CHAPTER

12 Boolean Algebra

The circuits in computers and other electronic devices have inputs, each of which is either a 0 or a 1, and produce outputs that are also 0s and 1s. Circuits can be constructed using any basic element that has two different states. Such elements include switches that can be in either the on or the off position and optical devices that can be either lit or unlit. In 1938 Claude Shannon showed how the basic rules of logic, first given by George Boole in 1854 in his *The Laws of Thought*, could be used to design circuits. These rules form the basis for Boolean algebra. In this chapter we develop the basic properties of Boolean algebra. The operation of a circuit is defined by a Boolean function that specifies the value of an output for each set of inputs. The first step in constructing a circuit is to represent its Boolean function by an expression built up using the basic operations of Boolean algebra. We will provide an algorithm for producing such expressions. The expression that we obtain may contain many more operations than are necessary to represent the function. Later in the chapter we will describe methods for finding an expression with the minimum number of sums and products that represents a Boolean function. The procedures that we will develop, Karnaugh maps and the Quine–McCluskey method, are important in the design of efficient circuits.

12.1 Boolean Functions

12.1.1 Introduction

Boolean algebra provides the operations and the rules for working with the set $\{0, 1\}$. Electronic and optical switches can be studied using this set and the rules of Boolean algebra. The three operations in Boolean algebra that we will use most are complementation, the Boolean sum, and the Boolean product. The **complement** of an element, denoted with a bar, is defined by $\overline{0} = 1$ and $\overline{1} = 0$. The Boolean sum, denoted by $+$ or by *OR*, has the following values:

$$1 + 1 = 1, \quad 1 + 0 = 1, \quad 0 + 1 = 1, \quad 0 + 0 = 0.$$

The Boolean product, denoted by $\cdot$ or by *AND*, has the following values:

$$1 \cdot 1 = 1, \quad 1 \cdot 0 = 0, \quad 0 \cdot 1 = 0, \quad 0 \cdot 0 = 0.$$

When there is no danger of confusion, the symbol $\cdot$ can be deleted, just as in writing algebraic products. Unless parentheses are used, the rules of precedence for Boolean operators are: first, all complements are computed, followed by all Boolean products, followed by all Boolean sums. This is illustrated in Example 1.

EXAMPLE

Find the value of $1 \cdot 0 + \overline{(0 + 1)}$.

Solution: Using the definitions of complementation, the Boolean sum, and the Boolean product, it follows that

$$\begin{aligned} 1 \cdot 0 + \overline{(0 + 1)} &= 0 + \overline{1} \\ &= 0 + 0 \\ &= 0. \end{aligned}$$

◀

The complement, Boolean sum, and Boolean product correspond to the logical operators, $\neg$, $\vee$, and $\wedge$, respectively, where 0 corresponds to **F** (false) and 1 corresponds to **T** (true). Equalities in Boolean algebra can be directly translated into equivalences of compound propositions. Conversely, equivalences of compound propositions can be translated into equalities in Boolean algebra. We will see later in this section why these translations yield valid logical equivalences and identities in Boolean algebra. Example 2 illustrates the translation from Boolean algebra to propositional logic.

EXAMPLE 2 Translate $1 \cdot 0 + \overline{(0 + 1)} = 0$, the equality found in Example 1, into a logical equivalence.

Solution: We obtain a logical equivalence when we translate each 1 into a **T**, each 0 into an **F**, each Boolean sum into a disjunction, each Boolean product into a conjunction, and each complementation into a negation. We obtain

$$(\mathbf{T} \wedge \mathbf{F}) \vee \neg(\mathbf{T} \vee \mathbf{F}) \equiv \mathbf{F}.$$

Example 3 illustrates the translation from propositional logic to Boolean algebra.

EXAMPLE 3 Translate the logical equivalence $(\mathbf{T} \wedge \mathbf{T}) \vee \neg\mathbf{F} \equiv \mathbf{T}$ into an identity in Boolean algebra.

Solution: We obtain an identity in Boolean algebra when we translate each **T** into a 1, each **F** into a 0, each disjunction into a Boolean sum, each conjunction into a Boolean product, and each negation into a complementation. We obtain

$$(1 \cdot 1) + \bar{0} = 1.$$

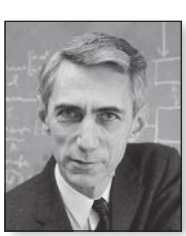
12.1.2 Boolean Expressions and Boolean Functions

Let $B = \{0, 1\}$. Then $B^n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in B \text{ for } 1 \le i \le n\}$ is the set of all possible n-tuples of 0s and 1s. The variable x is called a **Boolean variable** if it assumes values only from B, that is, if its only possible values are 0 and 1. A function from B^n to B is called a **Boolean function of degree n**.

EXAMPLE 4 The function $F(x, y) = x\bar{y}$ from the set of ordered pairs of Boolean variables to the set $\{0, 1\}$ is a Boolean function of degree 2 with $F(1, 1) = 0$, $F(1, 0) = 1$, $F(0, 1) = 0$, and $F(0, 0) = 0$. We display these values of F in Table 1.

Links

CLAUDE ELWOOD SHANNON (1916–2001) Claude Shannon was born in Petoskey, Michigan, and grew up in Gaylord, Michigan. His father was a businessman and a probate judge, and his mother was a language teacher and a high school principal. Shannon attended the University of Michigan, graduating in 1936. He continued his studies at M.I.T., where he took the job of maintaining the differential analyzer, a mechanical computing device consisting of shafts and gears built by his professor, Vannevar Bush. Shannon's master's thesis, written in 1936, studied the logical aspects of the differential analyzer. This master's thesis presents the first application of Boolean algebra to the design of switching circuits; it is perhaps the most famous master's thesis of the twentieth century. He received his Ph.D. from M.I.T. in 1940. Shannon joined Bell Laboratories in 1940, where he worked on transmitting data efficiently. He was one of the first people to use bits to represent information. At Bell Laboratories he worked on determining the amount of traffic that telephone lines can carry. Shannon made many fundamental contributions to information theory. In the early 1950s he was one of the founders of the study of artificial intelligence. He joined the M.I.T. faculty in 1956, where he continued his study of information theory.



©Alfred Eisenstaedt/The LIFE Picture Collection/ Getty Images

Shannon had an unconventional side. He is credited with inventing the rocket-powered Frisbee. He is also famous for riding a unicycle down the hallways of Bell Laboratories while juggling four balls. Shannon retired when he was 50 years old, publishing papers sporadically over the following 10 years. In his later years he concentrated on some pet projects, such as building a motorized pogo stick. One interesting quote from Shannon, published in *Omni Magazine* in 1987, is "I visualize a time when we will be to robots what dogs are to humans. And I am rooting for the machines."

TABLE 1		
x	y	$F(x, y)$
1	1	0
1	0	1
0	1	0
0	0	0

Boolean functions can be represented using expressions made up from variables and Boolean operations. The **Boolean expressions** in the variables $x_1, x_2, \ldots, x_n$ are defined recursively as

$0, 1, x_1, x_2, \ldots, x_n$ are Boolean expressions;

if E_1 and E_2 are Boolean expressions, then $\overline{E}_1$, (E_1E_2), and $(E_1 + E_2)$ are Boolean expressions.

Each Boolean expression represents a Boolean function. The values of this function are obtained by substituting 0 and 1 for the variables in the expression. In Section 12.2 we will show that every Boolean function can be represented by a Boolean expression.

EXAMPLE 5 Find the values of the Boolean function represented by $F(x, y, z) = xy + \bar{z}$.

Solution: The values of this function are displayed in Table 2. ◀

TABLE 2					
x	y	z	xy	$\bar{z}$	$F(x, y, z) = xy + \bar{z}$
1	1	1	1	0	1
1	1	0	1	1	1
1	0	1	0	0	0
1	0	0	0	1	1
0	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	0
0	0	0	0	1	1

Note that we can represent a Boolean function graphically by distinguishing the vertices of the n-cube that correspond to the n-tuples of bits where the function has value 1.

EXAMPLE 6 The function $F(x, y, z) = xy + \bar{z}$ from B^3 to B from Example 5 can be represented by distinguishing the vertices that correspond to the five 3-tuples (1, 1, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), and (0, 0, 0), where $F(x, y, z) = 1$, as shown in Figure 1. These vertices are displayed using solid black circles.

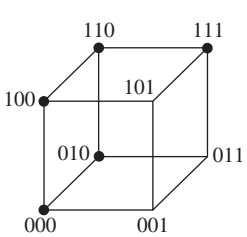


FIGURE 1

Boolean functions F and G of n variables are equal if and only if $F(b_1, b_2, \ldots, b_n) = G(b_1, b_2, \ldots, b_n)$ whenever $b_1, b_2, \ldots, b_n$ belong to B. Two different Boolean expressions that represent the same function are called **equivalent**. For instance, the Boolean expressions xy, $xy + 0$, and $xy \cdot 1$ are equivalent. The **complement** of the Boolean function F is the function $\overline{F}$, where $\overline{F}(x_1, \ldots, x_n) = \overline{F(x_1, \ldots, x_n)}$. Let F and G be Boolean functions of degree n. The **Boolean sum** $F + G$ and the **Boolean product** FG are defined by

$$(F + G)(x_1, \ldots, x_n) = F(x_1, \ldots, x_n) + G(x_1, \ldots, x_n),$$
$$(FG)(x_1, \ldots, x_n) = F(x_1, \ldots, x_n)G(x_1, \ldots, x_n).$$

A Boolean function of degree two is a function from a set with four elements, namely, pairs of elements from $B = \{0, 1\}$, to B, a set with two elements. Hence, there are 16 different Boolean functions of degree two. In Table 3 we display the values of the 16 different Boolean functions of degree two, labeled $F_1, F_2, \ldots, F_{16}$.

TABLE 3 The 16 Boolean Functions of Degree Two.

x	y	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}
1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
0	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

EXAMPLE 7 How many different Boolean functions of degree n are there?

Solution: From the product rule for counting, it follows that there are 2^n different n-tuples of 0s and 1s. Because a Boolean function is an assignment of 0 or 1 to each of these 2^n different n-tuples, the product rule shows that there are 2^{2^n} different Boolean functions of degree n. ◀

Table 4 displays the number of different Boolean functions of degrees one through six. The number of such functions grows extremely rapidly.

TABLE 4 The Number of Boolean Functions of Degree n.

Degree	*Number*
1	4
2	16
3	256
4	65,536
5	4,294,967,296
6	18,446,744,073,709,551,616

12.1.3 Identities of Boolean Algebra

There are many identities in Boolean algebra. The most important of these are displayed in Table 5. These identities are particularly useful in simplifying the design of circuits. Each of the identities in Table 5 can be proved using a table. We will prove one of the distributive laws in this way in Example 8. The proofs of the remaining properties are left as exercises for the reader.

EXAMPLE 8 Show that the distributive law $x(y + z) = xy + xz$ is valid.

Solution: The verification of this identity is shown in Table 6. The identity holds because the last two columns of the table agree. ◀

The reader should compare the Boolean identities in Table 5 to the logical equivalences in Table 6 of Section 1.3 and the set identities in Table 1 in Section 2.2. All are special cases of the same set of identities in a more abstract structure. Each collection of identities can be obtained by making the appropriate translations. For example, we can transform each of the identities in

TABLE 5 Boolean Identities.

