2. Find five other valid sentences, besides those given in Exercise 1.

3. Show that *the hare runs the sleepy tortoise* is not a valid sentence.

4. 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 $S \to 1S$, $S \to 00A$, $A \to 0A$, and $A \to 0$.
a) Show that 111000 belongs to the language generated by G.
b) Show that 11001 does not belong to the language generated by G.
c) What is the language generated by G?

5. 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 $S \to 0A$, $S \to 1A$, $A \to 0B$, $B \to 1A$, $B \to 1$.
a) Show that 10101 belongs to the language generated by G.
b) Show that 10110 does not belong to the language generated by G.
c) What is the language generated by G?

***6.** 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
a) $S \to AB$, $A \to ab$, $B \to bb$.
b) $S \to AB$, $S \to aA$, $A \to a$, $B \to ba$.
c) $S \to AB$, $S \to AA$, $A \to aB$, $A \to ab$, $B \to b$.
d) $S \to AA$, $S \to B$, $A \to aaA$, $A \to aa$, $B \to bB$, $B \to b$.
e) $S \to AB$, $A \to aAb$, $B \to bBa$, $A \to \lambda$, $B \to \lambda$.

7. Construct a derivation of $0^3 1^3$ using the grammar given in Example 5.

8. Show that the grammar given in Example 5 generates the set $\{0^n 1^n \mid n = 0, 1, 2, \ldots\}$.

9. a) Construct a derivation of $0^2 1^4$ using the grammar G_1 in Example 6.
b) Construct a derivation of $0^2 1^4$ using the grammar G_2 in Example 6.

10. a) Show that the grammar G_1 given in Example 6 generates the set $\{0^m 1^n \mid m, n = 0, 1, 2, \ldots\}$.
b) Show that the grammar G_2 in Example 6 generates the same set.

11. Construct a derivation of $0^2 1^2 2^2$ in the grammar given in Example 7.

***12.** Show that the grammar given in Example 7 generates the set $\{0^n 1^n 2^n \mid n = 0, 1, 2, \ldots\}$.

13. Find a phrase-structure grammar for each of these languages.
a) the set consisting of the bit strings 0, 1, and 11
b) the set of bit strings containing only 1s
c) the set of bit strings that start with 0 and end with 1
d) the set of bit strings that consist of a 0 followed by an even number of 1s

14. Find a phrase-structure grammar for each of these languages.
a) the set consisting of the bit strings 10, 01, and 101
b) the set of bit strings that start with 00 and end with one or more 1s
c) 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.
a) the set of all bit strings containing an even number of 0s and no 1s
b) the set of all bit strings made up of a 1 followed by an odd number of 0s
c) the set of all bit strings containing an even number of 0s and an even number of 1s
d) the set of all strings containing 10 or more 0s and no 1s
e) the set of all strings containing more 0s than 1s
f) the set of all strings containing an equal number of 0s and 1s
g) the set of all strings containing an unequal number of 0s and 1s

16. Construct phrase-structure grammars to generate each of these sets.
a) $\{1^n \mid n \geq 0\}$ **b)** $\{10^n \mid n \geq 0\}$
c) $\{(11)^n \mid n \geq 0\}$

17. Construct phrase-structure grammars to generate each of these sets.
a) $\{0^n \mid n \geq 0\}$ **b)** $\{1^n 0 \mid n \geq 0\}$
c) $\{(000)^n \mid n \geq 0\}$

18. Construct phrase-structure grammars to generate each of these sets.
a) $\{01^{2n} \mid n \geq 0\}$
b) $\{0^n 1^{2n} \mid n \geq 0\}$
c) $\{0^n 1^m 0^n \mid m \geq 0 \text{ and } n \geq 0\}$

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
a) $S \to aAB$, $A \to Bb$, $B \to \lambda$.

