CHAPTER 12

Propositional Logic

In this chapter, we introduce propositional logic, an algebra whose original purpose, dating back to Aristotle, was to model reasoning. In more recent times, this algebra, like many algebras, has proved useful as a design tool. For example, Chapter 13 shows how propositional logic can be used in computer circuit design. A third use of logic is as a data model for programming languages and systems, such as the language Prolog. Many systems for reasoning by computer, including theorem provers, program verifiers, and applications in the field of artificial intelligence, have been implemented in logic-based programming languages. These languages generally use "predicate logic," a more powerful form of logic that extends the capabilities of propositional logic. We shall meet predicate logic in Chapter 14.

12.1 What This Chapter Is About

Section 12.2 gives an intuitive explanation of what propositional logic is, and why it is useful. The next section, 12,3, introduces an algebra for logical expressions with Boolean-valued operands and with logical operators such as `AND`, `OR`, and `NOT` that operate on Boolean (true/false) values. This algebra is often called *Boolean algebra* after George Boole, the logician who first framed logic as an algebra. We then learn the following ideas.

Boolean algebra

- Truth tables are a useful way to represent the meaning of an expression in logic (Section 12.4).
- We can convert a truth table to a logical expression for the same logical function (Section 12.5).
- The Karnaugh map is a useful tabular technique for simplifying logical expressions (Section 12.6).
- There is a rich set of "tautologies," or algebraic laws that can be applied to logical expressions (Sections 12.7 and 12.8).

- Certain tautologies of propositional logic allow us to explain such common proof techniques as "proof by contradiction" or "proof by contrapositive" (Section 12.9).
- Propositional logic is also amenable to "deduction," that is, the development of proofs by writing a series of lines, each of which either is given or is justified by some previous lines (Section 12.10). This is the mode of proof most of us learned in a plane geometry class in high school.
- A powerful technique called "resolution" can help us find proofs quickly (Section 12.11).

12.2 What Is Propositional Logic?

Sam wrote a C program containing the if-statement

```
if (a < b || (a >= b && c == d)) ...
```
(12.1)

Sally points out that the conditional expression in the if-statement could have been written more simply as

```
if (a < b || c == d) ...
```
(12.2)

How did Sally draw this conclusion?

She might have reasoned as follows. Suppose `a < b`. Then the first of the two `OR`'ed conditions is true in both statements, so the then-branch is taken in either of the if-statements (12.1) and (12.2).

Now suppose `a < b` is false. In this case, we can only take the then-branch if the second of the two conditions is true. For statement (12.1), we are asking whether

```
a >= b && c == d
```

is true. Now `a >= b` is surely true, since we assume `a < b` is false. Thus we take the then-branch in (12.1) exactly when `c == d` is true. For statement (12.2), we clearly take the then-branch exactly when `c == d`. Thus no matter what the values of `a`, `b`, `c`, and `d` are, either both or neither of the if-statements cause the then-branch to be followed. We conclude that Sally is right, and the simplified conditional expression can be substituted for the first with no change in what the program does.

Propositional logic is a mathematical model that allows us to reason about the truth or falsehood of logical expressions. We shall define logical expressions formally in the next section, but for the time being we can think of a logical expression as a simplification of a conditional expression such as lines (12.1) or (12.2) above that abstracts away the order of evaluation contraints of the logical operators in C.

Propositions and Truth Values

Notice that our reasoning about the two if-statements above did not depend on what `a < b` or similar conditions "mean." All we needed to know was that the conditions `a < b` and `a >= b` are *complementary*, that is, when one is true the other is false and vice versa. We may therefore replace the statement `a < b` by a single symbol p, replace `a >= b` by the expression `NOT` p, and replace `c == d` by the symbol q. The symbols p and q are called *propositional variables,* since they can stand for any

Propositional variable

"proposition," that is, any statement that can have one of the *truth values*, true or false.

Logical expressions can contain logical operators such as `AND`, `OR`, and `NOT`. When the values of the operands of the logical operators in a logical expression are known, the value of the expression can be determined using rules such as

1. The expression p `AND` q is true only when both p and q are true; it is false otherwise.

2. The expression p `OR` q is true if either p or q, or both are true; it is false otherwise.

3. The expression `NOT` p is true if p is false, and false if p is true.

The operator `NOT` has the same meaning as the C operator `!`. The operators `AND` and `OR` are like the C operators `&&` and `||`, respectively, but with a technical difference. The C operators are defined to evaluate the second operand only when the first operand does not resolve the matter — that is, when the first operation of `&&` is true or the first operand of `||` is false. However, this detail is only important when the C expression has side effects. Since there are no "side effects" in the evaluation of logical expressions, we can take `AND` to be synonymous with the C operator `&&` and take `OR` to be synonymous with `||`.

For example, the condition in Equation (12.1) can be written as the logical expression

$$p \texttt{ OR } \big((\texttt{NOT } p) \texttt{ AND } q\big)$$

and Equation (12.2) can be written as p `OR` q. Our reasoning about the two if-statements (12.1) and (12.2) showed the general proposition that

$$\Big(p \texttt{ OR } \big((\texttt{NOT } p) \texttt{ AND } q\big)\Big) \equiv (p \texttt{ OR } q) \tag{12.3}$$

where $\equiv$ means "is equivalent to" or "has the same Boolean value as." That is, no matter what truth values are assigned to the propositional variables p and q, the left-hand side and right-hand side of $\equiv$ are either both true or both false. We discovered that for the equivalence above, both are true when p is true or when q is true, and both are false if p and q are both false. Thus, we have a valid equivalence.

As p and q can be any propositions we like, we can use equivalence (12.3) to simplify many different expressions. For example, we could let p be

```
a == b+1 && c < d
```

while q is `a == c || b == c`. In that case, the left-hand side of (12.3) is

```
(a == b+1 && c < d) ||                                          (12.4)
    ( !(a == b+1 && c < d) && (a == c || b == c))
```

Note that we placed parentheses around the values of p and q to make sure the resulting expression is grouped properly.

Equivalence (12.3) tells us that (12.4) can be simplified to the right-hand side of (12.3), which is

```
(a == b+1 && c < d) || (a == c || b == c)
```

What Propositional Logic Cannot Do

Propositional logic is a useful tool for reasoning, but it is limited because it cannot see inside propositions and take advantage of relationships among them. For example, Sally once wrote the if-statement

```
if (a < b && a < c && b < c) ...
```

Then Sam pointed out that it was sufficient to write

```
if (a < b && b < c) ...
```

If we let p, q, and r stand for the propositions `(a < b)`, `(a < c)`, and `(b < c)`, respectively, then it looks like Sam said that

$$(p \texttt{ AND } q \texttt{ AND } r) \equiv (p \texttt{ AND } r)$$

This equivalence, however, is not always true. For example, suppose p and r were true, but q were false. Then the right-hand side would be true and the left-hand side false.

It turns out that Sam's simplification is correct, but not for any reason that we can discover using propositional logic. You may recall from Section 7.10 that $<$ is a transitive relation. That is, whenever both p and r, that is, `a < b` and `b < c`, are true, it must also be that q, which is `a < c`, is true.

Predicate logic

In Chapter 14, we shall consider a more powerful model called predicate logic that allows us to attach arguments to propositions. That privilege allows us to exploit special properties of operators like $<$. (For our purposes, we can think of a predicate as the name for a relation in the set-theoretic sense of Chapters 7 and 8.) For example, we could create a predicate lt to represent operator $<$, and write p, q, and r as $lt(a, b)$, $lt(a, c)$, and $lt(b, c)$. Then, with suitable laws that expressed the properties of lt, such as transitivity, we could conclude that

$$\big(lt(a,b) \texttt{ AND } lt(a,c) \texttt{ AND } lt(b,c)\big) \equiv \big(lt(a,b) \texttt{ AND } lt(b,c)\big)$$

In fact, the above holds for any predicate lt that obeys the transitive law, not just for the predicate $<$.

As another example, we could let p be the proposition, "It is sunny," and q the proposition, "Joe takes his umbrella." Then the left-hand side of (12.3) is

"It is sunny, or it is not sunny and Joe takes his umbrella."

while the right-hand side, which says the same thing, is

"It is sunny or Joe takes his umbrella."