Identity	*Name*
$\overline{\overline{x}} = x$	Law of the double complement
$x + x = x$ $x \cdot x = x$	Idempotent laws
$x + 0 = x$ $x \cdot 1 = x$	Identity laws
$x + 1 = 1$ $x \cdot 0 = 0$	Domination laws
$x + y = y + x$ $xy = yx$	Commutative laws
$x + (y + z) = (x + y) + z$ $x(yz) = (xy)z$	Associative laws
$x + yz = (x + y)(x + z)$ $x(y + z) = xy + xz$	Distributive laws
$\overline{(xy)} = \overline{x} + \overline{y}$ $\overline{(x + y)} = \overline{x}\,\overline{y}$	De Morgan's laws
$x + xy = x$ $x(x + y) = x$	Absorption laws
$x + \overline{x} = 1$	Unit property
$x\overline{x} = 0$	Zero property

Compare these Boolean identities with the logical equivalences in Section 1.3 and the set identities in Section 2.2.

Table 5 into a logical equivalence by changing each Boolean variable into a propositional variable, each 0 into a **F**, each 1 into a **T**, each Boolean sum into a disjunction, each Boolean product into a conjunction, and each complementation into a negation, as we illustrate in Example 9.

EXAMPLE 9 Translate the distributive law $x + yz = (x + y)(x + z)$ in Table 5 into a logical equivalence.

Solution: To translate a Boolean identity into a logical equivalence, we change each Boolean variable into a propositional variable. Here we will change the Boolean variables x, y, and z into the propositional variables p, q, and r. Next, we change each Boolean sum into a disjunction and

TABLE 6 Verifying One of the Distributive Laws.

x	y	z	$y + z$	xy	xz	$x(y + z)$	$xy + xz$
1	1	1	1	1	1	1	1
1	1	0	1	1	0	1	1
1	0	1	1	0	1	1	1
1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0

each Boolean product into a conjunction. (Note that 0 and 1 do not appear in this identity and complementation also does not appear.) This transforms the Boolean identity into the logical equivalence

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r).$$

This logical equivalence is one of the distributive laws for propositional logic in Table 6 in Section 1.3. ◀

Identities in Boolean algebra can be used to prove further identities. We demonstrate this in Example 10.

EXAMPLE 10 Prove the **absorption law** $x(x + y) = x$ using the other identities of Boolean algebra shown in Table 5. (This is called an absorption law because absorbing $x + y$ into x leaves x unchanged.)

Extra Examples ❯

Solution: We display steps used to derive this identity and the law used in each step:

$x(x + y) = (x + 0)(x + y)$	Identity law for the Boolean sum
$= x + 0 \cdot y$	Distributive law of the Boolean sum over the Boolean product
$= x + y \cdot 0$	Commutative law for the Boolean product
$= x + 0$	Domination law for the Boolean product
$= x$	Identity law for the Boolean sum. ◀

12.1.4 Duality

The identities in Table 5 come in pairs (except for the law of the double complement and the unit and zero properties). To explain the relationship between the two identities in each pair we use the concept of a dual. The **dual** of a Boolean expression is obtained by interchanging Boolean sums and Boolean products and interchanging 0s and 1s.

EXAMPLE 11 Find the duals of $x(y + 0)$ and $\bar{x} \cdot 1 + (\bar{y} + z)$.

Solution: Interchanging $\cdot$ signs and $+$ signs and interchanging 0s and 1s in these expressions produces their duals. The duals are $x + (y \cdot 1)$ and $(\bar{x} + 0)(\bar{y}z)$, respectively. ◀

The dual of a Boolean function F represented by a Boolean expression is the function represented by the dual of this expression. This dual function, denoted by F^d, does not depend on the particular Boolean expression used to represent F. An identity between functions represented by Boolean expressions remains valid when the duals of both sides of the identity are taken. (See Exercise 30 for the reason why this is true.) This result, called the **duality principle**, is useful for obtaining new identities.

EXAMPLE 12 Construct an identity from the absorption law $x(x + y) = x$ by taking duals.

Solution: Taking the duals of both sides of this identity produces the identity $x + xy = x$, which is also called an absorption law and is shown in Table 5. ◀

12.1.5 The Abstract Definition of a Boolean Algebra

In this section we have focused on Boolean functions and expressions. However, the results we have established can be translated into results about propositions or results about sets. Because of this, it is useful to define Boolean algebras abstractly. Once it is shown that a particular structure is a Boolean algebra, then all results established about Boolean algebras in general apply to this particular structure.

Boolean algebras can be defined in several ways. The most common way is to specify the properties that operations must satisfy, as is done in Definition 1.

Definition 1 A *Boolean algebra* is a set B with two binary operations $\vee$ and $\wedge$, elements 0 and 1, and a unary operation $^-$ such that these properties hold for all x, y, and z in B:

$x \vee 0 = x$ $x \wedge 1 = x$	Identity laws
$x \vee \bar{x} = 1$ $x \wedge \bar{x} = 0$	Complement laws
$(x \vee y) \vee z = x \vee (y \vee z)$ $(x \wedge y) \wedge z = x \wedge (y \wedge z)$	Associative laws
$x \vee y = y \vee x$ $x \wedge y = y \wedge x$	Commutative laws
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	Distributive laws

Using the laws given in Definition 1, it is possible to prove many other laws that hold for every Boolean algebra, such as idempotent and domination laws. (See Exercises 35–42.)

From our previous discussion, $B = \{0, 1\}$ with the *OR* and *AND* operations and the complement operator, satisfies all these properties. The set of propositions in n variables, with the $\vee$ and $\wedge$ operators, **F** and **T**, and the negation operator, also satisfies all the properties of a Boolean algebra, as can be seen from Table 6 in Section 1.3. Similarly, the set of subsets of a universal set U with the union and intersection operations, the empty set and the universal set, and the set complementation operator, is a Boolean algebra as can be seen by consulting Table 1 in Section 2.2. So, to establish results about each of Boolean expressions, propositions, and sets, we need only prove results about abstract Boolean algebras.

Boolean algebras may also be defined using the notion of a lattice, discussed in Chapter 9. Recall that a lattice L is a partially ordered set in which every pair of elements x, y has a least upper bound, denoted by $\text{lub}(x, y)$ and a greatest lower bound denoted by $\text{glb}(x, y)$. Given two elements x and y of L, we can define two operations $\vee$ and $\wedge$ on pairs of elements of L by $x \vee y = \text{lub}(x, y)$ and $x \wedge y = \text{glb}(x, y)$.

For a lattice L to be a Boolean algebra as specified in Definition 1, it must have two properties. First, it must be **complemented**. For a lattice to be complemented it must have a least element 0 and a greatest element 1 and for every element x of the lattice there must exist an element $\bar{x}$ such that $x \vee \bar{x} = 1$ and $x \wedge \bar{x} = 0$. Second, it must be **distributive**. This means that for every x, y, and z in L, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$. Showing that a complemented, distributive lattice is a Boolean algebra has been left as Supplementary Exercise 39 in Chapter 9.

Exercises

1. Find the values of these expressions.
a) $1 \cdot \overline{0}$ **b)** $1 + \overline{1}$ **c)** $\overline{0} \cdot 0$ **d)** $\overline{(1 + 0)}$

2. Find the values, if any, of the Boolean variable x that satisfy these equations.
a) $x \cdot 1 = 0$ **b)** $x + x = 0$
c) $x \cdot 1 = x$ **d)** $x \cdot \overline{x} = 1$

3. **a)** Show that $(1 \cdot 1) + (\overline{0 \cdot 1} + 0) = 1$.
b) Translate the equation in part (a) into a propositional equivalence by changing each 0 into an **F**, each 1 into a **T**, each Boolean sum into a disjunction, each Boolean product into a conjunction, each complementation into a negation, and the equals sign into a propositional equivalence sign.

4. **a)** Show that $(\overline{1} \cdot \overline{0}) + (1 \cdot \overline{0}) = 1$.
b) Translate the equation in part (a) into a propositional equivalence by changing each 0 into an **F**, each 1 into a **T**, each Boolean sum into a disjunction, each Boolean product into a conjunction, each complementation into a negation, and the equals sign into a propositional equivalence sign.

5. Use a table to express the values of each of these Boolean functions.
a) $F(x, y, z) = \overline{x}y$
b) $F(x, y, z) = x + yz$
c) $F(x, y, z) = x\overline{y} + \overline{(xyz)}$
d) $F(x, y, z) = x(yz + \overline{y}\,\overline{z})$

6. Use a table to express the values of each of these Boolean functions.
a) $F(x, y, z) = \overline{z}$
b) $F(x, y, z) = \overline{x}y + \overline{y}z$
c) $F(x, y, z) = x\overline{y}z + \overline{(xyz)}$
d) $F(x, y, z) = \overline{y}(xz + \overline{x}\,\overline{z})$

7. Use a 3-cube Q_3 to represent each of the Boolean functions in Exercise 5 by displaying a black circle at each vertex that corresponds to a 3-tuple where this function has the value 1.

8. Use a 3-cube Q_3 to represent each of the Boolean functions in Exercise 6 by displaying a black circle at each vertex that corresponds to a 3-tuple where this function has the value 1.

9. What values of the Boolean variables x and y satisfy $xy = x + y$?

10. How many different Boolean functions are there of degree 7?

11. Prove the absorption law $x + xy = x$ using the other laws in Table 5.

12. Show that $F(x, y, z) = xy + xz + yz$ has the value 1 if and only if at least two of the variables x, y, and z have the value 1.

13. Show that $x\overline{y} + y\overline{z} + \overline{x}z = \overline{x}y + \overline{y}z + x\overline{z}$.

Exercises 14–23 deal with the Boolean algebra $\{0, 1\}$ with addition, multiplication, and complement defined at the beginning of this section. In each case, use a table as in Example 8.

14. Verify the law of the double complement.

15. Verify the idempotent laws.

16. Verify the identity laws.

17. Verify the domination laws.

18. Verify the commutative laws.

19. Verify the associative laws.

20. Verify the first distributive law in Table 5.

21. Verify De Morgan's laws.

22. Verify the unit property.

23. Verify the zero property.

The Boolean operator $\oplus$, called the *XOR* operator, is defined by $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, and $0 \oplus 0 = 0$.

24. Simplify these expressions.
a) $x \oplus 0$ **b)** $x \oplus 1$
c) $x \oplus x$ **d)** $x \oplus \overline{x}$

25. Show that these identities hold.
a) $x \oplus y = (x + y)\overline{(xy)}$
b) $x \oplus y = (x\overline{y}) + (\overline{x}y)$

26. Show that $x \oplus y = y \oplus x$.

27. Prove or disprove these equalities.
a) $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
b) $x + (y \oplus z) = (x + y) \oplus (x + z)$
c) $x \oplus (y + z) = (x \oplus y) + (x \oplus z)$

28. Find the duals of these Boolean expressions.
a) $x + y$ **b)** $\overline{x}\,\overline{y}$
c) $xyz + \overline{x}\,\overline{y}\,\overline{z}$ **d)** $x\overline{z} + x \cdot 0 + \overline{x} \cdot 1$

***29.** Suppose that F is a Boolean function represented by a Boolean expression in the variables $x_1, \ldots, x_n$. Show that $F^d(x_1, \ldots, x_n) = \overline{F(\overline{x}_1, \ldots, \overline{x}_n)}$.

***30.** Show that if F and G are Boolean functions represented by Boolean expressions in n variables and $F = G$, then $F^d = G^d$, where F^d and G^d are the Boolean functions represented by the duals of the Boolean expressions representing F and G, respectively. [*Hint:* Use the result of Exercise 29.]

***31.** How many different Boolean functions $F(x, y, z)$ are there such that $F(\overline{x}, \overline{y}, \overline{z}) = F(x, y, z)$ for all values of the Boolean variables x, y, and z?

***32.** How many different Boolean functions $F(x, y, z)$ are there such that $F(\overline{x}, y, z) = F(x, \overline{y}, z) = F(x, y, \overline{z})$ for all values of the Boolean variables x, y, and z?