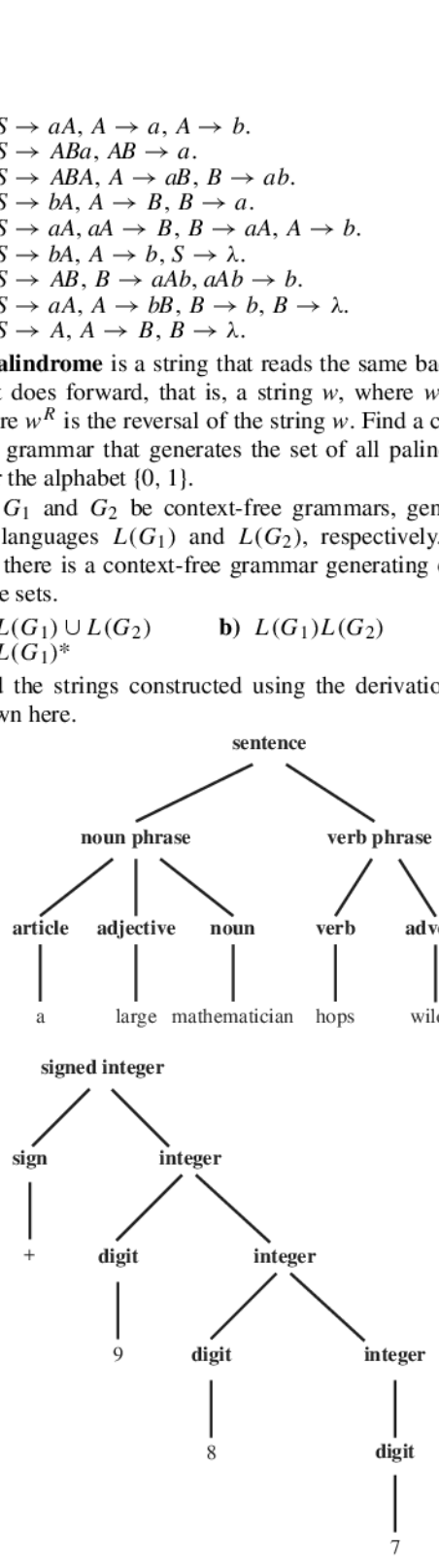
b) $S \to aA$, $A \to a$, $A \to b$.
c) $S \to ABa$, $AB \to a$.
d) $S \to ABA$, $A \to aB$, $B \to ab$.
e) $S \to bA$, $A \to B$, $B \to a$.
f) $S \to aA$, $aA \to B$, $B \to aA$, $A \to b$.
g) $S \to bA$, $A \to b$, $S \to \lambda$.
h) $S \to AB$, $B \to aAb$, $aAb \to b$.
i) $S \to aA$, $A \to bB$, $B \to b$, $B \to \lambda$.
j) $S \to A$, $A \to B$, $B \to \lambda$.

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.

a) $L(G_1) \cup L(G_2)$ **b)** $L(G_1)L(G_2)$
c) $L(G_1)^*$

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 $S \to abS$, $S \to bcS$, $S \to bbS$, $S \to a$, and $S \to cb$. Construct derivation trees for

a) *bcbba.* **b)** *bbbcbba.*
c) *bcabbbbbcb.*

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

a) *baba* **b)** *abab*
c) *cbaba* **d)** *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:

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

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

a) one or more lowercase letters.
b) at least three but no more than six lowercase letters.
c) one to six uppercase or lowercase letters beginning with an uppercase letter.
d) 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.

a) *string* ::= $L+D?L+$
$L ::= a \mid b \mid c$
$D ::= 0 \mid 1$

b) *string* ::= *sign* $D+ \mid D+$
sign ::= $+ \mid -$
$D ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

c) *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.

⟨*expression*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩⟨*term*⟩⟨*addOperator*⟩
⟨*addOperator*⟩ ::= $+ \mid -$
⟨*term*⟩ ::= ⟨*factor*⟩ | ⟨*factor*⟩⟨*factor*⟩⟨*mulOperator*⟩
⟨*mulOperator*⟩ ::= $* \mid /$
⟨*factor*⟩ ::= ⟨*identifier*⟩ | ⟨*expression*⟩
⟨*identifier*⟩ ::= $a \mid b \mid \cdots \mid z$

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

a) $abc*+$ **b)** $xy++$
c) $xy-z*$ **d)** $wxyz-*/$
e) $ade-*$

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

Introduction

Links

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.

TABLE 1 State Table for a Vending Machine.

	Next State					Output				
	Input					*Input*				
State	**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

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.

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

The machine can be in any of seven different states s_i, $i = 0, 1, 2, \ldots, 6$, 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 with the input and the output for that transition. Figure 1 shows such a directed graph for the vending machine.

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.

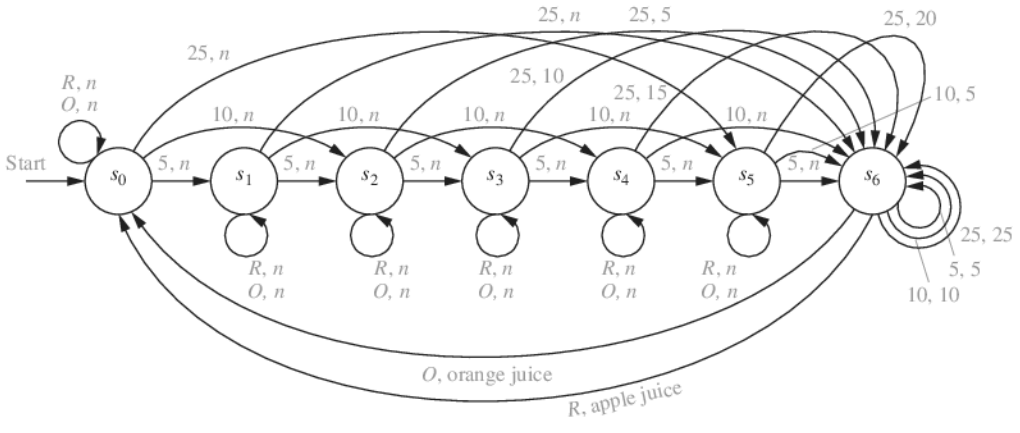


FIGURE 1 A Vending Machine.