12.3 Logical Expressions

As mentioned in the previous section, *logical expressions* are defined recursively as follows.

BASIS. Propositional variables and the logical constants, `TRUE` and `FALSE`, are logical expressions. These are the atomic operands.

INDUCTION. If E and F are logical expressions, then so are

a) E `AND` F. The value of this expression is `TRUE` if both E and F are `TRUE` and `FALSE` otherwise.

b) E `OR` F. The value of this expression is `TRUE` if either E or F or both are `TRUE`, and the value is `FALSE` if both E and F are `FALSE`.

c) `NOT` E. The value of this expression is `TRUE` if E is `FALSE` and `FALSE` if E is `TRUE`.

That is, logical expressions can be built from the binary infix operators `AND` and `OR`, the unary prefix operator `NOT`. As with other algebras, we need parentheses for grouping, but in some cases we can use the precedence and associativity of operators to eliminate redundant pairs of parentheses, as we do in the conditional expressions of C that involve these logical operators. In the next section, we shall see more logical operators than can appear in logical expressions.

✦ **Example 12.1.** Some examples of logical expressions are:

1. `TRUE`
2. `TRUE OR FALSE`
3. `NOT` p
4. p `AND` (q `OR` r)
5. (q `AND` p) `OR` (`NOT` p)

In these expressions, p, q, and r are propositional variables. ✦

Precedence of Logical Operators

As with expressions of other sorts, we assign a precedence to logical operators, and we can use this precedence to eliminate certain pairs of parentheses. The precedence order for the operators we have seen so far is `NOT` (highest), then `AND`, then `OR` (lowest). Both `AND` and `OR` are normally grouped from the left, although we shall see that they are associative, and that the grouping is therefore irrelevant. `NOT`, being a unary prefix operator, can only group from the right.

✦ **Example 12.2.** `NOT NOT` p `OR` q is grouped $\big($`NOT` (`NOT` p)$\big)$ `OR` q. `NOT` p `OR` q `AND` r is grouped (`NOT` p) `OR` (q `AND` r). You should observe that there is an analogy between the precedence and associativity of `AND`, `OR`, and `NOT` on one hand, and the arithmetic operators $\times$, $+$, and unary $-$ on the other. For instance, the second of the above expressions can be compared with the arithmetic expression $-p + q \times r$, which has the same grouping, $(-p) + (q \times r)$. ✦

Evaluating Logical Expressions

When all of the propositional variables in a logical expression are assigned truth values, the expression itself acquires a truth value. We can then evaluate a logical expression just as we would an arithmetic expression or a relational expression.

Truth assignment

The process is best seen in terms of the expression tree for an expression, such as that shown in Fig. 12.1 for the expression p `AND` $(q$ `OR` $r)$ `OR` s. Given a *truth assignment,* that is, an assignment of `TRUE` or `FALSE` to each variable, we begin at the leaves, which are atomic operands. Each atomic operand is either one of the logical constants `TRUE` or `FALSE`, or is a variable that is given one of the values `TRUE` or `FALSE` by the truth assignment. We then work up the tree. Once the value of the children of an interior node v are known, we can apply the operator at v to these values and produce a truth value for node v. The truth value at the root is the truth value of the whole expression.

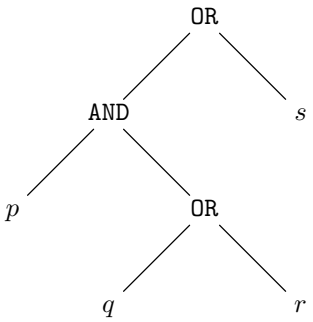


Fig. 12.1. Expression tree for the logical expression p `AND` $(q$ `OR` $r)$ `OR` s.

◆ **Example 12.3.** Suppose we want to evaluate the expression p `AND` $(q$ `OR` $r)$ `OR` s with the truth assignment `TRUE`, `FALSE`, `TRUE`, `FALSE`, for p, q, r, and s, respectively. We first consider the lowest interior node in Fig. 12.1, which represents the expression q `OR` r. Since q is `FALSE`, but r is `TRUE`, the value of q `OR` r is `TRUE`.

We now work on the node with the `AND` operator. Both its children, representing expressions p and q `OR` r, have the value `TRUE`. Thus this node, representing expression p `AND` $(q$ `OR` $r)$, also has the value `TRUE`.

Finally, we work on the root, which has operator `OR`. Its left child, we just discovered has value `TRUE`, and its right child, which represents expression s, has value `FALSE` according to the truth assignment. Since `TRUE OR FALSE` evaluates to `TRUE`, the entire expression has value `TRUE`. ◆

Boolean Functions

The "meaning" of any expression can be described formally as a function from the values of its arguments to a value for the whole expression. For example, the arithmetic expression $x \times (x + y)$ is a function that takes values for x and y (say reals) and returns the value obtained by adding the two arguments and multiplying the sum by the first argument. The behavior is similar to that of a C function declared

```
float foo(float x, float y)
{
    return x*(x+y);
}
```

In Chapter 7 we learned about functions as sets of pairs with a domain and range. We could also represent an arithmetic expression like $x \times (x+y)$ as a function whose domain is pairs of reals and whose range is the reals. This function consists of pairs of the form $\big((x, y),\ x \times (x + y)\big)$. Note that the first component of each pair is itself a pair, (x, y). This set is infinite; it contains members like $\big((3, 4), 21\big)$, or $\big((10, 12.5), 225\big)$.

Similarly, a logical expression's meaning is a function that takes truth assignments as arguments and returns either `TRUE` or `FALSE`. Such functions are called *Boolean functions.* For example, the logical expression

E: p `AND` $(p$ `OR` $q)$

is similar to a C function declared

```
BOOLEAN foo(BOOLEAN p, BOOLEAN q)
{
    return p && (p || q);
}
```

Like arithmetic expressions, Boolean expressions can be thought of as sets of pairs. The first component of each pair is a truth assignment, that is, a tuple giving the truth value of each propositional variable in some specified order. The second component of the pair is the value of the expression for that truth assignment.

✦ **Example 12.4.** The expression $E = p$ `AND` $(p$ `OR` $q)$ can be represented by a function consisting of four members. We shall represent truth values by giving the value for p before the value for q. Then $\big((\texttt{TRUE}, \texttt{FALSE}), \texttt{TRUE}\big)$ is one of the pairs in the set representing E as a function. It says that when p is true and q is false, p `AND` $(p$ `OR` $q)$ is true. We can determine this value by working up the expression tree for E, by the process in Example 12.3. The reader can evaluate E for the other three truth assignments, and thus build the entire Boolean function that E represents. ✦

EXERCISES

12.3.1: Evaluate the following expressions for all possible truth values, to express their Boolean functions as a set-theoretic function.

a) p `AND` $(p$ `OR` $q)$

b) `NOT` p `OR` q

c) $(p$ `AND` $q)$ `OR` (`NOT` p `AND` `NOT` $q)$

12.3.2: Write C functions to implement the logical expressions in Exercise 12.3.1.

12.4 Truth Tables

It is convenient to display a Boolean function as a *truth table,* in which the rows correspond to all possible combinations of truth values for the arguments. There is a column for each argument and a column for the value of the function.

p	q	p AND q
0	0	0
0	1	0
1	0	0
1	1	1

p	q	p OR q
0	0	0
0	1	1
1	0	1
1	1	1

p	NOT p
0	1
1	0

Fig. 12.2. Truth tables for AND, OR, and NOT.

Example 12.5. The truth tables for AND, OR, and NOT are shown in Fig. 12.2. Here, and frequently in this chapter, we shall use the shorthand that 1 stands for TRUE and 0 stands for FALSE. Thus the truth table for AND says that the result is TRUE if and only if both operands are TRUE; the second truth table says that the result of applying the OR operator is TRUE when either of the operands, or both, are TRUE; the third truth table says that the result of applying the NOT operator is TRUE if and only if the operand has the value FALSE. ◆

The Size of Truth Tables

Suppose a Boolean function has k arguments. Then a truth assignment for this function is a list of k elements, each element either TRUE or FALSE. Counting the number of truth assignments for k variables is an example of the assignment-counting problem considered in Section 4.2. That is, we assign one of the two truth values to each of k items, the propositional variables. That is analogous to painting k houses with two choices of color. The number of truth assignments is thus 2^k.