33. Show that you obtain De Morgan's laws for propositions (in Table 6 in Section 1.3) when you transform De Morgan's laws for Boolean algebra in Table 6 into logical equivalences.

34. Show that you obtain the absorption laws for propositions (in Table 6 in Section 1.3) when you transform the absorption laws for Boolean algebra in Table 6 into logical equivalences.

In Exercises 35–42, use the laws in Definition 1 to show that the stated properties hold in every Boolean algebra.

35. Show that in a Boolean algebra, the **idempotent laws** $x \vee x = x$ and $x \wedge x = x$ hold for every element x.

36. Show that in a Boolean algebra, every element x has a unique complement $\bar{x}$ such that $x \vee \bar{x} = 1$ and $x \wedge \bar{x} = 0$.

37. Show that in a Boolean algebra, the complement of the element 0 is the element 1 and vice versa.

38. Prove that in a Boolean algebra, the **law of the double complement** holds; that is, $\bar{\bar{x}} = x$ for every element x.

39. Show that **De Morgan's laws** hold in a Boolean algebra. That is, show that for all x and y, $\overline{(x \vee y)} = \bar{x} \wedge \bar{y}$ and $\overline{(x \wedge y)} = \bar{x} \vee \bar{y}$.

40. Show that in a Boolean algebra, the **modular properties** hold. That is, show that $x \wedge (y \vee (x \wedge z)) = (x \wedge y) \vee (x \wedge z)$ and $x \vee (y \wedge (x \vee z)) = (x \vee y) \wedge (x \vee z)$.

41. Show that in a Boolean algebra, if $x \vee y = 0$, then $x = 0$ and $y = 0$, and that if $x \wedge y = 1$, then $x = 1$ and $y = 1$.

42. Show that in a Boolean algebra, the **dual** of an identity, obtained by interchanging the $\vee$ and $\wedge$ operators and interchanging the elements 0 and 1, is also a valid identity.

43. Show that a complemented, distributive lattice is a Boolean algebra.

12.2 Representing Boolean Functions

Two important problems of Boolean algebra will be studied in this section. The first problem is: Given the values of a Boolean function, how can a Boolean expression that represents this function be found? This problem will be solved by showing that any Boolean function can be represented by a Boolean sum of Boolean products of the variables and their complements. The solution of this problem shows that every Boolean function can be represented using the three Boolean operators $\cdot$, $+$, and $\bar{}$. The second problem is: Is there a smaller set of operators that can be used to represent all Boolean functions? We will answer this question by showing that all Boolean functions can be represented using only one operator. Both of these problems have practical importance in circuit design.

12.2.1 Sum-of-Products Expansions

We will use examples to illustrate one important way to find a Boolean expression that represents a Boolean function.

EXAMPLE 1 Find Boolean expressions that represent the functions $F(x, y, z)$ and $G(x, y, z)$, which are given in Table 1.

TABLE 1

x	y	z	F	G
1	1	1	0	0
1	1	0	0	1
1	0	1	1	0
1	0	0	0	0
0	1	1	0	0
0	1	0	0	1
0	0	1	0	0
0	0	0	0	0

Solution: An expression that has the value 1 when $x = z = 1$ and $y = 0$, and the value 0 otherwise, is needed to represent F. Such an expression can be formed by taking the Boolean product of x, $\bar{y}$, and z. This product, $x\bar{y}z$, has the value 1 if and only if $x = \bar{y} = z = 1$, which holds if and only if $x = z = 1$ and $y = 0$.

To represent G, we need an expression that equals 1 when $x = y = 1$ and $z = 0$, or $x = z = 0$ and $y = 1$. We can form an expression with these values by taking the Boolean sum of two different Boolean products. The Boolean product $xy\bar{z}$ has the value 1 if and only if $x = y = 1$ and $z = 0$. Similarly, the product $\bar{x}y\bar{z}$ has the value 1 if and only if $x = z = 0$ and $y = 1$. The Boolean sum of these two products, $xy\bar{z} + \bar{x}y\bar{z}$, represents G, because it has the value 1 if and only if $x = y = 1$ and $z = 0$, or $x = z = 0$ and $y = 1$. ◀

Example 1 illustrates a procedure for constructing a Boolean expression representing a function with given values. Each combination of values of the variables for which the function has the value 1 leads to a Boolean product of the variables or their complements.

Definition 1 A *literal* is a Boolean variable or its complement. A *minterm* of the Boolean variables $x_1, x_2, \ldots, x_n$ is a Boolean product $y_1 y_2 \cdots y_n$, where $y_i = x_i$ or $y_i = \bar{x}_i$. Hence, a minterm is a product of n literals, with one literal for each variable.

A minterm has the value 1 for one and only one combination of values of its variables. More precisely, the minterm $y_1 y_2 \ldots y_n$ is 1 if and only if each y_i is 1, and this occurs if and only if $x_i = 1$ when $y_i = x_i$ and $x_i = 0$ when $y_i = \bar{x}_i$.

EXAMPLE 2 Find a minterm that equals 1 if $x_1 = x_3 = 0$ and $x_2 = x_4 = x_5 = 1$, and equals 0 otherwise.

Solution: The minterm $\bar{x}_1 x_2 \bar{x}_3 x_4 x_5$ has the correct set of values. ◀

By taking Boolean sums of distinct minterms we can build up a Boolean expression with a specified set of values. In particular, a Boolean sum of minterms has the value 1 when exactly one of the minterms in the sum has the value 1. It has the value 0 for all other combinations of values of the variables. Consequently, given a Boolean function, a Boolean sum of minterms can be formed that has the value 1 when this Boolean function has the value 1, and has the value 0 when the function has the value 0. The minterms in this Boolean sum correspond to those combinations of values for which the function has the value 1. The sum of minterms that represents the function is called the **sum-of-products expansion** or the **disjunctive normal form** of the Boolean function.

Links

(See Exercise 46 in Section 1.3 for the development of disjunctive normal form in propositional calculus.)

EXAMPLE 3 Find the sum-of-products expansion for the function $F(x, y, z) = (x + y)\bar{z}$.

Extra Examples

Solution: We will find the sum-of-products expansion of $F(x, y, z)$ in two ways. First, we will use Boolean identities to expand the product and simplify. We find that

$$\begin{aligned} F(x, y, z) &= (x + y)\bar{z} \\ &= x\bar{z} + y\bar{z} && \text{Distributive law} \\ &= x1\bar{z} + 1y\bar{z} && \text{Identity law} \\ &= x(y + \bar{y})\bar{z} + (x + \bar{x})y\bar{z} && \text{Unit property} \\ &= xy\bar{z} + x\bar{y}\,\bar{z} + xy\bar{z} + \bar{x}y\bar{z} && \text{Distributive law} \\ &= xy\bar{z} + x\bar{y}\,\bar{z} + \bar{x}y\,\bar{z}. && \text{Idempotent law} \end{aligned}$$

Second, we can construct the sum-of-products expansion by determining the values of F for all possible values of the variables x, y, and z. These values are found in Table 2. The sum-of-products expansion of F is the Boolean sum of three minterms corresponding to the three rows of this table that give the value 1 for the function. This gives

$$F(x, y, z) = xy\bar{z} + x\bar{y}\,\bar{z} + \bar{x}y\bar{z}.$$ ◀

It is also possible to find a Boolean expression that represents a Boolean function by taking a Boolean product of Boolean sums. The resulting expansion is called the **conjunctive normal form** or **product-of-sums expansion** of the function. These expansions can be found from sum-of-products expansions by taking duals. How to find such expansions directly is described in Exercise 10.

TABLE 2

x	y	z	$x + y$	$\bar{z}$	$(x + y)\bar{z}$
1	1	1	1	0	0
1	1	0	1	1	1
1	0	1	1	0	0
1	0	0	1	1	1
0	1	1	1	0	0
0	1	0	1	1	1
0	0	1	0	0	0
0	0	0	0	1	0

12.2.2 Functional Completeness

Every Boolean function can be expressed as a Boolean sum of minterms. Each minterm is the Boolean product of Boolean variables or their complements. This shows that every Boolean function can be represented using the Boolean operators $\cdot$, $+$, and $^-$. Because every Boolean function can be represented using these operators we say that the set $\{\cdot, +, ^-\}$ is **functionally complete**. Can we find a smaller set of functionally complete operators? We can do so if one of the three operators of this set can be expressed in terms of the other two. This can be done using one of De Morgan's laws. We can eliminate all Boolean sums using the identity

$$x + y = \overline{\bar{x}\,\bar{y}},$$

which is obtained by taking complements of both sides in the second De Morgan law, given in Table 5 in Section 12.1, and then applying the double complementation law. This means that the set $\{\cdot, ^-\}$ is functionally complete. Similarly, we could eliminate all Boolean products using the identity

$$xy = \overline{\bar{x} + \bar{y}},$$

which is obtained by taking complements of both sides in the first De Morgan law, given in Table 5 in Section 12.1, and then applying the double complementation law. Consequently $\{+, ^-\}$ is functionally complete. Note that the set $\{+, \cdot\}$ is not functionally complete, because it is impossible to express the Boolean function $F(x) = \bar{x}$ using these operators (see Exercise 19).

We have found sets containing two operators that are functionally complete. Can we find a smaller set of functionally complete operators, namely, a set containing just one operator? Such sets exist. Define two operators, the $|$ or ***NAND*** operator, defined by $1 \mid 1 = 0$ and $1 \mid 0 = 0 \mid 1 = 0 \mid 0 = 1$; and the $\downarrow$ or ***NOR*** operator, defined by $1 \downarrow 1 = 1 \downarrow 0 = 0 \downarrow 1 = 0$ and $0 \downarrow 0 = 1$. Both of the sets $\{\mid\}$ and $\{\downarrow\}$ are functionally complete. To see that $\{\mid\}$ is functionally complete, because $\{\cdot, ^-\}$ is functionally complete, all that we have to do is show that both of the operators $\cdot$ and $^-$ can be expressed using just the $|$ operator. This can be done as

$$\bar{x} = x \mid x,$$
$$xy = (x \mid y) \mid (x \mid y).$$

The reader should verify these identities (see Exercise 14). We leave the demonstration that $\{\downarrow\}$ is functionally complete for the reader (see Exercises 15 and 16).

Exercises

1. Find a Boolean product of the Boolean variables x, y, and z, or their complements, that has the value 1 if and only if
   a) $x = y = 0, z = 1$. **b)** $x = 0, y = 1, z = 0$.
   c) $x = 0, y = z = 1$. **d)** $x = y = z = 0$.
2. Find the sum-of-products expansions of these Boolean functions.
   a) $F(x, y) = \bar{x} + y$ **b)** $F(x, y) = x\bar{y}$
   c) $F(x, y) = 1$ **d)** $F(x, y) = \bar{y}$
3. Find the sum-of-products expansions of these Boolean functions.
   a) $F(x, y, z) = x + y + z$
   b) $F(x, y, z) = (x + z)y$
   c) $F(x, y, z) = x$
   d) $F(x, y, z) = x\bar{y}$
4. Find the sum-of-products expansions of the Boolean function $F(x, y, z)$ that equals 1 if and only if
   a) $x = 0$. **b)** $xy = 0$.
   c) $x + y = 0$. **d)** $xyz = 0$.
5. Find the sum-of-products expansion of the Boolean function $F(w, x, y, z)$ that has the value 1 if and only if an odd number of $w, x, y,$ and z have the value 1.
6. Find the sum-of-products expansion of the Boolean function $F(x_1, x_2, x_3, x_4, x_5)$ that has the value 1 if and only if three or more of the variables x_1, x_2, x_3, x_4, and x_5 have the value 1.