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. ◀

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. ◀

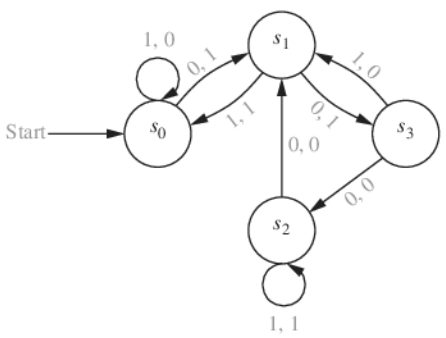


FIGURE 2 The State Diagram for the Finite-State Machine Shown in Table 2.

TABLE 2

	f		*g*	
	Input		*Input*	
State	**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

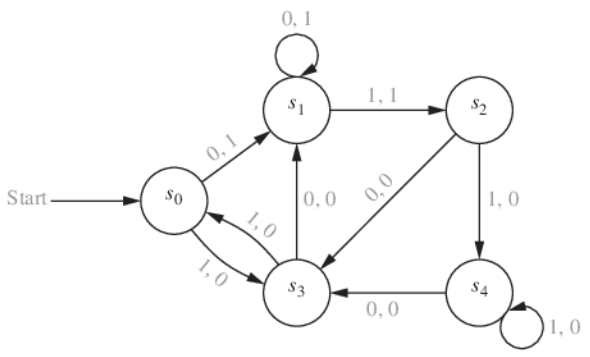


FIGURE 3 **A Finite-State Machine.**

TABLE 3				
	f		*g*	
	Input		*Input*	
State	**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

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 $x = x_1x_2 \ldots x_k$. 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 $j = 1, 2, \ldots, k$, ending at state $s_k = f(s_{k-1}, x_k)$. This sequence of transitions produces an output string $y_1y_2 \ldots y_k$, 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 $j = 1, 2, \ldots, k$. Hence, we can extend the definition of 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 $0x_1x_2 \ldots x_{k-1}$ given the input bit string $x_1x_2 \ldots x_k$?

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

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	—

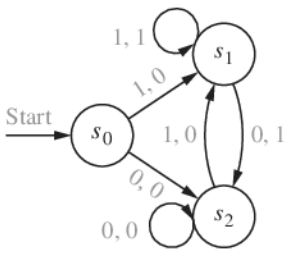


FIGURE 4 **A Unit-Delay Machine.**

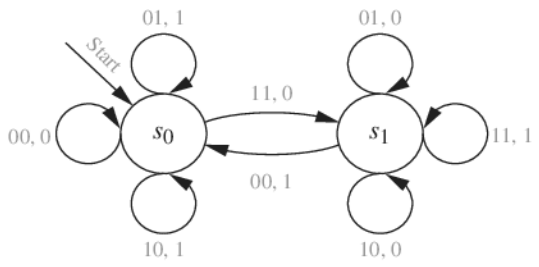


FIGURE 5 **A Finite-State Machine for Addition.**

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 $x_1 \ldots x_k$ is the string that begins with 0, followed by x_1, followed by $x_2, \ldots$, 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 $(x_n \ldots x_1 x_0)_2$ and $(y_n \ldots y_1 y_0)_2$ 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 nth 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 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

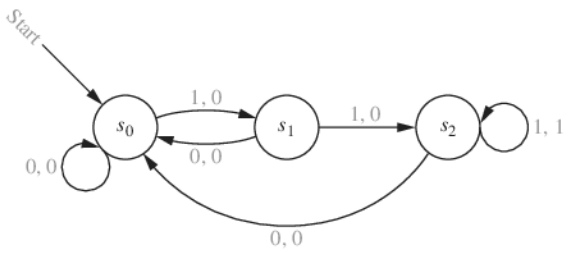


FIGURE 6 A Finite-State Machine That Gives an Output of 1 If and Only If the Input String Read So Far Ends with 111.