The truth table for a Boolean function of k arguments thus has 2^k rows, one for each truth assignment. For example, if $k = 2$ there are four rows, corresponding to 00, 01, 10, and 11, as we see in the truth tables for AND and OR in Fig. 12.2.

While truth tables involving one, two, or three variables are relatively small, the fact that the number of rows is 2^k for a k-ary function tells us that k does not have to get too big before it becomes unfeasible to draw truth tables. For example, a function with ten arguments has over 1000 rows. In later sections we shall have to contend with the fact that, while truth tables are finite and in principle tell us everything we need to know about a Boolean function, their exponentially growing size often forces us to find other means to understand, evaluate, or compare Boolean functions.

Understanding "Implies"

The meaning of the implication operator $\rightarrow$ may appear unintuitive, since we must get used to the notion that "falsehood implies everything." We should not confuse $\rightarrow$ with causation. That is, $p \rightarrow q$ may be true, yet p does not "cause" q in any sense. For example, let p be "it is raining," and q be "Sue takes her umbrella." We might assert that $p \rightarrow q$ is true. It might even appear that the rain is what caused Sue to take her umbrella. However, it could also be true that Sue is the sort of person who doesn't believe weather forecasts and prefers to carry an umbrella at all times.

Counting the Number of Boolean Functions

While the number of rows in a truth table for a k-argument Boolean function grows exponentially in k, the number of different k-ary Boolean functions grows much faster still. To count the number of k-ary Boolean functions, note that each such function is represented by a truth table with 2^k rows, as we observed. Each row is assigned a value, either `TRUE` or `FALSE`. Thus, the number of different Boolean functions of k arguments is the same as the number of assignments to 2^k items of 2 values. This number is 2^{2^k}. For example, when $k = 2$, there are $2^{2^2} = 16$ functions, and for $k = 5$ there are $2^{2^5} = 2^{32}$, or about four billion functions.

Of the 16 Boolean functions of 2 arguments, we already met two: `AND` and `OR`. Some others are trivial, such as the function that has value 1 no matter what its arguments are. However, there are a number of other functions of two arguments that are useful, and we shall meet them later in this section. We have also seen `NOT`, a useful function of one argument, and one often uses Boolean functions of three or more arguments as well.

Additional Logical Operators

There are four other Boolean functions of two arguments that will prove very useful in what follows.

1. *Implication,* written $\rightarrow$. We write $p \rightarrow q$ to mean that "if p is true, then q is true." The truth table for $\rightarrow$ is shown in Fig. 12.3. Notice that there is only one way $p \rightarrow q$ can be made false: p must be true and q must be false. If p is false, then $p \rightarrow q$ is always true, and if q is true, then $p \rightarrow q$ is always true.

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Fig. 12.3. Truth table for "implies."

2. *Equivalence,* written $\equiv$, means "if and only if"; that is, $p \equiv q$ is true when both p and q are true, or when both are false, but not otherwise. Its truth table is shown in Fig. 12.4. Another way of looking at the $\equiv$ operator is that it asserts that the operands on the left and right have the same truth value. That is what we meant in Section 12.2 when we claimed, for example, that $\big(p \texttt{ OR } (\texttt{NOT } p \texttt{ AND } q)\big) \equiv (p \texttt{ OR } q)$.

3. The NAND, or "not-and," operator applies AND to its operands and then complements the result by applying NOT. We write $p \texttt{ NAND } q$ to denote $\texttt{NOT } (p \texttt{ AND } q)$.

4. Similarly, the NOR, or "not-or," operator takes the OR of its operands and complements the result; $p \texttt{ NOR } q$ denotes $\texttt{NOT } (p \texttt{ OR } q)$. The truth tables for NAND and NOR are shown in Fig. 12.4.

p	q	$p \equiv q$
0	0	1
0	1	0
1	0	0
1	1	1

p	q	p NAND q
0	0	1
0	1	1
1	0	1
1	1	0

p	q	p NOR q
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 12.4. Truth tables for equivalence, NAND, and NOR.

Operators with Many Arguments

Some logical operators can take more than two arguments as a natural extension. For example, it is easy to see that AND is associative [$(p \texttt{ AND } q) \texttt{ AND } r$ is equivalent to $p \texttt{ AND } (q \texttt{ AND } r)$]. Thus an expression of the form $p_1 \texttt{ AND } p_2 \texttt{ AND } \cdots \texttt{ AND } p_k$ can be grouped in any order; its value will be TRUE exactly when all of $p_1, p_2, \ldots, p_k$ are TRUE. We may thus write this expression as a function of k arguments,

$$\texttt{AND } (p_1, p_2, \ldots, p_k)$$

Its truth table is suggested in Fig. 12.5. As we see, the result is 1 only when all arguments are 1.

p_1	p_2	$\cdots$	p_{k-1}	p_k	AND $(p_1, p_2, \ldots, p_k)$
0	0	$\cdots$	0	0	0
0	0	$\cdots$	0	1	0
0	0	$\cdots$	1	0	0
0	0	$\cdots$	1	1	0
.	.		.	.	.
.	.		.	.	.
.	.		.	.	.
1	1	$\cdots$	1	0	0
1	1	$\cdots$	1	1	1

Fig. 12.5. Truth table for k-argument AND.

The Significance of Some Operators

The reason that we are especially interested in k-ary AND, OR, NAND, and NOR is that these operators are particularly easy to implement electronically. That is, there are simple means to build "gates," which are electronic circuits that take k inputs and produce the AND, OR, NAND, or NOR of these inputs. While the details of the underlying electronic technologies are beyond the scope of this book, the general idea is to represent 1 and 0, or TRUE and FALSE, by two different voltage levels. Some other operators, such as $\equiv$ or $\rightarrow$, are not that easy to implement electronically, and we generally use several gates of the NAND or NOR type to implement them. The NOT operator, however, can be thought of as either a 1-argument NAND or a 1-argument NOR, and therefore is also "easy" to implement.

Similarly, OR is associative, and we can denote the logical expression p_1 OR p_2 OR $\cdots$ OR p_k as a single Boolean function OR $(p_1, p_2, \ldots, p_k)$. The truth table for this k-ary OR, which we shall not show, has 2^k rows, like the table for k-ary AND. For the OR, however, the first row, where p_1, $p_2, \ldots, p_k$ are all assigned 0, has the value 0; the remaining $2^k - 1$ rows have the value 1.

The binary operators NAND and NOR are commutative, but not associative. Thus the expression without parentheses, p_1 NAND p_2 NAND $\cdots$ NAND p_k, has no intrinsic meaning. When we speak of k-ary NAND, we do not mean any of the possible groupings of

$$p_1 \text{ NAND } p_2 \text{ NAND } \cdots \text{ NAND } p_k$$

Rather, we define NAND $(p_1, p_2, \ldots, p_k)$ to be equivalent to the expression

$$\text{NOT } (p_1 \text{ AND } p_2 \text{ AND } \cdots \text{ AND } p_k)$$

That is, NAND $(p_1, p_2, \ldots, p_k)$ has value 0 if all of $p_1, p_2, \ldots, p_k$ have value 1, and it has value 1 for all the $2^k - 1$ other combinations of input values.

Similarly, NOR $(p_1, p_2, \ldots, p_k)$ stands for NOT $(p_1$ OR p_2 OR $\cdots$ OR $p_k)$. It has value 1 if p_1, $p_2, \ldots, p_k$ all have value 0; it has value 0 otherwise.

Associativity and Precedence of Logical Operators

The order of precedence we shall use is

1. NOT (highest)
2. NAND
3. NOR
4. AND
5. OR
6. $\rightarrow$
7. $\equiv$ (lowest)

Thus, for example, $p \rightarrow q \equiv$ NOT p OR q is grouped $(p \rightarrow q) \equiv \big((\text{NOT } p) \text{ OR } q\big)$.

As we mentioned earlier, AND and OR are associative and commutative; so is $\equiv$. We shall assume that they group from the left if it is necessary to be specific. The other binary operators listed above are not associative. We shall generally show parentheses around them explicitly to avoid ambiguity, but each of the operators $\rightarrow$, NAND, and NOR will be grouped from the left in strings of two or more of the same operator.