Another way to find a Boolean expression that represents a Boolean function is to form a Boolean product of Boolean sums of literals. Exercises 7–11 are concerned with representations of this kind.

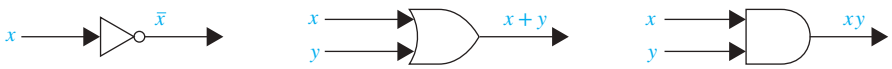
7. Find a Boolean sum containing either x or $\bar{x}$, either y or $\bar{y}$, and either z or $\bar{z}$ that has the value 0 if and only if
   a) $x = y = 1, z = 0$. **b)** $x = y = z = 0$.
   c) $x = z = 0, y = 1$.
8. Find a Boolean product of Boolean sums of literals that has the value 0 if and only if $x = y = 1$ and $z = 0, x = z = 0$ and $y = 1$, or $x = y = z = 0$. [*Hint:* Take the Boolean product of the Boolean sums found in parts (a), (b), and (c) in Exercise 7.]
9. Show that the Boolean sum $y_1 + y_2 + \cdots + y_n$, where $y_i = x_i$ or $y_i = \bar{x}_i$, has the value 0 for exactly one combination of the values of the variables, namely, when $x_i = 0$ if $y_i = x_i$ and $x_i = 1$ if $y_i = \bar{x}_i$. This Boolean sum is called a **maxterm**.
10. Show that a Boolean function can be represented as a Boolean product of maxterms. This representation is called the **product-of-sums expansion** or **conjunctive normal form** of the function. [*Hint:* Include one maxterm in this product for each combination of the variables where the function has the value 0.]
11. Find the product-of-sums expansion of each of the Boolean functions in Exercise 3.
12. Express each of these Boolean functions using the operators $\cdot$ and $^-$.
    a) $x + y + z$ **b)** $x + \bar{y}(\bar{x} + z)$
    c) $\overline{x + \bar{y}}$ **d)** $\bar{x}(x + \bar{y} + \bar{z})$
13. Express each of the Boolean functions in Exercise 12 using the operators $+$ and $^-$.
14. Show that
    a) $\bar{x} = x \mid x$. **b)** $xy = (x \mid y) \mid (x \mid y)$.
    c) $x + y = (x \mid x) \mid (y \mid y)$.
15. Show that
    a) $\bar{x} = x \downarrow x$.
    b) $xy = (x \downarrow x) \downarrow (y \downarrow y)$.
    c) $x + y = (x \downarrow y) \downarrow (x \downarrow y)$.
16. Show that $\{ \downarrow \}$ is functionally complete using Exercise 15.
17. Express each of the Boolean functions in Exercise 3 using the operator $\mid$.
18. Express each of the Boolean functions in Exercise 3 using the operator $\downarrow$.
19. Show that the set of operators $\{+, \cdot\}$ is not functionally complete.
20. Are these sets of operators functionally complete?
    a) $\{+, \oplus\}$ **b)** $\{^-, \oplus\}$ **c)** $\{\cdot, \oplus\}$

12.3 Logic Gates

12.3.1 Introduction

Links

Boolean algebra is used to model the circuitry of electronic devices. Each input and each output of such a device can be thought of as a member of the set $\{0, 1\}$. A computer, or other electronic device, is made up of a number of circuits. Each circuit can be designed using the rules of Boolean algebra that were studied in Sections 12.1 and 12.2. The basic elements of circuits are called **gates**, and were introduced in Section 1.2. Each type of gate implements a Boolean operation. In this section we define several types of gates. Using these gates, we will apply the



(a) Inverter (b) OR gate (c) AND gate

FIGURE 1 **Basic types of gates.**

rules of Boolean algebra to design circuits that perform a variety of tasks. The circuits that we will study in this chapter give output that depends only on the input, and not on the current state of the circuit. In other words, these circuits have no memory capabilities. Such circuits are called **combinational circuits** or **gating networks**.

We will construct combinational circuits using three types of elements. The first is an **inverter**, which accepts the value of one Boolean variable as input and produces the complement of this value as its output. The symbol used for an inverter is shown in Figure 1(a). The input to the inverter is shown on the left side entering the element, and the output is shown on the right side leaving the element.

The next type of element we will use is the **OR gate**. The inputs to this gate are the values of two or more Boolean variables. The output is the Boolean sum of their values. The symbol used for an OR gate is shown in Figure 1(b). The inputs to the OR gate are shown on the left side entering the element, and the output is shown on the right side leaving the element.

The third type of element we will use is the **AND gate**. The inputs to this gate are the values of two or more Boolean variables. The output is the Boolean product of their values. The symbol used for an AND gate is shown in Figure 1(c). The inputs to the AND gate are shown on the left side entering the element, and the output is shown on the right side leaving the element.

We will permit multiple inputs to AND and OR gates. The inputs to each of these gates are shown on the left side entering the element, and the output is shown on the right side. Examples of AND and OR gates with n inputs are shown in Figure 2.

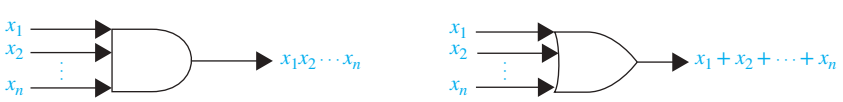


FIGURE 2 **Gates with n inputs.**

12.3.2 Combinations of Gates

Combinational circuits can be constructed using a combination of inverters, OR gates, and AND gates. When combinations of circuits are formed, some gates may share inputs. This is shown in one of two ways in depictions of circuits. One method is to use branchings that indicate all the gates that use a given input. The other method is to indicate this input separately for each gate. Figure 3 illustrates the two ways of showing gates with the same input values. Note also that output from a gate may be used as input by one or more other elements, as shown in Figure 3. Both drawings in Figure 3 depict the circuit that produces the output $xy + \bar{x}y$.

EXAMPLE 1 Construct circuits that produce the following outputs: (a) $(x + y)\bar{x}$, (b) $\bar{x}\,\overline{(y + \bar{z})}$, and (c) $(x + y + z)(\bar{x}\,\bar{y}\,\bar{z})$.

Solution: Circuits that produce these outputs are shown in Figure 4. ◀

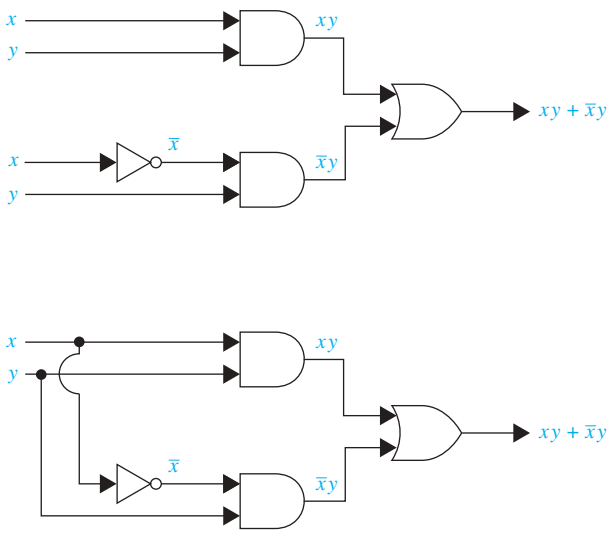


FIGURE 3 Two ways to draw the same circuit.

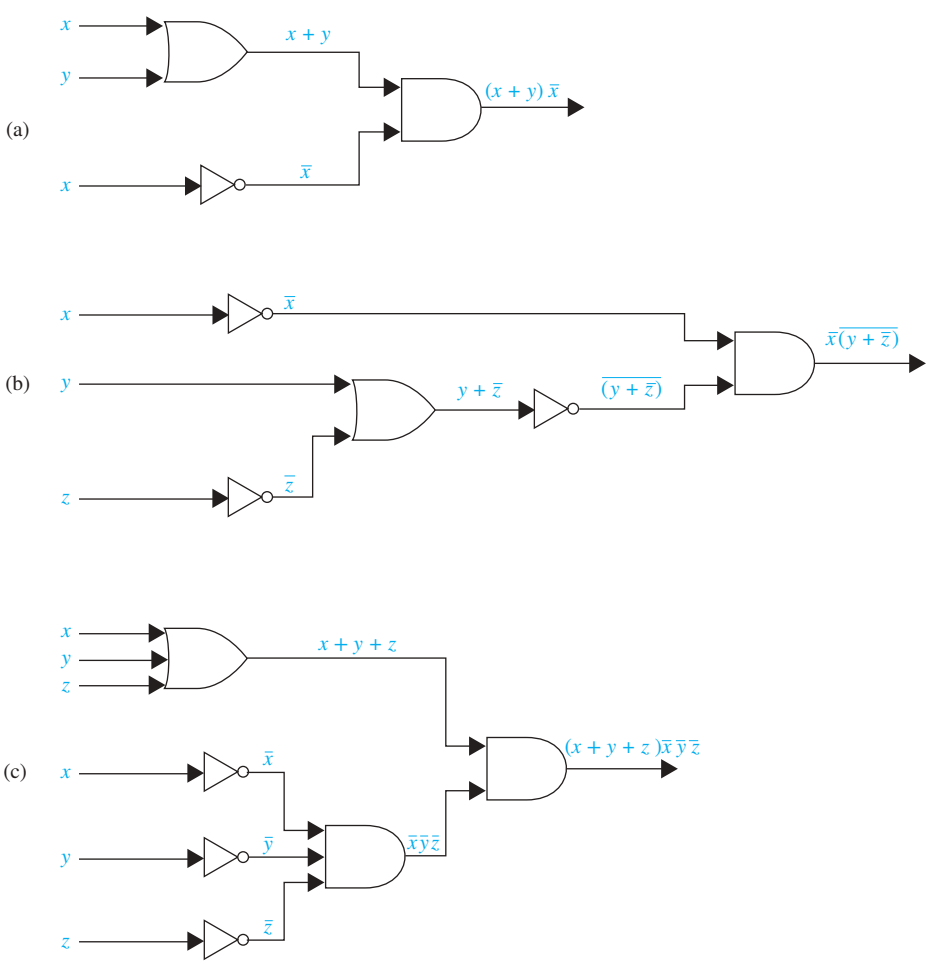


FIGURE 4 Circuits that produce the outputs specified in Example 1.

12.3.3 Examples of Circuits

We will give some examples of circuits that perform some useful functions.

EXAMPLE 2 A committee of three individuals decides issues for an organization. Each individual votes either yes or no for each proposal that arises. A proposal is passed if it receives at least two yes votes. Design a circuit that determines whether a proposal passes.

Extra Examples

Solution: Let $x = 1$ if the first individual votes yes, and $x = 0$ if this individual votes no; let $y = 1$ if the second individual votes yes, and $y = 0$ if this individual votes no; let $z = 1$ if the third individual votes yes, and $z = 0$ if this individual votes no. Then a circuit must be designed that produces the output 1 from the inputs x, y, and z when two or more of x, y, and z are 1. One representation of the Boolean function that has these output values is $xy + xz + yz$ (see Exercise 12 in Section 12.1). The circuit that implements this function is shown in Figure 5. ◀

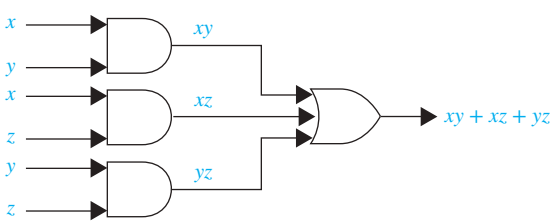


FIGURE 5 **A circuit for majority voting.**

EXAMPLE 3 Sometimes light fixtures are controlled by more than one switch. Circuits need to be designed so that flipping any one of the switches for the fixture turns the light on when it is off and turns the light off when it is on. Design circuits that accomplish this when there are two switches and when there are three switches.