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. ◀

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)

	f		g	
	Input		*Input*	
State	**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)

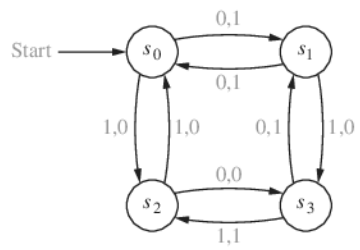
	f		g	
	Input		*Input*	
State	**0**	**1**	**0**	**1**
s_0	s_1	s_0	0	0
s_1	s_2	s_0	1	1
s_2	s_0	s_3	0	1
s_3	s_1	s_2	1	0

c)

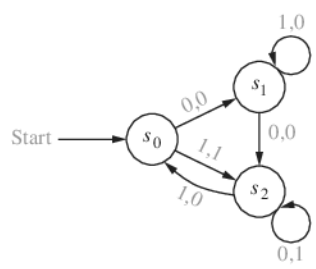
	f		*g*	
	Input		***Input***	
State	**0**	**1**	**0**	**1**
s_0	s_0	s_4	1	1
s_1	s_0	s_3	0	1
s_2	s_0	s_2	0	0
s_3	s_1	s_1	1	1
s_4	s_1	s_0	1	0

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

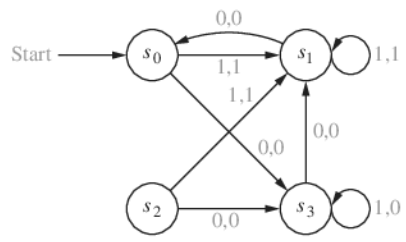
a)



b)



c)



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

4. Find the output generated from the input string 10001 for the finite-state machine with the state diagram in
- **a)** Exercise 2(a).
- **b)** Exercise 2(b).
- **c)** 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, which is considered to be a single input, and then a password, 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 islocked.

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.

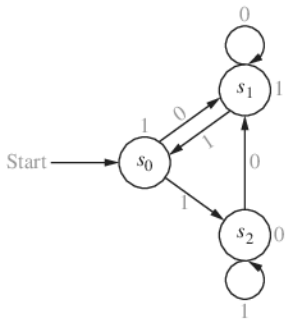
19. Construct a finite-state machine that determines whether the word *computer* has been read as the last eight characters in the input read so far, where the input can be any string of English letters.

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 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.

	f		
	Input		
State	**0**	**1**	g
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 $a_1a_2 \dots a_k$ 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

Introduction

Links

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.

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 $n = 0, 1, 2, \ldots$. This is done recursively by specifying that

$$A^0 = \{\lambda\},$$
$$A^{n+1} = A^n A \quad \text{for } n = 0, 1, 2, \ldots.$$

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, $A^* = \{0^n \mid n = 0, 1, 2, \ldots\}$. 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 closurebreak 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, $C^* = \{1^{2n} \mid n = 0, 1, 2, \ldots\}$. ◀

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.

TABLE 1		
	f	
	Input	
State	**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

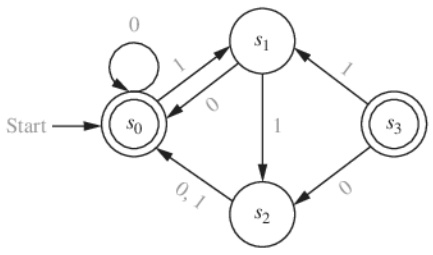


FIGURE 1 The State Diagram for a Finite-State Automaton.

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 $f : S \times I \to S$), 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.

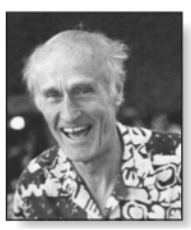
EXAMPLE 4 Construct the state diagram for the finite-state automaton $M = (S, I, f, s_0, F)$, where $S = \{s_0, s_1, s_2, s_3\}$, $I = \{0, 1\}$, $F = \{s_0, s_3\}$, and the transition function f is given in Table 1.

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. ◀

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 $f : S \times I^* \to S$. Let $x = x_1 x_2 \ldots x_k$ 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 $a \in I$.

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.

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 $y \in I^*$.

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.