Using Truth Tables to Evaluate Logical Expressions

The truth table is a convenient way to calculate and display the value of an expression E for all possible truth assignments, as long as there are not too many variables in the expression. We begin with columns for each of the variables appearing in E, and follow with columns for the various subexpressions of E, in an order that represents a bottom-up evaluation of the expression tree for E.

When we apply an operator to the columns representing the values of some nodes, we perform an operation on the columns that corresponds to the operator in a simple way. For example, if we wish to take the `AND` of two columns, we put 1 in those rows that have 1 in both columns, and we put 0's in the other rows. To take the `OR` of two columns, we put a 1 in those rows where one or both of the columns have 1, and we put 0's elsewhere. To take the `NOT` of a column, we *complement* the column, putting a 1 where the column has a 0 and vice-versa. As a last example, to apply the operator $\rightarrow$ to two columns, the result has a 0 only where the first has 1 and the second has 0; other rows have 1 in the result.

The rule for some other operators is left for an exercise. In general, we apply an operator to columns by applying that operator, row by row, to the values in that row.

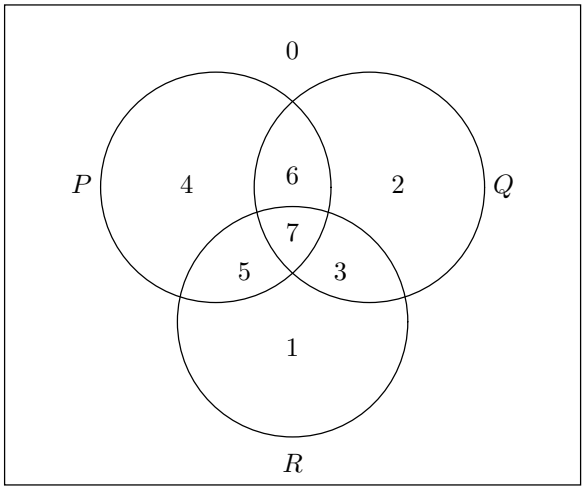
◆ **Example 12.6.** Consider the expression E: (p `AND` q) $\rightarrow$ (p `OR` r). Figure 12.6 shows the truth table for this expression and its subexpressions. Columns (1), (2), and (3) give the values of the variables p, q, and r in all combinations. Column (4) gives the value of subexpression p `AND` q, which is computed by putting a 1 wherever there is a 1 in both columns (1) and (2). Column (5) shows the value of expression p `OR` r; it is obtained by putting a 1 in those rows where either column (1) or (3), or both, has a 1. Finally, column (6) represents the whole expression E. It is formed from columns (4) and (5); it has a 1 except in those rows where column (4) has 1 and column (5) has 0. Since there is no such row, column (6) is all 1's, which says that E has the truth value 1 no matter what its arguments are. Such an expression is called a "tautology," as we shall see in Section 12.7. ◆

(1)	(2)	(3)	(4)	(5)	(6)
p	q	r	p `AND` q	p `OR` r	E
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Fig. 12.6. Truth table for (p `AND` q) $\rightarrow$ (p `OR` r).

Venn Diagrams and Truth Tables

There is a similarity between truth tables and the Venn diagrams for set operations that we discussed in Section 7.3. First, the operation union on sets acts like `OR` on truth values, and intersection of sets acts like `AND`. We shall see in Section 12.8 that these two pairs of operations obey the same algebraic laws. Just as an expression involving k sets as arguments results in a Venn diagram with 2^k regions, a logical expression with k variables results in a truth table with 2^k rows. Further, there is a natural correspondence between the regions and the rows. For example, a logical expression with variables p, q, and r corresponds to a set expression involving sets P, Q, and R. Consider the Venn diagram for these sets:



Here, the region 0 corresponds to the set of elements that are in none of P, Q, and R, region 1 corresponds to the elements that are in R, but not in P or Q. In general, if we look at the 3-place binary representation of a region number, say abc, then the elements of the region are in P if $a = 1$, in Q if $b = 1$, and in R if $c = 1$. Thus the region numbered $(abc)_2$ corresponds to the row of the truth table where p, q, and r have truth values a, b, and c, respectively.

When dealing with Venn diagrams, we took the union of two sets of regions to include the regions in either set. In analogy, when we take the `OR` of columns in a truth table, we put 1 in the union of the rows that have 1 in the first column and the rows that have 1 in the second column. Similarly, we intersect sets of regions in a Venn diagram by taking only those regions in both sets, and we take the `AND` of columns by putting a 1 in the intersection of the set of rows that have 1 in the first column and the set of rows with 1 in the second column.

The logical `NOT` operator does not quite correspond to a set operator. However, if we imagine that the union of all the regions is a "universal set," then logical `NOT` corresponds to taking a set of regions and producing the set consisting of the remaining regions of the Venn diagram, that is, subtracting the given set from the universal set.

EXERCISES

12.4.1: Give the rule for computing the (a) `NAND` (b) `NOR` (c) $\equiv$ of two columns of a truth table.

12.4.2: Compute the truth table for the following expressions and their subexpressions.

a) $(p \rightarrow q) \equiv (\texttt{NOT}\ p\ \texttt{OR}\ q)$

b) $p \rightarrow \big(q \rightarrow (r\ \texttt{OR}\ \texttt{NOT}\ p)\big)$

c) $(p\ \texttt{OR}\ q) \rightarrow (p\ \texttt{AND}\ q)$

12.4.3*: To what set operator does the logical expression p `AND` `NOT` q correspond? (See the box comparing Venn diagrams and truth tables.)

12.4.4*: Give examples to show that $\rightarrow$, `NAND`, and `NOR` are not associative.

12.4.5**: A Boolean function f *does not depend on the first argument* if

$$f(\texttt{TRUE}, x_2, x_3, \ldots, x_k) = f(\texttt{FALSE}, x_2, x_3, \ldots, x_k)$$

for any truth values $x_2, x_3, \ldots, x_k$. Similarly, we can say f does not depend on its ith argument if the value of f never changes when its ith argument is switched between `TRUE` and `FALSE`. How many Boolean functions of two arguments do not depend on their first or second argument (or both)?

12.4.6*: Construct truth tables for the 16 Boolean functions of two variables. How many of these functions are commutative?

Exclusive or

12.4.7: The binary *exclusive-or* function, $\oplus$, is defined to have value `TRUE` if and only if exactly one of its arguments are `TRUE`.

a) Draw the truth table for $\oplus$.

b)* Is $\oplus$ commutative? Is it associative?

12.5 From Boolean Functions to Logical Expressions

Now, let us consider the problem of designing a logical expression from a truth table. We start with a truth table as the specification of the logical expression, and our goal is to find an expression with the given truth table. Generally, there is an infinity of different expressions we could use; we usually limit our selection to a particular set of operators, and we often want an expression that is "simplest" in some sense.

This problem is a fundamental one in circuit design. The logical operators in the expression may be taken as the gates of the circuit, and so there is a straightforward translation from a logical expression to an electronic circuit, by a process we shall discuss in the next chapter.

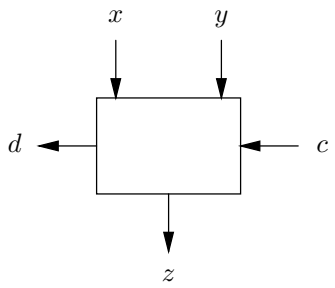


Fig. 12.7. A one-bit adder: $(dz)_2$ is the sum $x + y + c$.

✦ **Example 12.7.** As we saw in Section 1.3, we can design a 32-bit adder out of one-bit adders of the type shown in Fig. 12.7. The one-bit adder sums two input bits x and y, and a carry-in bit c, to produce a carry-out bit d and a sum bit z.

The truth table in Fig. 12.8 tells us the value of the carry-out bit d and the sum-bit z, as a function of x, y, and c for each of the eight combinations of input values. The carry-out bit d is 1 if at least two of x, y, and c have the value 1, and $d = 0$ if only zero or one of the inputs is 1. The sum bit z is 1 if an odd number of x, y, and c are 1, and 0 if not.

	x	y	c	d	z
0)	0	0	0	0	0
1)	0	0	1	0	1
2)	0	1	0	0	1
3)	0	1	1	1	0
4)	1	0	0	0	1
5)	1	0	1	1	0
6)	1	1	0	1	0
7)	1	1	1	1	1

Fig. 12.8. Truth table for the carry-out bit d and the sum-bit z.