TABLE 1

x	y	$F(x, y)$
1	1	1
1	0	0
0	1	0
0	0	1

Solution: We will begin by designing the circuit that controls the light fixture when two different switches are used. Let $x = 1$ when the first switch is closed and $x = 0$ when it is open, and let $y = 1$ when the second switch is closed and $y = 0$ when it is open. Let $F(x, y) = 1$ when the light is on and $F(x, y) = 0$ when it is off. We can arbitrarily decide that the light will be on when both switches are closed, so that $F(1, 1) = 1$. This determines all the other values of F. When one of the two switches is opened, the light goes off, so $F(1, 0) = F(0, 1) = 0$. When the other switch is also opened, the light goes on, so $F(0, 0) = 1$. Table 1 displays these values. Note that $F(x, y) = xy + \bar{x}\,\bar{y}$. This function is implemented by the circuit shown in Figure 6.

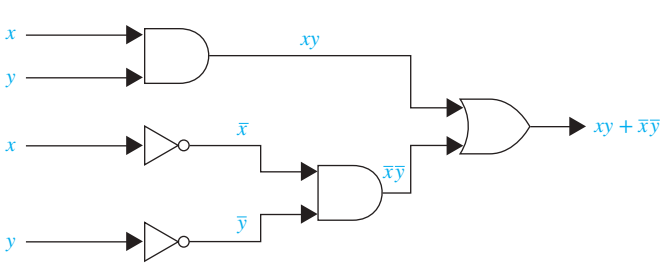


FIGURE 6 **A circuit for a light controlled by two switches.**

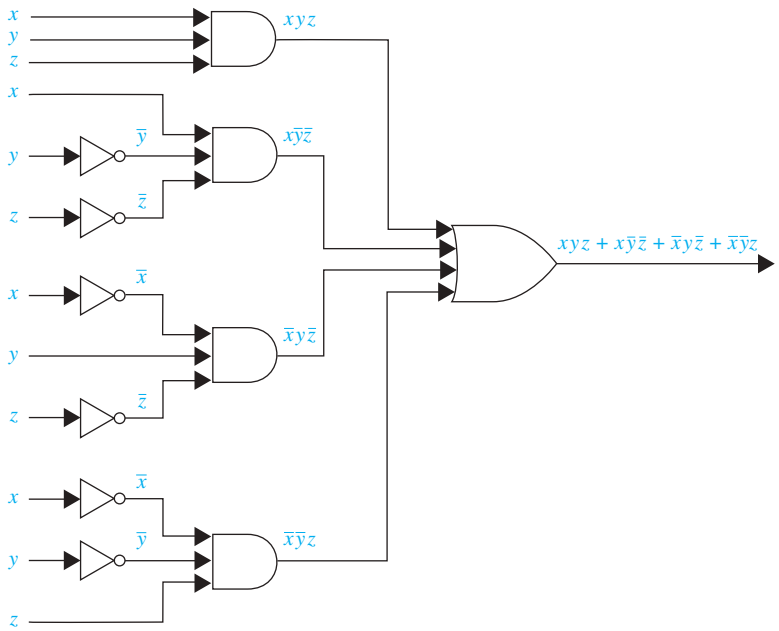


FIGURE 7 **A circuit for a fixture controlled by three switches.**

TABLE 2

x	y	z	$F(x, y, z)$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

We will now design a circuit for three switches. Let x, y, and z be the Boolean variables that indicate whether each of the three switches is closed. We let $x = 1$ when the first switch is closed, and $x = 0$ when it is open; $y = 1$ when the second switch is closed, and $y = 0$ when it is open; and $z = 1$ when the third switch is closed, and $z = 0$ when it is open. Let $F(x, y, z) = 1$ when the light is on and $F(x, y, z) = 0$ when the light is off. We can arbitrarily specify that the light be on when all three switches are closed, so that $F(1, 1, 1) = 1$. This determines all other values of F. When one switch is opened, the light goes off, so $F(1, 1, 0) = F(1, 0, 1) = F(0, 1, 1) = 0$. When a second switch is opened, the light goes on, so $F(1, 0, 0) = F(0, 1, 0) = F(0, 0, 1) = 1$. Finally, when the third switch is opened, the light goes off again, so $F(0, 0, 0) = 0$. Table 2 shows the values of this function.

The function F can be represented by its sum-of-products expansion as $F(x, y, z) = xyz + x\bar{y}\,\bar{z} + \bar{x}yz + \bar{x}\,\bar{y}z$. The circuit shown in Figure 7 implements this function. ◀

Links ❯

12.3.4 Adders

TABLE 3
Input and Output for the Half Adder.

Input		*Output*	
x	y	s	c
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

We will illustrate how logic circuits can be used to carry out addition of two positive integers from their binary expansions. We will build up the circuitry to do this addition from some component circuits. First, we will build a circuit that can be used to find $x + y$, where x and y are two bits. The input to our circuit will be x and y, because these each have the value 0 or the value 1. The output will consist of two bits, namely, s and c, where s is the sum bit and c is the carry bit. This circuit is called a **multiple output circuit** because it has more than one output. The circuit that we are designing is called the **half adder**, because it adds two bits, without considering a carry from a previous addition. We show the input and output for the half adder in Table 3. From Table 3 we see that $c = xy$ and that $s = x\bar{y} + \bar{x}y = (x + y)\overline{(xy)}$. Hence, the circuit shown in Figure 8 computes the sum bit s and the carry bit c from the bits x and y.

We use the **full adder** to compute the sum bit and the carry bit when two bits and a carry are added. The inputs to the full adder are the bits x and y and the carry c_i. The outputs are the sum bit s and the new carry c_{i+1}. The inputs and outputs for the full adder are shown in Table 4.

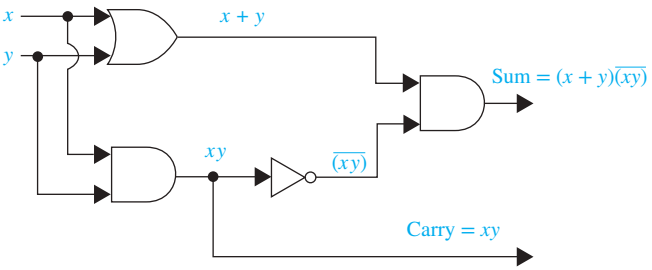


FIGURE 8 **The half adder.**

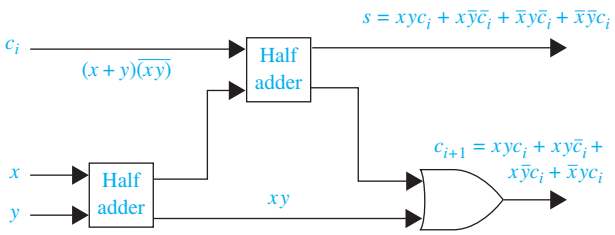


FIGURE 9 **A full adder.**

TABLE 4 Input and Output for the Full Adder.

Input			*Output*	
x	y	c_i	s	c_{i+1}
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

The two outputs of the full adder, the sum bit s and the carry c_{i+1}, are given by the sum-of-products expansions $xyc_i + x\bar{y}\,\bar{c}_i + \bar{x}y\bar{c}_i + \bar{x}\,\bar{y}c_i$ and $xyc_i + xy\bar{c}_i + x\bar{y}c_i + \bar{x}yc_i$, respectively. However, instead of designing the full adder from scratch, we will use half adders to produce the desired output. A full adder circuit using half adders is shown in Figure 9.

Finally, in Figure 10 we show how full and half adders can be used to add the two three-bit integers $(x_2x_1x_0)_2$ and $(y_2y_1y_0)_2$ to produce the sum $(s_3s_2s_1s_0)_2$. Note that s_3, the highest-order bit in the sum, is given by the carry c_2.

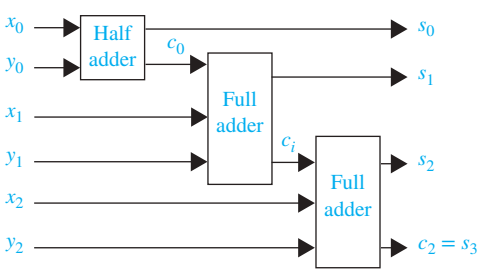
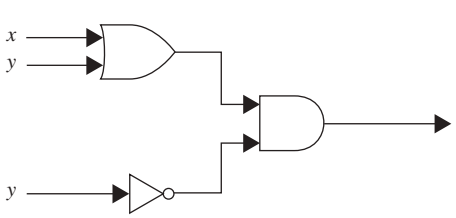


FIGURE 10 **Adding two three-bit integers with full and half adders.**

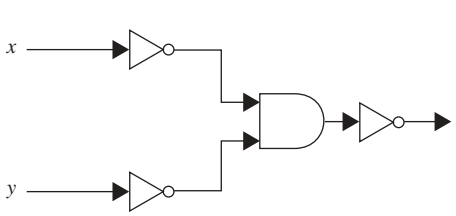
Exercises

In Exercises 1–5 find the output of the given circuit.

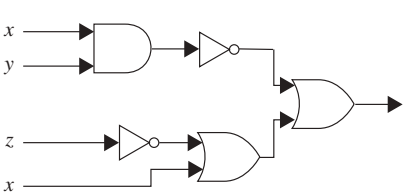
1.



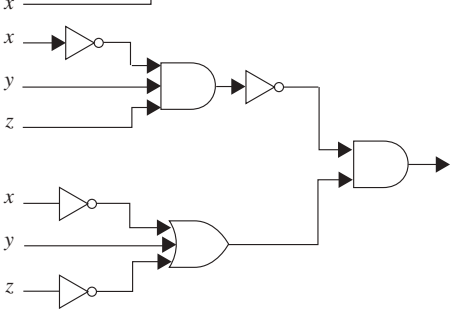
2.



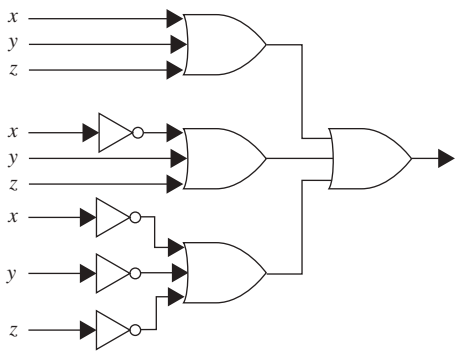
3.



4.



5.



6. Construct circuits from inverters, AND gates, and OR gates to produce these outputs.

a) $\bar{x} + y$ **b)** $\overline{(x + y)}x$

c) $xyz + \bar{x}\bar{y}\bar{z}$ **d)** $\overline{(\bar{x} + z)(y + \bar{z})}$

7. Design a circuit that implements majority voting for five individuals.

8. Design a circuit for a light fixture controlled by four switches, where flipping one of the switches turns the light on when it is off and turns it off when it is on.

9. Show how the sum of two five-bit integers can be found using full and half adders.

10. Construct a circuit for a half subtractor using AND gates, OR gates, and inverters. A **half subtractor** has two bits as input and produces as output a difference bit and a borrow.

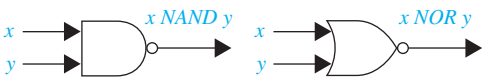
11. Construct a circuit for a full subtractor using AND gates, OR gates, and inverters. A **full subtractor** has two bits and a borrow as input, and produces as output a difference bit and a borrow.