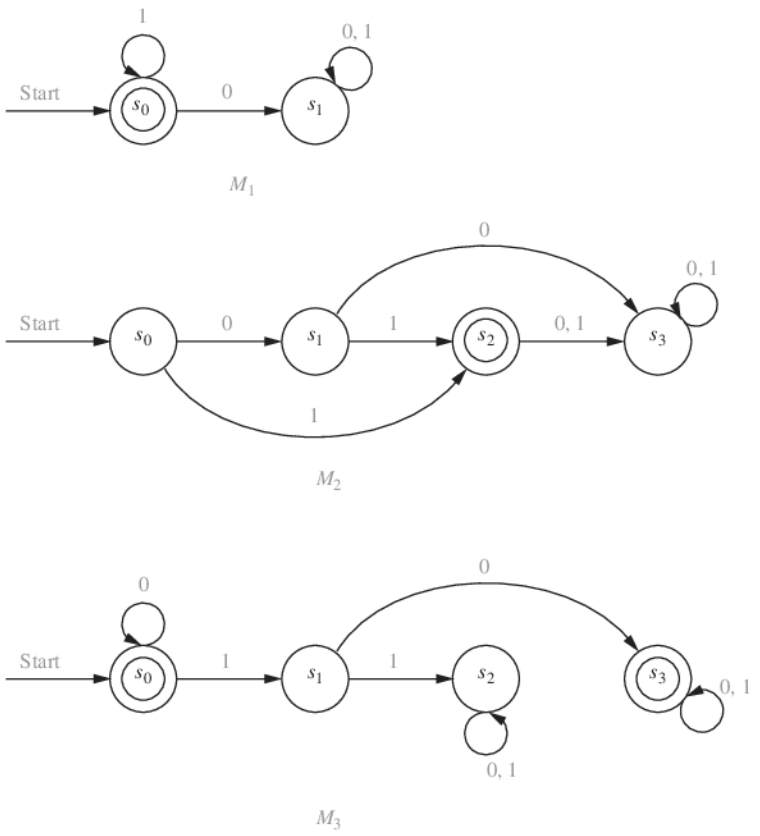


FIGURE 2 Some Finite-State Automata.

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, $L(M_1) = \{1^n \mid n = 0, 1, 2, \ldots\}$.

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, ..., 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, $L(M_3) = \{0^n, 0^n 10x \mid n = 0, 1, 2, \ldots,$ and x is any string$\}$. ◀

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

(a) the set of bit strings that begin with two 0s
(b) the set of bit strings that contain two consecutive 0s
(c) the set of bit strings that do not contain two consecutive 0s
(d) the set of bit strings that end with two 0s
(e) 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.

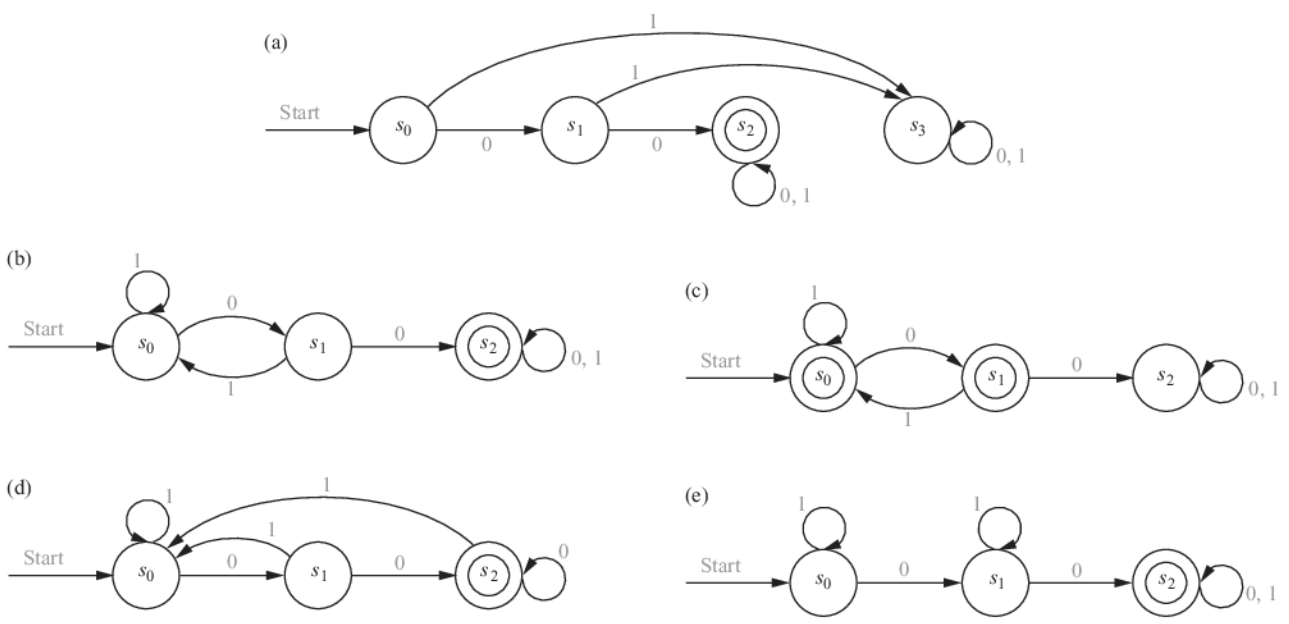


FIGURE 3 Deterministic Finite-State Automata Recognizing the Languages in Example 6.