We shall present a general way to go from a truth table to a logical expression momentarily. However, given the carry-out function d of Fig. 12.8, we might reason in the following manner to construct a corresponding logical expression.

1. From rows 3 and 7, d is 1 if both y and c are 1.
2. From rows 5 and 7, d is 1 if both x and c are 1.
3. From rows 6 and 7, d is 1 if both x and y are 1.

Condition (1) can be modeled by the logical expression y `AND` c, because y `AND` c is true exactly when both y and c are 1. Similarly, condition (2) can be modeled by x `AND` c, and condition (3) by x `AND` y.

All the rows that have $d = 1$ are included in at least one of these three pairs of rows. Thus we can write a logical expression that is true whenever one or more of the three conditions hold by taking the logical `OR` of these three expressions:

$$(y \texttt{ AND } c) \texttt{ OR } (x \texttt{ AND } c) \texttt{ OR } (x \texttt{ AND } y) \tag{12.5}$$

The correctness of this expression is checked in Fig. 12.9. The last four columns correspond to the subexpressions y AND c, x AND c, x AND y, and expression (12.5). ✥

x	y	c	y AND c	x AND c	x AND y	d
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	1
1	0	0	0	0	0	0
1	0	1	0	1	0	1
1	1	0	0	0	1	1
1	1	1	1	1	1	1

Fig. 12.9. Truth table for carry-out expression (12.5) and its subexpressions.

Shorthand Notation

Before proceeding to describe how we build expressions from truth tables, there are some simplifications in notation that will prove helpful.

1. We can represent the AND operator by juxtaposition, that is, by no operator at all, just as we often represent multiplication, and as we represented concatenation in Chapter 10.
2. The OR operator can be represented by $+$.
3. The NOT operator can be represented by an overbar. This convention is especially useful when the NOT applies to a single variable, and we shall often write NOT p as $\bar{p}$.

✥ **Example 12.8.** The expression p AND q OR r can be written $pq + r$. The expression p AND NOT q OR NOT r can be written $p\bar{q} + \bar{r}$. We can even mix our original notation with the shorthand notation. For example, the expression

$$\big((p \texttt{ AND } q) \rightarrow r\big) \texttt{ AND } (p \rightarrow s)$$

could be written $(pq \rightarrow r)$ AND $(p \rightarrow s)$ or even as $(pq \rightarrow r)(p \rightarrow s)$. ✥

One important reason for the new notation is that it allows us to think of AND and OR as if they were multiplication and addition in arithmetic. Thus we can apply such familiar laws as commutativity, associativity, and distributivity, which we shall see in Section 12.8 apply to these logical operators, just as they do to the corresponding arithmetic operators. For example, we shall see that $p(q + r)$ can be replaced by $pq + pr$, and then by $rp + qp$, whether the operators involved are AND and OR, or multiplication and addition.

Product, sum, conjunction, disjunction

Because of this shorthand notation, it is common to refer to the AND of expressions as a *product* and to the OR of expressions as a *sum.* Another name for the

AND of expressions is a *conjunction*, and for the OR of expressions another name is *disjunction.*

Constructing a Logical Expression from a Truth Table

Any Boolean function whatsoever can be represented by a logical expression using the operators AND, OR, and NOT. Finding the simplest expression for a given Boolean function is generally hard. However, we can easily construct *some* expression for any Boolean function. The technique is straightforward. Starting with the truth table for the function, we construct a logical expression of the form

$$m_1 \texttt{ OR } m_2 \texttt{ OR } \cdots \texttt{ OR } m_n$$

Each m_i is a term that corresponds to one of the rows of the truth table for which the function has value 1. Thus there are as many terms in the expression as there are 1's in the column for that function. Each of the terms m_i is called a *minterm* and has a special form that we shall describe below.

Minterm

To begin our explanation of minterms, a *literal* is an expression that is either a single propositional variable, such as p, or a negated variable, such as NOT p, which we shall often write as $\bar{p}$. If the truth table has k variable columns, then each minterm consists of the logical AND, or "product," of k literals. Let r be a row for which we wish to construct the minterm. If the variable p has the value 1 in row r, then select literal p. If p has value 0 in row r, then select $\bar{p}$ as the literal. The minterm for row r is the product of the literals for each variable. Clearly, the minterm can only have the value 1 if all the variables have the values that appear in row r of the truth table.

Literal

Now construct an expression for the function by taking the logical OR, or "sum," of those minterms that correspond to rows with 1 as the value of the function. The resulting expression is in "sum of products" form, or *disjunctive normal form.* The expression is correct, because it has the value 1 exactly when there is a minterm with value 1; this minterm cannot be 1 unless the values of the variables correspond to the row of the truth table for that minterm, and that row has value 1.

Sum of products; disjunctive normal form

✦ **Example 12.9.** Let us construct a sum-of-products expression for the carry-out function d defined by the truth table of Fig. 12.8. The rows with value 1 are numbered 3, 5, 6, and 7. The minterm for row 3, which has $x = 0$, $y = 1$, and $c = 1$, is $\bar{x}$ AND y AND c, which we abbreviate $\bar{x}yc$. Similarly, the minterm for row 5 is $x\bar{y}c$, that for row 6 is $xy\bar{c}$, and that for row 7 is xyc. Thus the desired expression for d is the logical OR of these expressions, which is

$$\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc \tag{12.6}$$

This expression is more complex than (12.5). However, we shall see in the next section how expression (12.5) can be derived.

Similarly, we can construct a logical expression for the sum-bit z by taking the sum of the minterms for rows 1, 2, 4, and 7 to obtain

$$\bar{x}\bar{y}c + \bar{x}y\bar{c} + x\bar{y}\bar{c} + xyc$$

✦

Complete Sets of Operators

The minterm technique for designing sum-of-products expressions like (12.6) shows that the set of logical operators `AND`, `OR`, and `NOT` is a *complete* set, meaning that every Boolean function has an expression using just these operators. It is not hard to show that the `NAND` operator by itself is complete. We can express the functions `AND`, `OR`, and `NOT`, with `NAND` alone as follows:

1. $(p$ `AND` $q) \equiv \big((p$ `NAND` $q)$ `NAND` `TRUE`$\big)$
2. $(p$ `OR` $q) \equiv \big((p$ `NAND` `TRUE`$)$ `NAND` $(q$ `NAND` `TRUE`$)\big)$
3. $($`NOT` $p) \equiv (p$ `NAND` `TRUE`$)$

We can convert any sum-of-products expression to one involving only `NAND`, by substituting the appropriate `NAND`-expression for each use of `AND`, `OR`, and `NOT`. Similarly, `NOR` by itself is complete.

An example of a set of operators that is not complete is `AND` and `OR` by themselves. For example, they cannot express the function `NOT`. To see why, note that `AND` and `OR` are *monotone,* meaning that when you change any one input from 0 to 1, the output cannot change from 1 to 0. It can be shown by induction on the size of an expression that any expression with operators `AND` and `OR` is monotone. But `NOT` is not monotone, obviously. Hence there is no way to express `NOT` by `AND`'s and `OR`'s.

Monotone function

p	q	r	a	b
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Fig. 12.10. Two Boolean functions for exercises.

EXERCISES

12.5.1: Figure 12.10 is a truth table that defines two Boolean functions, a and b, in terms of variables p, q, and r. Write sum-of-products expressions for each of these functions.

12.5.2: Write product-of-sums expressions (see the box on "Product-of-Sums Expressions") for

a) Function a of Fig. 12.10.
b) Function b of Fig. 12.10.
c) Function z of Fig. 12.8.

Product-of-Sums Expressions

Conjunctive normal form

There is a dual way to convert a truth table into an expression involving AND, OR, and NOT; this time, the expression will be a product (logical AND) of sums (logical OR) of literals. This form is called "product-of-sums," or *conjunctive normal form.*

Maxterm

For each row of a truth table, we can define a *maxterm,* which is the sum of those literals that disagree with the value of one of the argument variables in that row. That is, if the row has value 0 for variable p, then use literal p, and if the value of that row for p is 1, then use $\bar{p}$. The value of the maxterm is thus 1 unless each variable p has the value specified for p by that row.