12. Use the circuits from Exercises 10 and 11 to find the difference of two four-bit integers, where the first integer is greater than the second integer.

***13.** Construct a circuit that compares the two-bit integers $(x_1x_0)_2$ and $(y_1y_0)_2$, returning an output of 1 when the first of these numbers is larger and an output of 0 otherwise.

***14.** Construct a circuit that computes the product of the two-bit integers $(x_1x_0)_2$ and $(y_1y_0)_2$. The circuit should have four output bits for the bits in the product.

Two gates that are often used in circuits are NAND and NOR gates. When NAND or NOR gates are used to represent circuits, no other types of gates are needed. The notation for these gates is as follows:



***15.** Use NAND gates to construct circuits with these outputs.

a) $\bar{x}$ **b)** $x + y$

c) xy **d)** $x \oplus y$

***16.** Use NOR gates to construct circuits for the outputs given in Exercise 15.

***17.** Construct a half adder using NAND gates.

***18.** Construct a half adder using NOR gates.

A **multiplexer** is a switching circuit that produces as output one of a set of input bits based on the value of control bits.

19. Construct a multiplexer using AND gates, OR gates, and inverters that has as input the four bits x_0, x_1, x_2, and x_3 and the two control bits c_0 and c_1. Set up the circuit so that x_i is the output, where i is the value of the two-bit integer $(c_1c_0)_2$.

The **depth** of a combinatorial circuit can be defined by specifying that the depth of the initial input is 0 and if a gate has n different inputs at depths $d_1, d_2, \ldots, d_n$, respectively, then its outputs have depth equal to $\max(d_1, d_2, \ldots, d_n) + 1$; this value is also defined to be the depth of the gate. The depth of a combinatorial circuit is the maximum depth of the gates in the circuit.

20. Find the depth of

a) the circuit constructed in Example 2 for majority voting among three people.

b) the circuit constructed in Example 3 for a light controlled by two switches.

c) the half adder shown in Figure 8.

d) the full adder shown in Figure 9.

12.4 Minimization of Circuits

12.4.1 Introduction

The efficiency of a combinational circuit depends on the number and arrangement of its gates. The process of designing a combinational circuit begins with the table specifying the output for each combination of input values. We can always use the sum-of-products expansion of a circuit to find a set of logic gates that will implement this circuit. However, the sum-of-products expansion may contain many more terms than are necessary. Terms in a sum-of-products expansion that differ in just one variable, so that in one term this variable occurs and in the other term the complement of this variable occurs, can be combined. For instance, consider the circuit that has

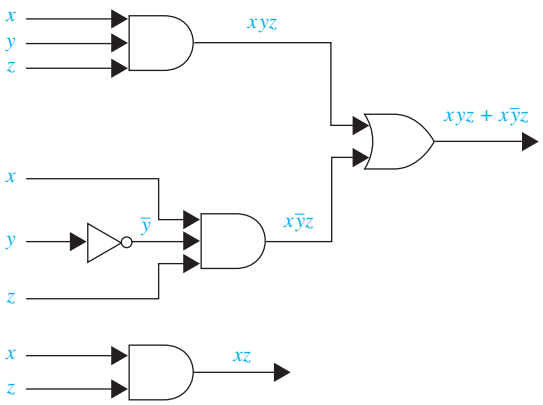


FIGURE 1 **Two circuits with the same output.**

output 1 if and only if $x = y = z = 1$ or $x = z = 1$ and $y = 0$. The sum-of-products expansion of this circuit is $xyz + x\bar{y}z$. The two products in this expansion differ in exactly one variable, namely, y. They can be combined as

$$\begin{aligned} xyz + x\bar{y}z &= (y + \bar{y})(xz) \\ &= 1 \cdot (xz) \\ &= xz. \end{aligned}$$

Hence, xz is a Boolean expression with fewer operators that represents the circuit. We show two different implementations of this circuit in Figure 1. The second circuit uses only one gate, whereas the first circuit uses three gates and an inverter.

This example shows that combining terms in the sum-of-products expansion of a circuit leads to a simpler expression for the circuit. We will describe two procedures that simplify sum-of-products expansions.

The goal of both procedures is to produce Boolean sums of Boolean products that represent a Boolean function with the fewest products of literals such that these products contain the fewest literals possible among all sums of products that represent a Boolean function. Finding such a sum of products is called **minimization of the Boolean function**. Minimizing a Boolean function makes it possible to construct a circuit for this function that uses the fewest gates and fewest inputs to the *AND* gates and *OR* gates in the circuit, among all circuits for the Boolean expression we are minimizing.

Until the early 1960s logic gates were individual components. To reduce costs it was important to use the fewest gates to produce a desired output. However, in the mid-1960s, integrated circuit technology was developed that made it possible to combine gates on a single chip. Even though it is now possible to build increasingly complex integrated circuits on chips at low cost, minimization of Boolean functions remains important.

Reducing the number of gates on a chip can lead to a more reliable circuit and can reduce the cost to produce the chip. Also, minimization makes it possible to fit more circuits on the same chip. Furthermore, minimization reduces the number of inputs to gates in a circuit. This reduces the time used by a circuit to compute its output. Moreover, the number of inputs to a gate may be limited because of the particular technology used to build logic gates.

The first procedure we will introduce, known as Karnaugh maps (or K-maps), was designed in the 1950s to help minimize circuits by hand. K-maps are useful in minimizing circuits with up to six variables, although they become rather complex even for five or six variables. The 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.

12.4.2 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.

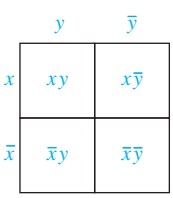


FIGURE 2 **K-maps in two variables.**

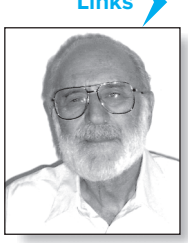
There are four possible minterms in the sum-of-products expansion of a Boolean function in the two variables x and y. A K-map for a Boolean function in these two variables consists of four cells, where a 1 is placed in the cell representing a minterm if this minterm is present in the expansion. Cells 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 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

Links



Courtesy of Maurice Karnaugh

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.

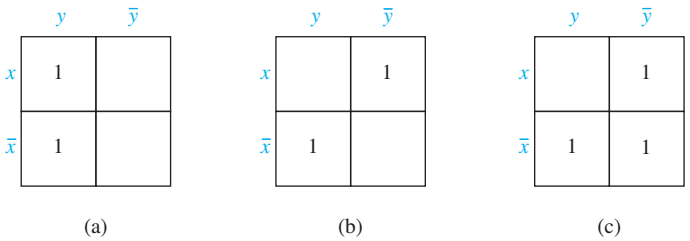


FIGURE 3 **K-maps for the sum-of-products expansions in Example 1.**

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) $x\bar{y} + \bar{x}y$, and (c) $\bar{x} + \bar{y}$. ◀

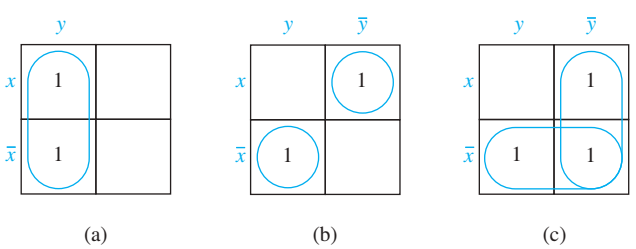


FIGURE 4 **Simplifying the sum-of-products expansions from Example 2.**

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.

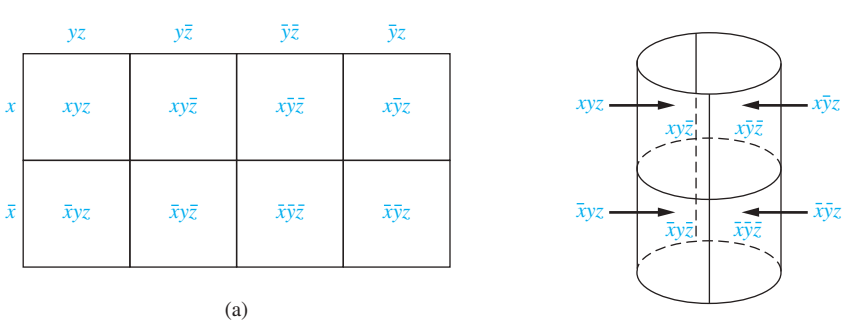


FIGURE 5 **K-maps in three variables.**

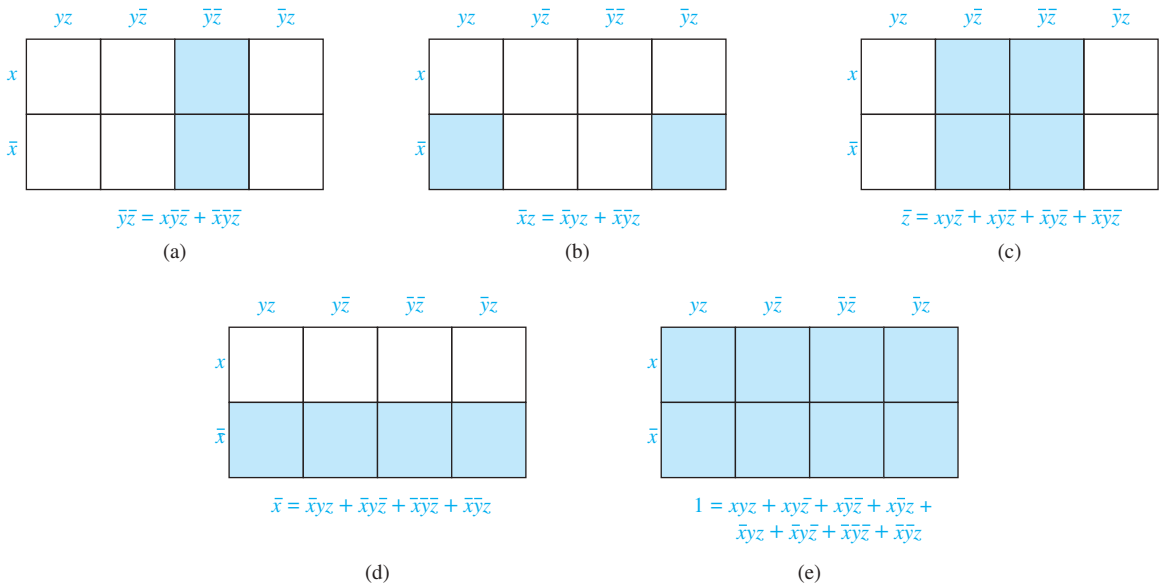


FIGURE 6 **Blocks in K-maps in three variables.**

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

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) $x\bar{z} + \bar{y}\,\bar{z} + \bar{x}yz$, (b) $\bar{y} + \bar{x}z$, (c) $x + \bar{y} + z$, and (d) $x\bar{z} + \bar{x}\,\bar{y}$. In part (d) note that the prime implicants $x\bar{z}$ and $\bar{x}\,\bar{y}$ are essential prime implicants, but the prime implicant $\bar{y}\,\bar{z}$ is a