(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. ◀

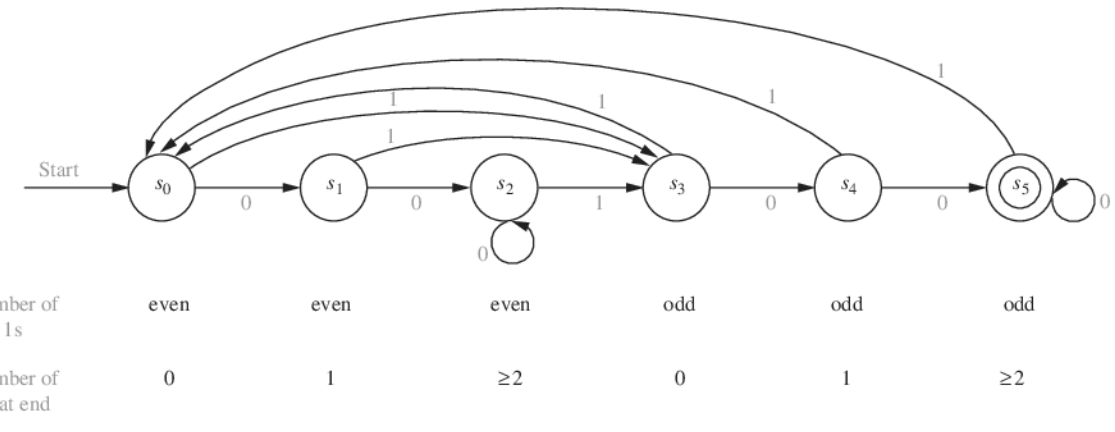


FIGURE 4 A Deterministic Finite-State Automaton Recognizing the Set of Bit Strings Containing an Odd Number of 1s and Ending with at Least Two 0s.

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.

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

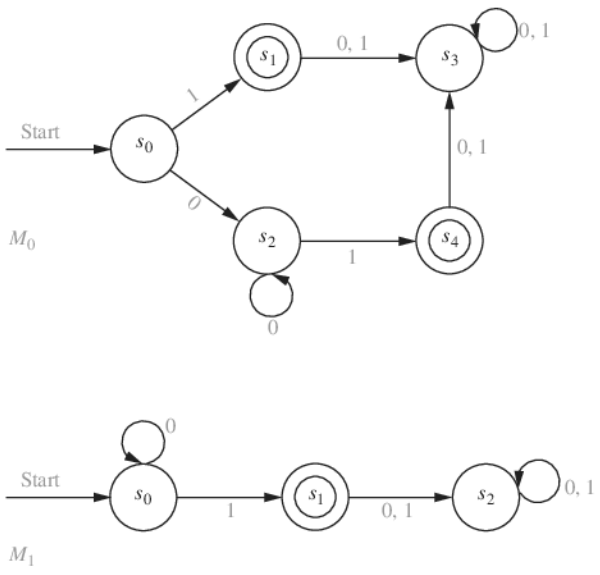


FIGURE 5 M_0 and M_1 Are Equivalent Finite-State Automata.

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.

Links

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 the first commercial computer, UNIVAC. 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*.

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. ◀

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 $x = x_1x_2 \ldots x_k$? 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 a 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.

TABLE 2

	f	
	Input	
State	**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

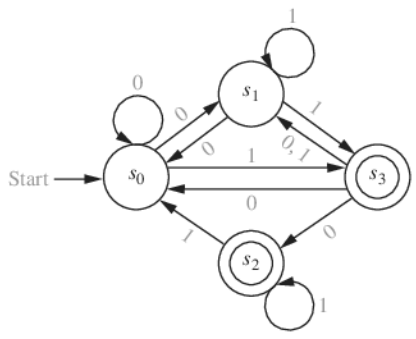


FIGURE 6 The Nondeterministic Finite-State Automaton with State Table Given in Table 2.

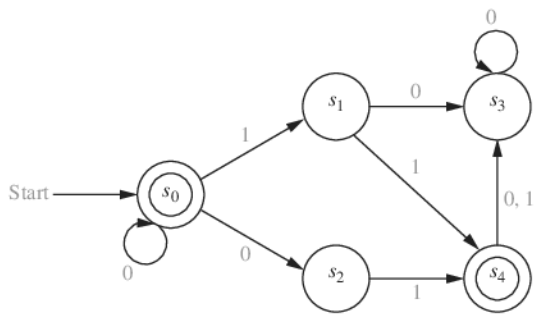


FIGURE 7 A Nondeterministic Finite-State Automaton.

TABLE 3

	f	
	Input	
State	**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

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 $\{0^n, 0^n01, 0^n11 \mid n \geq 0\}$. ◀

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 $\{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ 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 $f(s_{i_1}, x)$, $f(s_{i_2}, x), \ldots, f(s_{i_k}, x)$. 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.

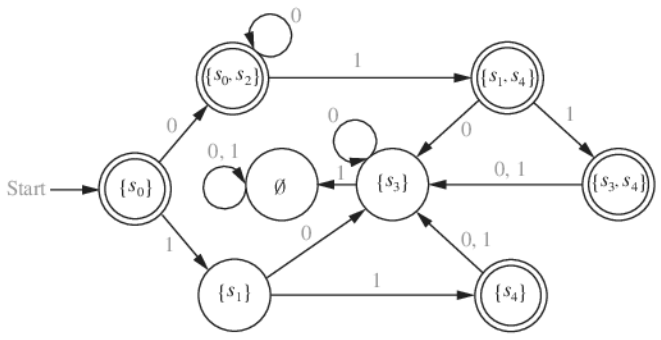


FIGURE 8 A Deterministic Automaton Equivalent to the Nondeterministic Automaton in Example 10.