Thus, if we look at all the rows of the truth table for which the value is 0, and take the logical AND of the maxterms for all those rows, our expression will be 0 exactly when the inputs match one of the rows for which the function is to be 0. It follows that the expression has value 1 for all the other rows, that is, those rows for which the truth table gives the value 1. For example, the rows with value 0 for d in Fig. 12.8 are numbered 0, 1, 2, and 4. The maxterm for row 0 is $x + y + c$, and that for row 1 is $x + y + \bar{c}$, for example. The product-of-sums expression for d is

$$(x + y + c)(x + y + \bar{c})(x + \bar{y} + c)(\bar{x} + y + c)$$

This expression is equivalent to (12.5) and (12.6).

12.5.3**: Which of the following logical operators form a complete set of operators by themselves: (a) $\equiv$ (b) $\rightarrow$ (c) NOR? Prove your answer in each case.

12.5.4**: Of the 16 Boolean functions of two variables, how many are complete by themselves?

12.5.5*: Show that the AND and OR of monotone functions is monotone. Then show that any expression with operators AND and OR only, is monotone.

12.6 Designing Logical Expressions by Karnaugh Maps

In this section, we present a tabular technique for finding sum-of-products expressions for Boolean functions. The expressions produced are often simpler than those constructed in the previous section by the expedient of taking the logical OR of all the necessary minterms in the truth table.

For instance, in Example 12.7 we did an ad hoc design of an expression for the carry-out function of a one-bit adder. We saw that it was possible to use a product of literals that was not a minterm; that is, it was missing literals for some of the variables. For example, we used the product of literals xy to *cover* the sixth and seventh rows of Fig. 12.8, in the sense that xy has value 1 exactly when the variables x, y, and c have the values indicated by one of those two rows.

Similarly, in Example 12.7 we used the expression xc to cover rows 5 and 7, and we used yc to cover rows 3 and 7. Note that row 7 is covered by all three expressions. There is no harm in that. In fact, had we used only the minterms for rows 5 and 3, which are $x\bar{y}c$ and $\bar{x}yc$, respectively, in place of xc and yc, we would have obtained an expression that was correct, but that had two more occurrences of operators than the expression $xy + xc + yc$ obtained in Example 12.7.

The essential concept here is that if we have two minterms differing only by the negation of one variable, such as $xy\bar{c}$ and xyc for rows 6 and 7, respectively, we can combine the two minterms by taking the common literals and dropping the variable in which the terms differ. This observation follows from the general law

$$(pq + \bar{p}q) \equiv q$$

To see this equivalence, note that if q is true, then either pq is true, or $\bar{p}q$ is true, and conversely, when either pq or $\bar{p}q$ is true, then it must be that q is true.

We shall see a technique for verifying such laws in the next section, but, for the moment, we can let the intuitive meaning of our law justify its use. Note also that use of this law is not limited to minterms. We could, for example, let p be any propositional variable and q be any product of literals. Thus we can combine any two products of literals that differ only in one variable (one product has the variable itself and the other its complement), replacing the two products by the one product of the common literals.

Karnaugh Maps

There is a graphical technique for designing sum-of-products expressions from truth tables; the method works well for Boolean functions up to four variables. The idea is to write a truth table as a two-dimensional array called a *Karnaugh map* (pronounced "*car*-no") whose entries, or "points," each represent a row of the truth table. By keeping adjacent the points that represent rows differing in only one variable, we can "see" useful products of literals as certain rectangles, all of whose points have the value 1.

Two-Variable Karnaugh Maps

The simplest Karnaugh maps are for Boolean functions of two variables. The rows correspond to values of one of the variables, and the columns correspond to values of the other. The entries of the map are 0 or 1, depending on whether that combination of values for the two variables makes the function have value 0 or 1. Thus the Karnaugh map is a two-dimensional representation of the truth table for a Boolean function.

✦ **Example 12.10.** In Fig. 12.11 we see the Karnaugh map for the "implies" function, $p \rightarrow q$. There are four points corresponding to the four possible values for p and q. Note that "implies" has value 1 except when $p = 1$ and $q = 0$, and so the only point in the Karnaugh map with value 0 is the entry for $p = 1$ and $q = 0$; all the other points have value 1. ✦

Implicants

An *implicant* for a Boolean function f is a product x of literals for which no assignment of values to the variables of f makes x true and f false. For example, every minterm for which the function f has value 1 is an implicant of f. However, there are other products that can also be implicants, and we shall learn to read these off of the Karnaugh map for f.

		q = 0	q = 1
p	0	1	1
	1	0	1

Fig. 12.11. Karnaugh map for $p \rightarrow q$.

✦ **Example 12.11.** The minterm pq is an implicant for the "implies" function of Fig. 12.11, because the only assignment of values for p and q that makes pq true, namely $p = 1$ and $q = 1$, also makes the "implies" function true.

As another example, $\bar{p}$ by itself is an implicant for the "implies" function because the two assignments of values for p and q that make $\bar{p}$ true also make $p \rightarrow q$ true. These two assignments are $p = 0$, $q = 0$ and $p = 0$, $q = 1$. ✦

Covering points of a Karnaugh map

An implicant is said to *cover* the points for which it has the value 1. A logical expression can be constructed for a Boolean function by taking the OR of a set of implicants that together cover all points for which that function has value 1.

✦ **Example 12.12.** Figure 12.12 shows two implicants in the Karnaugh map for the "implies" function. The larger, which covers two points, corresponds to the single literal, $\bar{p}$. This implicant covers the top two points of the map, both of which have 1's in them. The smaller implicant, pq, covers the point $p = 1$ and $q = 1$. Since these two implicants together cover all the points that have value 1, their sum, $\bar{p} + pq$, is an equivalent expression for $p \rightarrow q$; that is, $(p \rightarrow q) \equiv (\bar{p} + pq)$. ✦

		q = 0	q = 1
p	0	1	1
	1	0	1

Fig. 12.12. Two implicants $\bar{p}$ and pq in the Karnaugh map for $p \rightarrow q$.

Rectangles that correspond to implicants in Karnaugh maps must have a special "look." For the simple maps that come from 2-variable functions, these rectangles can only be

1. Single points,
2. Rows or columns, or
3. The entire map.

A single point in a Karnaugh map corresponds to a minterm, whose expression we can find by taking the product of the literals for each variable appropriate to the row and column of the point. That is, if the point is in the row or column for 0, then we take the negation of the variable corresponding to the row, or column, respectively. If the point is in the row or column for 1, then we take the corresponding variable, unnegated. For instance, the smaller implicant in Fig. 12.12 is in the row for $p = 1$ and the column for $q = 1$, which is the reason that we took the product of the unnegated literals p and q for that implicant.

A row or column in a two-variable Karnaugh map corresponds to a pair of points that agree in one variable and disagree in the other. The corresponding "product" of literals reduces to a single literal. The remaining literal has the variable whose common value the points share. The literal is negated if that common value is 0, and unnegated if the shared value is 1. Thus the larger implicant in Fig. 12.12 — the first row — has points with a common value of p. That value is 0, which justifies the use of the product-of-literals $\bar{p}$ for that implicant.

An implicant consisting of the entire map is a special case. In principle, it corresponds to a product that reduces to the constant 1, or `TRUE`. Clearly, the Karnaugh map for the logical expression `TRUE` has 1's in all points of the map.

Prime Implicants

A *prime implicant* x for a Boolean function f is an implicant for f that ceases to be an implicant for f if any literal in x is deleted. In effect, a prime implicant is an implicant that has as few literals as possible.

Note that the bigger a rectangle is, the fewer literals there are in its product. We would generally prefer to replace a product with many literals by one with fewer literals, which involves fewer occurrences of operators, and thus is "simpler." We are thus motivated to consider only those implicants that are prime, when selecting a set of implicants to cover a map.

Remember that every implicant for a given Karnaugh map consists only of points with 1's. An implicant is a prime implicant because expanding it further by doubling its size would force us to cover a point with value 0.