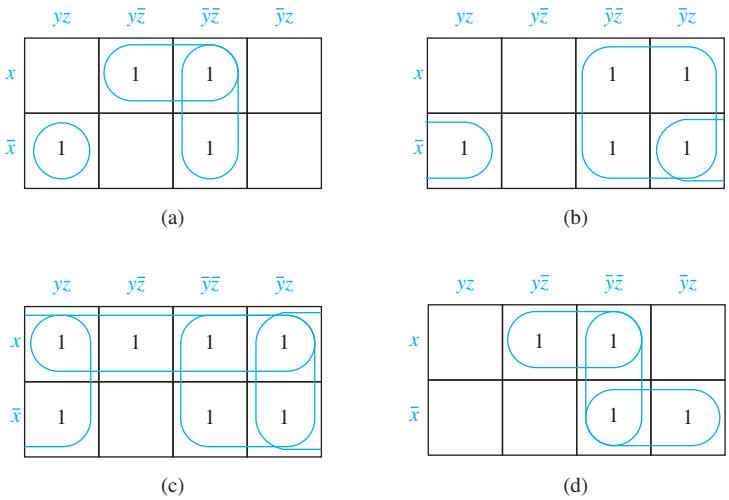


FIGURE 7 **Using K-maps in three variables.**

prime implicant that is not essential, because the cells it covers are covered by the other two prime implicants. ◀

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.**

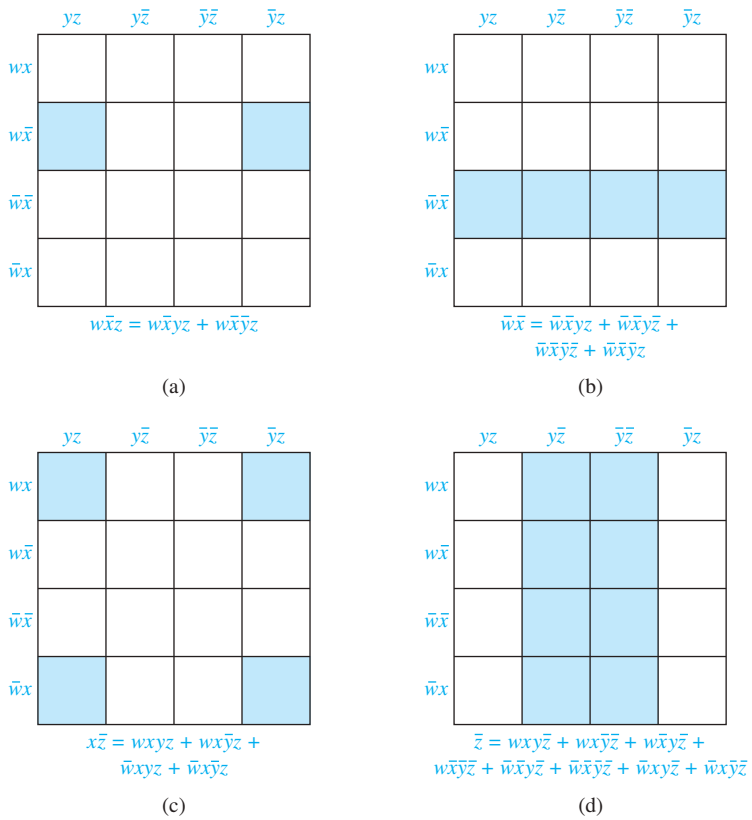


FIGURE 9 **Blocks in K-maps in four variables.**

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

(a) $wxyz + wxyz̄ + wxȳz̄ + wx̄yz + wx̄ȳz + wx̄ȳz̄ + w̄xȳz +$
$w̄x̄yz + w̄x̄yz̄$

(b) $wxȳz̄ + wx̄yz + wx̄yz̄ + wx̄ȳz̄ + w̄xȳz̄ + w̄x̄yz̄ + w̄x̄ȳz̄$

(c) $wxyz̄ + wxȳz̄ + wx̄yz + wx̄yz̄ + wx̄ȳz̄ + w̄xyz + w̄xyz̄ + w̄xȳz̄ +$
$w̄xȳz + w̄x̄yz̄ + w̄x̄ȳz̄$

Solution: The K-maps for these expansions are shown in Figure 10. Using the blocks shown leads to the sum of products (a) $wyz + wxz̄ + wx̄ȳ + w̄x̄y + w̄xȳz$, (b) $ȳz̄ + wx̄y + x̄z̄$, and (c) $z̄ + w̄x + wx̄y$. 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

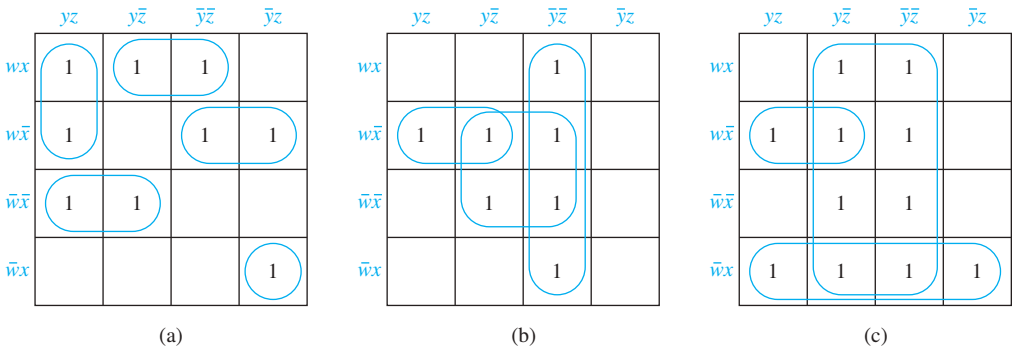


FIGURE 10 **Using K-maps in four variables.**

the top row and bottom row and in the leftmost column and rightmost column in each of these 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 $\bar{x}_1 x_2 x_3 x_4 \bar{x}_5$.

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, $w\bar{x}$, $\overline{w}\,\overline{x}$, and $\overline{w}x$, respectively, and the columns correspond to the products yz, $y\bar{z}$, $\overline{y}\,\overline{z}$, 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, $x_1\bar{x}_2$, $\bar{x}_1\bar{x}_2$, and $\bar{x}_1x_2$, respectively. We label the eight columns using the Gray code 111,110,100,101,001,000,010,011 corresponding to the terms $x_3x_4x_5$, $x_3x_4\bar{x}_5$, $x_3\bar{x}_4\bar{x}_5$, $x_3\bar{x}_4x_5$, $\bar{x}_3\bar{x}_4x_5$, $\bar{x}_3\bar{x}_4\bar{x}_5$, $\bar{x}_3x_4\bar{x}_5$, and $\bar{x}_3x_4x_5$, 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.

12.4.3 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

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

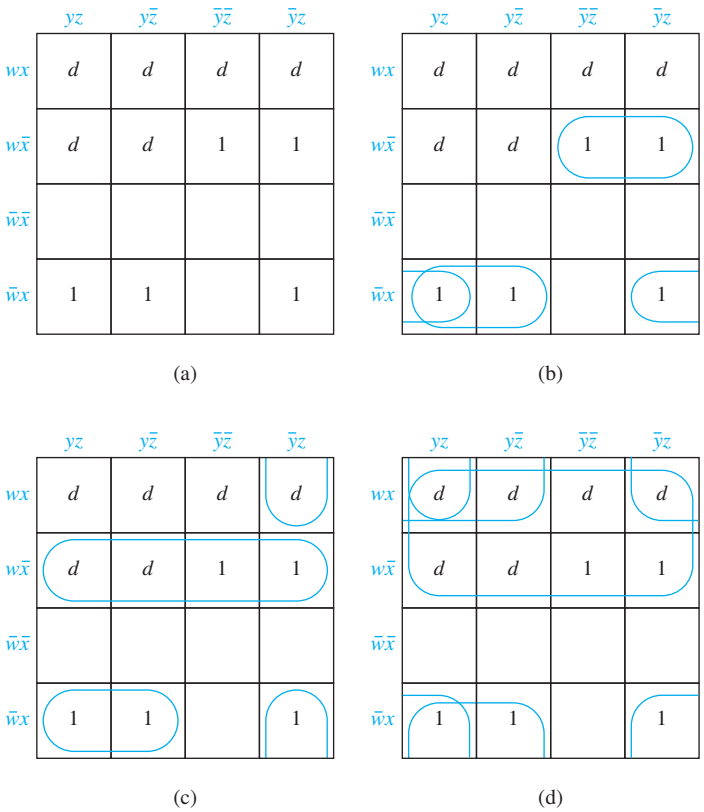


FIGURE 11 **The K-map for F showing its *don't care* positions.**

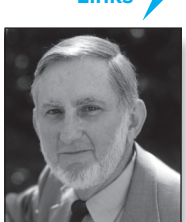
don't care positions, is shown in Figure 11(a). We can either include or exclude squares with ds from blocks. This gives us many possible choices for the blocks. For example, excluding all squares with ds and forming blocks, as shown in Figure 11(b), produces the expression $w\bar{x}\,\bar{y} + \overline{w}xy + \overline{w}xz$. Including some of the ds and excluding others and forming blocks, as shown in Figure 11(c), produces the expression $w\bar{x} + \overline{w}xy + x\bar{y}z$. Finally, including all the ds and using the blocks shown in Figure 11(d) produces the simplest sum-of-products expansion possible, namely, $F(x, y, z) = w + xy + xz$. ◀

12.4.4 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

Links



©Stanford University News Service

EDWARD J. McCLUSKEY (1929–2016) 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 worked in a variety of areas in computer science, including fault-tolerant computing, computer architecture, testing, and logic design. He was the director of the Center for Reliable Computing at Stanford University and an ACM Fellow.

TABLE 2

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

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 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

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

We will represent the minterms in this expansion by bit strings. The first bit will be 1 if x occurs and 0 if $\bar{x}$ occurs. The second bit will be 1 if y occurs and 0 if $\bar{y}$ occurs. The third bit will be 1 if z occurs and 0 if $\bar{z}$ 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 $x\bar{y}z$ and $\bar{x}\,\bar{y}z$, represented by bit strings 101 and 001, can be combined into $\bar{y}z$, 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

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	$x\bar{y}z$	101	(1,3)	yz	–11			
3	$\bar{x}yz$	011	(2,4)	$\bar{y}z$	–01			
4	$\bar{x}\,\bar{y}z$	001	(3,4)	$\bar{x}z$	0–1			
5	$\bar{x}\,\bar{y}\,\bar{z}$	000	(4,5)	$\bar{x}\,\bar{y}$	00–			

TABLE 4

	xyz	$x\bar{y}z$	$\bar{x}yz$	$\bar{x}\bar{y}z$	$\bar{x}\bar{y}\bar{z}$
z	X	X	X	X	
$\bar{x}\bar{y}$				X	X

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 $\bar{y}z$, 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.

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 $\bar{x}\,\bar{y}$ are needed. Hence, the final answer is $z + \bar{x}\,\bar{y}$. ◀

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 $\bar{x}_i$ 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 $\bar{x}_i$ occurs, and a dash in this position if there is no literal involving x_i in the product.

Links

Courtesy of Douglas Quine

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 $wxy\bar{z} + w\bar{x}yz + w\bar{x}y\bar{z} + \bar{w}xyz + \bar{w}xy\bar{z} + \bar{w}\,\bar{x}yz + \bar{w}\,\bar{x}\,\bar{y}z$.