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 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

1. Let $A = \{0, 11\}$ and $B = \{00, 01\}$. Find each of these sets.
a) AB **b)** BA **c)** A^2 **d)** B^3

2. Show that if A is a set of strings, then $A\emptyset = \emptyset A = \emptyset$.

3. Find all pairs of sets of strings A and B for which $AB = \{10, 111, 1010, 1000, 10111, 101000\}$.

4. Show that these equalities hold.
a) $\{\lambda\}^* = \{\lambda\}$
b) $(A^*)^* = A^*$ for every set of strings A

5. Describe the elements of the set A^* for these values of A.
a) $\{10\}$ **b)** $\{111\}$ **c)** $\{0, 01\}$ **d)** $\{1, 101\}$

6. Let V be an alphabet, and let A and B be subsets of V^*. Show that $|AB| \leq |A||B|$.

7. Let V be an alphabet, and let A and B be subsets of V^* with $A \subseteq B$. Show that $A^* \subseteq B^*$.

8. 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)** $|A^n| = |A|^n$

9. 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)** $\{11, 0\}\{00, 101\}$

10. 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)** $\{01\}^*\{01\}^*$

11. 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

12. 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

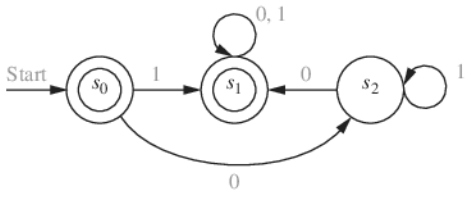
13. Determine whether all the strings in each of these sets are recognized by the deterministic finite-state automaton in Figure 1.
a) $\{0\}^*$ **b)** $\{0\}\{0\}^*$ **c)** $\{1\}\{0\}^*$
d) $\{01\}^*$ **e)** $\{0\}^*\{1\}^*$ **f)** $\{1\}\{0, 1\}^*$

14. 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.)

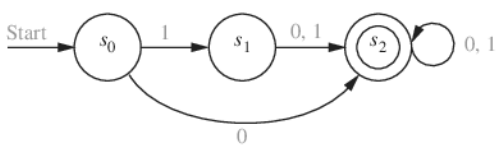
15. Given a deterministic finite-state automaton $M = (S, I, f, s_0, F)$, use structural induction and the recursive definition of the extended transition function f to prove that $f(s, xy) = f(f(s, x), y)$ for all states $s \in S$ and all strings $x \in I^*$ and $y \in I^*$.

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

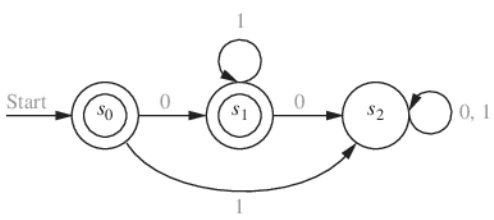
16.



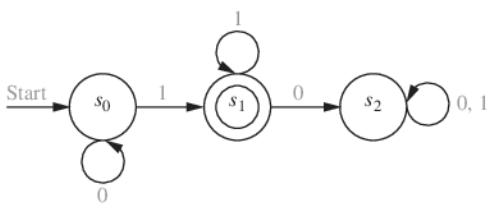
17.



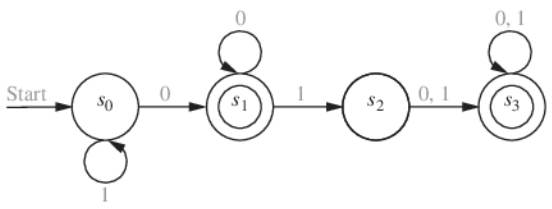
18.



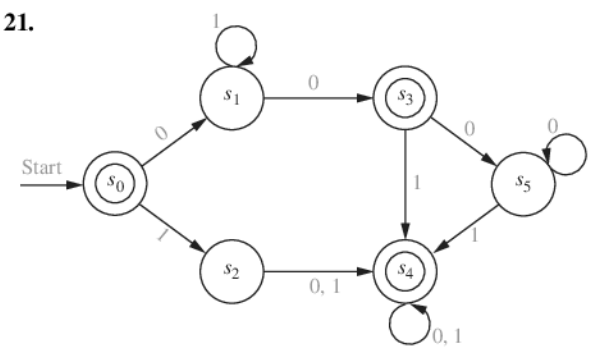
19.



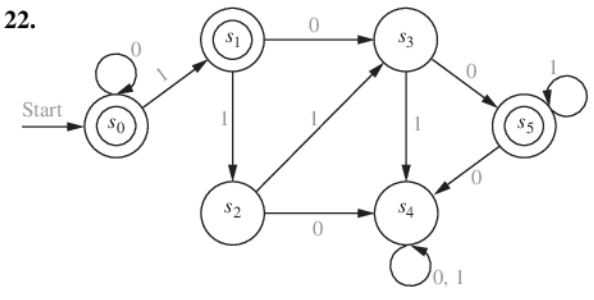
20.