◆ **Example 12.13.** In Fig. 12.12, the larger implicant $\bar{p}$ is prime, since the only possible larger implicant is the entire map, which cannot be used because it contains a 0. The smaller implicant pq is not prime, since it is contained in the second column, which consists only of 1's, and is therefore an implicant for the "implies" Karnaugh map. Figure 12.13 shows the only possible choice of prime implicants for the "implies" map.[1] They correspond to the products $\bar{p}$ and q, and they give rise to the expression $\bar{p} + q$, which we noted in Section 12.3 was equivalent to $p \rightarrow q$. ◆

[1] In general, there may be many sets of prime implicants that cover a given Karnaugh map.

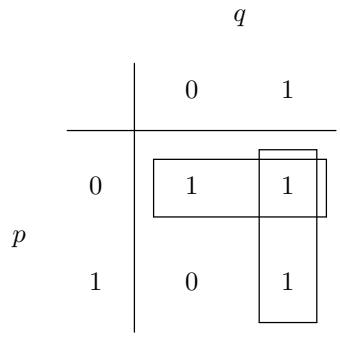


Fig. 12.13. Prime implicants $\bar{p}$ and q for the "implies" function.

Three-Variable Karnaugh Maps

When we have three variables in our truth table, we can use a two-row, four-column map like that shown in Fig. 12.14, which is a map for the carry-out truth table of Fig. 12.8. Notice the unusual order in which the columns correspond to pairs of values for two variables (variables y and c in this example). The reason is that we want adjacent columns to correspond to assignments of truth values that differ in only one variable. Had we chosen the usual order, 00, 01, 10, 11, the middle two columns would differ in both y and c. Note also that the first and last columns are "adjacent," in the sense that they differ only in variable y. Thus, when we select implicants, we can regard the first and last columns as a 2×2 rectangle, and we can regard the first and last points of either row as a 1×2 rectangle.

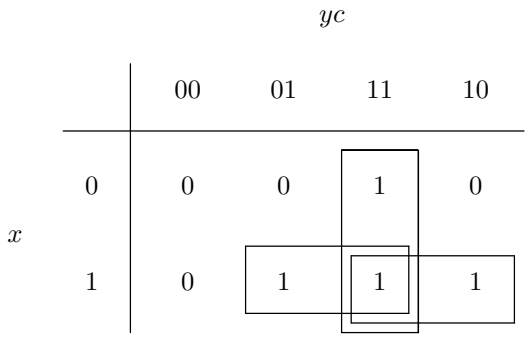


Fig. 12.14. Karnaugh map for the carry-out function with prime implicants xc, yc, and xy.

We need to deduce which rectangles of a three-variable map represent possible implicants. First, a permissible rectangle must correspond to a product of literals. In any product, each variable appears in one of three ways: negated, unnegated, or not at all. When a variable appears negated or unnegated, it cuts in half the number of points in the corresponding implicant, since only points with the proper value for that variable belong in the implicant. Hence, the number of points in an implicant will always be a power of 2. Each permissible implicant is thus a collection of points that, for each variable, either

Reading Implicants from the Karnaugh Map

No matter how many variables are involved, we can take any rectangle that represents an implicant and produce the product of literals that is `TRUE` for exactly the points in the rectangle. If p is any variable, then

1. If every point in the rectangle has $p = 1$, then p is a literal in the product.
2. If every point in the rectangle has $p = 0$, then $\bar{p}$ is a literal in the product.
3. If the rectangle has points with $p = 0$ and other points with $p = 1$, then the product has no literal with variable p.

a) Includes only points with that variable equal to 0,

b) Includes only points with that variable equal to 1, or

c) Does not discriminate on the basis of the value of that variable.

For three-variable maps, we can enumerate the possible implicants as follows.

1. Any point.
2. Any column.
3. Any pair of horizontally adjacent points, including the end-around case, that is, a pair in columns 1 and 4 of either row.
4. Any row.
5. Any 2×2 square consisting of two adjacent columns, including the end-around case, that is, columns 1 and 4.
6. The entire map.

✦ **Example 12.14.** The three prime implicants for the carry-out function were indicated in Fig. 12.14. We may convert each to a product of literals; see the box "Reading Implicants from the Karnaugh Map." The corresponding products are xc for the leftmost one, yc for the vertical one, and xy for the rightmost one. The sum of these three expressions is the sum-of-products that we obtained informally in Example 12.7; we now see how this expression was obtained. ✦

✦ **Example 12.15.** Figure 12.15 shows the Karnaugh map for the three-variable Boolean function `NAND` (p, q, r). The prime implicants are

1. The first row, which corresponds to $\bar{p}$.
2. The first two columns, which correspond to $\bar{q}$.
3. Columns 1 and 4, which correspond to $\bar{r}$.

The sum-of-products expression for this map is $\bar{p} + \bar{q} + \bar{r}$. ✦

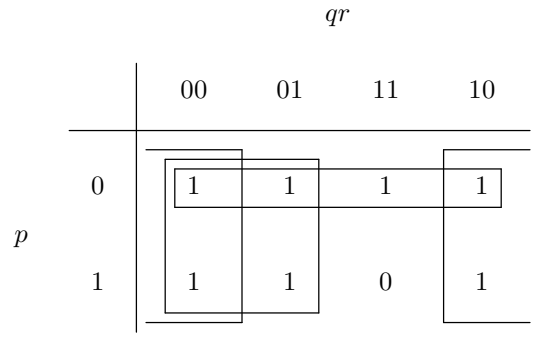


Fig. 12.15. Karnaugh map with prime implicants $\bar{p}$, $\bar{q}$, and $\bar{r}$ for $\texttt{NAND}(p, q, r)$.

Four-Variable Karnaugh Maps

A four-argument function can be represented by a 4×4 Karnaugh map, in which two variables correspond to the rows, and two variables correspond to the columns. In both the rows and columns, the special order of the values that we used for the columns in three-variable maps must be used, as shown in Fig. 12.16. For four-variable maps, adjacency of both rows and columns must be interpreted in the end-around sense. That is, the top and bottom rows are adjacent, and the left and right columns are adjacent. As an important special case, the four corner points form a 2×2 rectangle; they correspond in Fig. 12.16 to the product of literals $\bar{q}\bar{s}$ (which is not an implicant in Fig. 12.16, because the lower right corner is 0).

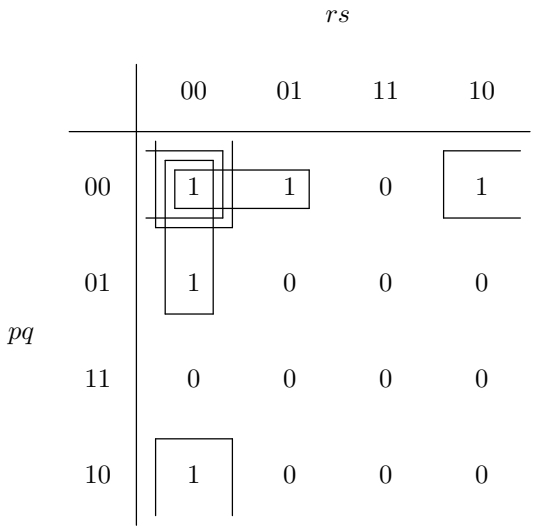


Fig. 12.16. Karnaugh map with prime implicants for the "at most one 1" function.

The rectangles in a four-variable Karnaugh map that correspond to products of literals are as follows:

1. Any point.
2. Any two horizontally or vertically adjacent points, including those that are adjacent in the end-around sense.
3. Any row or column.
4. Any 2×2 square, including those in the end-around sense, such as two adjacent points in the top row and the two points in the bottom row that are in the same columns. The four corners is, as we mentioned, a special case of a "square" as well.
5. Any 2×4 or 4×2 rectangle, including those in the end-around sense, such as the first and last columns.
6. The entire map.