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 $\bar{w}z$, $wy\bar{z}$, $w\bar{x}y$, and $\bar{x}yz$. In Table 7 we show the minterms covered by each of these products. To cover these minterms we must include $\bar{w}z$ and $wy\bar{z}$, because these products are the only products that cover $\bar{w}xyz$ and $wxy\bar{z}$, respectively. Once these two products are included, we see that only one of the two products left is needed. Consequently, we can take either $\bar{w}z + wy\bar{z} + w\bar{x}y$ or $\bar{w}z + wy\bar{z} + \bar{x}yz$ as the final answer. ◀

TABLE 5

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

TABLE 6

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

TABLE 7

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

Exercises

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

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	$\bar{y}$
x	1	
$\bar{x}$	1	1

b)

	y	$\bar{y}$
x	1	1
$\bar{x}$		

c)

	y	$\bar{y}$
x	1	1
$\bar{x}$	1	1

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

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

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

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) $\bar{x}y + \bar{x}\,\bar{y}$

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

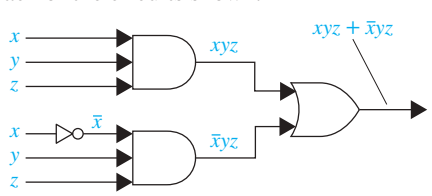
c) $xy + x\bar{y} + \bar{x}y + \bar{x}\,\bar{y}$

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

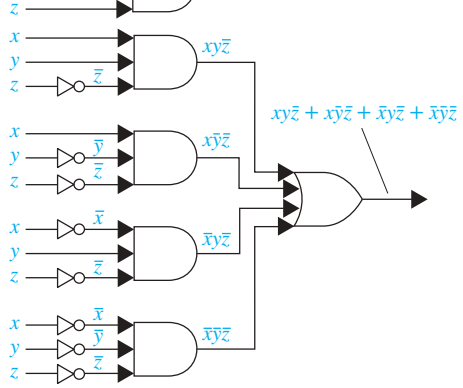
b) Which minterms are represented by cells adjacent to this cell?

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

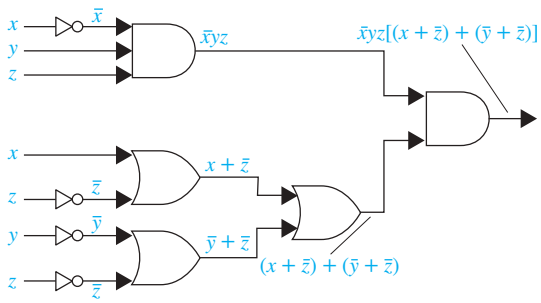
a)



b)



c)



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

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

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

8. Construct a K-map for $F(x, y, z) = xz + yz + xy\bar{z}$. 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 $F(x, y, z) = x\bar{z} + xyz + y\bar{z}$. 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, $\bar{x}y$, and $\bar{y}\,\bar{z}$.

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) $\bar{x}yz + \bar{x}\,\bar{y}z$

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

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

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

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

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

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

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

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

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)** $\bar{x}_1x_3x_5$ **c)** x_2x_4

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

16. How many cells in a K-map for Boolean functions with six variables are needed to represent x_1, $\bar{x}_1x_6$, $\bar{x}_1x_2\bar{x}_6$, $x_2x_3x_4x_5$, and $x_1\bar{x}_2x_4\bar{x}_5$, 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 $(x + y + z)(x + y + \bar{z})(x + \bar{y} + \bar{z})(x + \bar{y} + z)(\bar{x} + y + z)$.

***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 *d*s.

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 $\preceq$.

***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 $x \odot y = xy + \bar{x}\bar{y}$.

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

14. Show that each of these identities holds.

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

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

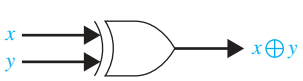
15. Is it always true that $(x \odot y) \odot z = x \odot (y \odot z)$?

***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?

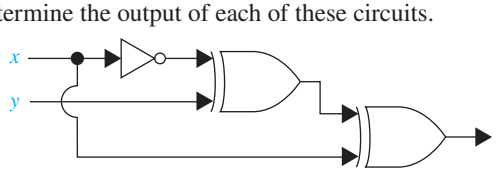
a) $\{^-\}$ **b)** $\{\cdot\}$ **c)** $\{+\}$ **d)** $\{\cdot, +\}$

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

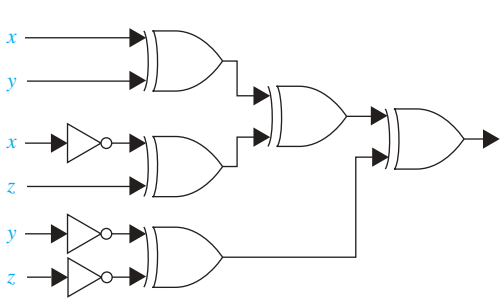


18. Determine the output of each of these circuits.

a)



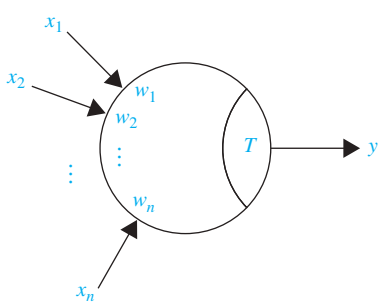
b)



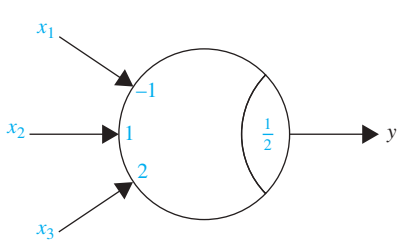
19. Show how a half adder can be constructed using fewer gates than are used in Figure 8 of Section 12.3 when XOR gates can be used in addition to OR gates, AND gates, and inverters.

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) $F(x) = \bar{x}$ **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) $F(w, x, y, z) = wxz + x\bar{y}z$

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

*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

13.1.1 Introduction

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

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

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

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

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

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

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

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

13.1.2 Phrase-Structure Grammars

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

Definition 1 A *vocabulary* (or *alphabet*) V is a finite, nonempty set of elements called *symbols*. A *word* (or *sentence*) over V is a string of finite length of elements of V. The *empty string* or *null string*, denoted by λ (and sometimes by ϵ), is the string containing no symbols. The set of all words over V is denoted by V^*. A *language over* V is a subset of V^*.

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

Languages can be specified in various ways. One way is to list all the words in the language. Another is to give some criteria that a word must satisfy to be in the language. In this section, we describe another important way to specify a language, namely, through the use of a grammar, such as the set of rules we gave in the introduction to this section. A grammar provides a set of symbols of various types and a set of rules for producing words. More precisely, a grammar has a **vocabulary** V, which is a set of symbols used to derive members of the language. Some of the elements of the vocabulary cannot be replaced by other symbols. These are called **terminals**, and the other members of the vocabulary, which can be replaced by other symbols, are called **nonterminals**. The sets of terminals and nonterminals are usually denoted by T and N, respectively. In the example given in the introduction of the section, the set of terminals is {*a, 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 $z_0 \to z_1$ 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 → 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 $P = \{S \to ABa, A \to BB, B \to ab, AB \to b\}$. 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 $z_0 \to z_1$ 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 $w_0 \overset{*}{\Rightarrow} w_n$. 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?

Extra Examples ❯

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 different 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 $S \to \lambda$. Then, G_1 generates the correct set, because using the first production m times puts m 0s at the beginning of the string, and using the second production n times puts n 1s at the end of the string. Furthermore, every string produced by this grammar is of the form $0^m 1^n$, because the only way to add a symbol to the beginning of a string is to add a 0 by applying the production $S \to 0S$, and the only way to add a symbol to the end of the string is to add a 1 by applying the production $S \to S1$.

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. ◀

13.1.3 Types of Phrase-Structure Grammars

Links

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

A **type 0** grammar has no restrictions on its productions. A **type 1** grammar can have productions of the form $w_1 \to w_2$, 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.

13.1.4 Derivation Trees

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

TABLE 1 Types of Grammars.

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

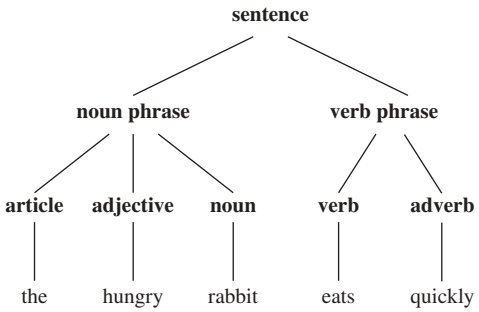


FIGURE 1 **A derivation tree.**

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.

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

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

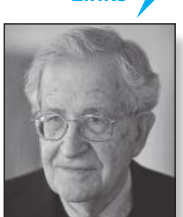
The problem of determining whether a string is in the language generated by a context-free grammar arises in many applications, such as in the construction of compilers. When given a string, the naive approach for determining whether it is in the language generated by a grammar is to look for a sequence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

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

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

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

Links



©SASCHA SCHUERMANN/ AFP/Getty Images

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

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

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

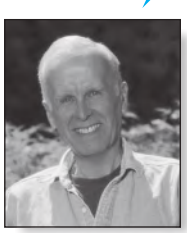
Example 12 presents the solution of a parsing problem using both top-down and bottom-up parsing. Both approaches were easy to use for that problem. However, parsing problems can be quite challenging. That is, it can be quite challenging to determine whether a string is in the language generated by a context-free grammar. Because parsing is so important, many strategies and algorithms have been devised for top-down and for bottom-up parsing. These algorithms are beyond the scope of this book. The interested reader should consult [AhLaSeUl06] to learn more.

13.1.5 Backus–Naur Form

Links

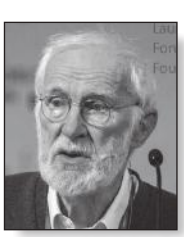
There is another notation that is sometimes used to specify a type 2 grammar, called the **Backus–Naur form (BNF)**, after John Backus, who invented it, and Peter Naur, who refined

Links



Courtesy of Louis Bachrach

JOHN BACKUS (1924–2007) John Backus was born in Philadelphia and grew up in Wilmington, Delaware. He attended the Hill School in Pottstown, Pennsylvania. He needed to attend summer school every year because he disliked studying and was not a serious student. But he enjoyed spending his summers in New Hampshire where he attended summer school and amused himself with summer activities, including sailing. He obliged his father by enrolling at the University of Virginia to study chemistry. But he quickly decided chemistry was not for him, and in 1943 he entered the army, where he received medical training and worked in a 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. (The development of ALGOL led to the development of other programming languages, including Pascal and C.) Later, Backus worked on the mathematics of families of sets and on a functional style of programming. Backus became an IBM Fellow in 1963, and he received the National Medal of Science in 1974 and the prestigious Turing Award from the Association for Computing Machinery in 1977.



©Heidelberg Laureate Forum Foundation

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

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

it for use in the specification of the programming language ALGOL. (Surprisingly, a notation quite similar to the Backus–Naur form was used approximately 2500 years ago to describe the grammar of Sanskrit.) The Backus–Naur form is used to specify the syntactic rules of many computer languages, including Java. The productions in a type 2 grammar have a single 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.

EXAMPLE 13 *Extra Examples*

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

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

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

EXAMPLE 14 What is the Backus–Naur form of the grammar for the subset of English described in the introduction to this section?

Solution: The Backus–Naur form of this grammar is

$\langle sentence\rangle ::= \langle noun\ phrase\rangle\langle verb\ phrase\rangle$
$\langle noun\ phrase\rangle ::= \langle article\rangle\langle adjective\rangle\langle noun\rangle \mid \langle article\rangle\langle noun\rangle$
$\langle verb\ phrase\rangle ::= \langle verb\rangle\langle adverb\rangle \mid \langle verb\rangle$
$\langle article\rangle ::= a \mid the$
$\langle adjective\rangle ::= large \mid hungry$
$\langle noun\rangle ::= rabbit \mid mathematician$
$\langle verb\rangle ::= eats \mid hops$
$\langle adverb\rangle ::= quickly \mid wildly$ ◀

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

$\langle signed\ integer\rangle ::= \langle sign\rangle\langle integer\rangle$
$\langle sign\rangle ::= + \mid -$
$\langle integer\rangle ::= \langle digit\rangle \mid \langle digit\rangle\langle integer\rangle$
$\langle digit\rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$ ◀