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)

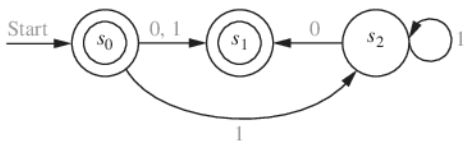
41. Use the procedure you described in Exercise 39 and the finite-state automata you constructed in Exercise 25 to find a deterministic finite-state automaton that recognizes the set of all bit strings that do not contain the string 101.

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.

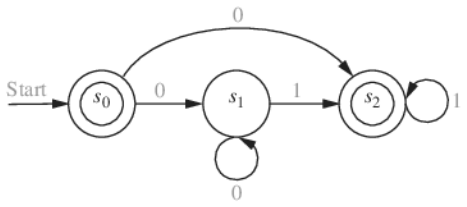
In Exercises 43–49 find the language recognized by the given nondeterministic finite-state automaton.

43.

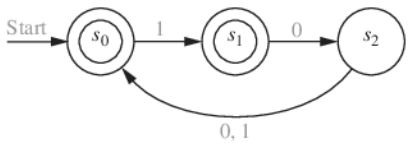
44.



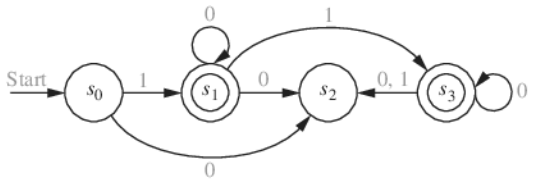
45.



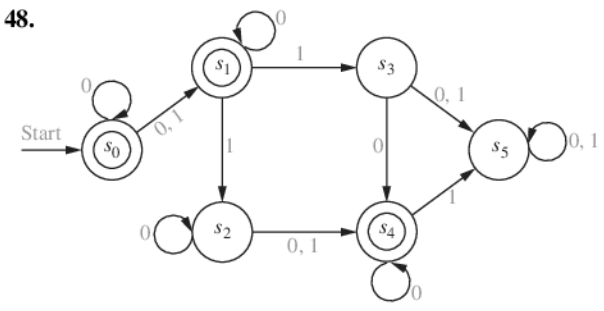
46.



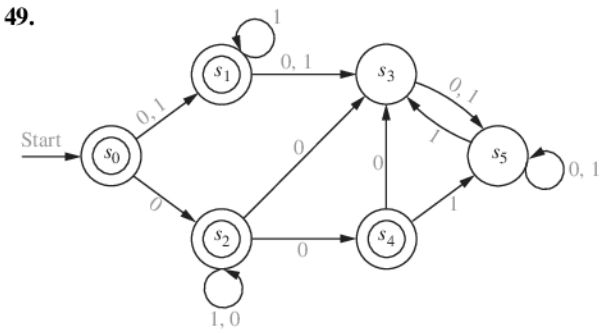
47.



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) $\{1^n \mid n = 2, 3, 4, \ldots\}$

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 $s R_k t$ 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_kt.

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

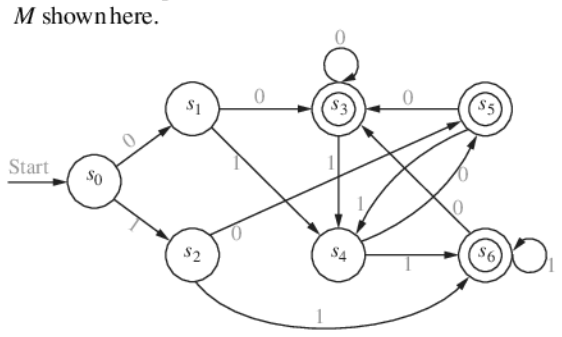
b) Show that if k is a positive integer, then s and t are k-equivalent if and only if s and t are $(k-1)$-equivalent and for every input symbol $a \in I$, $f(s, a)$ and $f(t, a)$ are $(k-1)$-equivalent. Conclude that the transition function $\overline{f}$ is well-defined.

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 $\overline{M}$ 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 $\overline{M}$ 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 $\overline{M}$ of M.

13.4 Language Recognition

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 $\boldsymbol{x}$ 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;
$\boldsymbol{\lambda}$ represents the set $\{\lambda\}$, which is the set containing the empty string;
$\boldsymbol{x}$ 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 $\mathbf{10}^*$, $(\mathbf{10})^*$, $\mathbf{0} \cup \mathbf{01}$, $\mathbf{0}(\mathbf{0} \cup \mathbf{1})^*$, and $(\mathbf{0}^*\mathbf{1})^*$?

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

TABLE 1

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