◆ **Example 12.16.** Figure 12.16 shows the Karnaugh map of a Boolean function of four variables, p, q, r, and s, that has the value 1 when at most one of the inputs is 1. There are four prime implicants, all of size 2, and two of them are end-around. The implicant consisting of the first and last points of the top row has points that agree in variables p, q, and s; the common value is 0 for each variable. Thus its product of literals is $\bar{p}\bar{q}\bar{s}$. Similarly, the other implicants have products $\bar{p}\bar{q}\bar{r}$, $\bar{p}\bar{r}\bar{s}$, and $\bar{q}\bar{r}\bar{s}$. The expression for the function is thus

$$\bar{p}\bar{q}\bar{r} + \bar{p}\bar{q}\bar{s} + \bar{p}\bar{r}\bar{s} + \bar{q}\bar{r}\bar{s}$$

◆

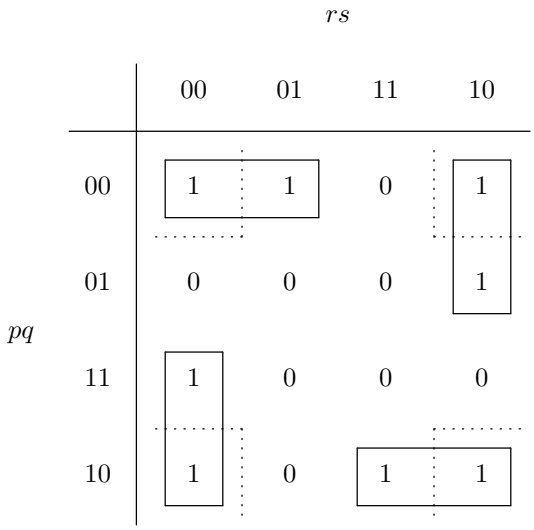


Fig. 12.17. Karnaugh map with an all-corners prime implicant.

◆ **Example 12.17.** The map of Fig. 12.17 was chosen for the pattern of its 1's, rather than for any significance its function has. It does illustrate an important point. Five prime implicants that together cover all the 1 points are shown, including the all-corners implicant (shown dashed), for which the product of literals expression is $\bar{q}\bar{s}$; the other four prime implicants have products $\bar{p}\bar{q}\bar{r}$, $\bar{p}r\bar{s}$, $p\bar{q}r$, and $p\bar{r}\bar{s}$.

We might think, from the examples seen so far, that to form the logical expression for this map we should take the logical `OR` of all five implicants. However, a moment's reflection tells us that the largest implicant, $\bar{q}\bar{s}$, is superfluous, since all its points are covered by other prime implicants. Moreover, this is the only prime implicant that we have the option to eliminate, since each other prime implicant has a point that only it covers. For example, $\bar{p}\bar{q}\bar{r}$ is the only prime implicant to cover the point in the first row and second column. Thus

$$\bar{p}\bar{q}\bar{r} + \bar{p}r\bar{s} + p\bar{q}r + p\bar{r}\bar{s}$$

is the preferred sum-of-products expression obtained from the map of Fig. 12.17. ◆

EXERCISES

12.6.1: Draw the Karnaugh maps for the following functions of variables p, q, r, and s.

a) The function that is `TRUE` if one, two, or three of p, q, r, and s are `TRUE`, but not if zero or all four are `TRUE`.
b) The function that is `TRUE` if up to two of p, q, r, and s are `TRUE`, but not if three or four are `TRUE`.
c) The function that is `TRUE` if one, three, or four of p, q, r, and s are `TRUE`, but not if zero or two are `TRUE`.
d) The function represented by the logical expression $pqr \rightarrow s$.
e) The function that is `TRUE` if $pqrs$, regarded as a binary number, has value less than ten.

12.6.2: Find the implicants — other than the minterms — for each of your Karnaugh maps from Exercise 12.6.1. Which of them are prime implicants? For each function, find a sum of prime implicants that covers all the 1's of the map. Do you need to use all the prime implicants?

12.6.3: Show that every product in a sum-of-products expression for a Boolean function is an implicant of that function.

12.6.4*: One can also construct a product-of-sums expression from a Karnaugh map. We begin by finding rectangles of the types that form implicants, but with all points 0, instead of all points 1. Call such a rectangle an "anti-implicant." We can construct for each anti-implicant a sum of literals that is 1 on all points but those of the anti-implicant. For each variable x, this sum has literal x if the anti-implicant includes only points for which $x = 0$, and it has literal $\bar{x}$ if the anti-implicant has only points for which $x = 1$. Otherwise, the sum does not have a literal involving x. Find all the prime anti-implicants for your Karnaugh maps of Exercise 12.6.1.

Anti-implicant

12.6.5: Using your answer to Exercise 12.6.4, write product-of-sums expressions for each of the functions of Exercise 12.6.1. Include as few sums as you can.

12.6.6**: How many (a) 1×2 (b) 2×2 (c) 1×4 (d) 2×4 rectangles that form implicants are there in a 4×4 Karnaugh map? Describe their implicants as products of literals, assuming the variables are p, q, r, and s.

12.7 Tautologies

A *tautology* is a logical expression whose value is true regardless of the values of its propositional variables. For a tautology, all the rows of the truth table, or all the points in the Karnaugh map, have the value 1. Simple examples of tautologies are

`TRUE`
$p + \bar{p}$
$(p + q) \equiv (p + \bar{p}q)$

Tautologies have many important uses. For example, suppose we have an expression of the form $E_1 \equiv E_2$ that is a tautology. Then, whenever we have an instance of E_1 within any expression, we can replace E_1 by E_2, and the resulting expression will represent the same Boolean function.

Figure 12.18(a) shows the expression tree for a logical expression F containing E_1 as a subexpression. Figure 12.18(b) shows the same expression with E_1 replaced by E_2. If $E_1 \equiv E_2$, the values of the roots of the two trees must be the same, no matter what assignment of truth values is made to the variables. The reason is that we know the nodes marked n in the two trees, which are the roots of the expression trees for E_1 and E_2, must get the same value in both trees, because $E_1 \equiv E_2$. The evaluation of the trees above n will surely yield the same value, proving that the two trees are equivalent. The ability to substitute equivalent expressions for one another is colloquially known as the "substitution of equals for equals." Note that in other algebras, such as those for arithmetic, sets, relations, or regular expressions we also may substitute one expression for another that has the same value.

Substitution of equals for equals

(a) Expression containing E_1 (b) Expression containing E_2

Fig. 12.18. Expression trees showing substitution of equals for equals.

◆ **Example 12.18.** Consider the associative law for the logical operator `OR`, which can be phrased as the expression

$$\big((p + q) + r\big) \equiv \big(p + (q + r)\big) \tag{12.7}$$

The truth table for the various subexpressions appears in Fig. 12.19. The final column, labeled E, represents the entire expression. Observe that every row has value 1 for E, showing that the expression (12.7) is a tautology. As a result, any time we see an expression of the form $(p + q) + r$, we are free to replace it by $p + (q + r)$. Note that p, q, and r can stand for any expressions, as long as the same expression is used for both occurrences of p, and q and r are likewise treated consistently. ✦

p	q	r	$p+q$	$(p+q)+r$	$q+r$	$p+(q+r)$	E
0	0	0	0	0	0	0	1
0	0	1	0	1	1	1	1
0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1
1	0	1	1	1	1	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1

Fig. 12.19. Truth table proving the associative law for `OR`.

The Substitution Principle

As we pointed out in Example 12.18, when we have a law involving a particular set of propositional variables, the law applies not only as written, but with any substitution of an expression for each variable. The underlying reason is that tautologies remain tautologies when we make any substitution for one or more of its variables. This fact is known as the *substitution principle*.[2] Of course, we must substitute the same expression for each occurrence of a given variable.

✦ **Example 12.19.** The commutative law for the logical operator `AND` can be verified by showing that the logical expression $pq \equiv qp$ is a tautology. To get some instances of this law, we can perform substitutions on this expression. For example, we could substitute $r + s$ for p and $\bar{r}$ for q to get the equivalence

$$(r + s)(\bar{r}) \equiv (\bar{r})(r + s)$$

Note that we put parentheses around each substituted expression to avoid accidentally changing the grouping of operators because of our operator-precedence conventions. In this case, the parentheses around $r + s$ are essential, but the parentheses around $\bar{r}$ could be omitted.

Some other substitution instances follow. We could replace p by r and not replace q, to get $rq \equiv qr$. We could leave p alone and replace q by the constant expression 1 (`TRUE`), to get p `AND` $1 \equiv 1$ `AND` p. However, we cannot substitute r for the first occurrence of p and substitute a different expression, say $r + s$, for the

[2] We should not confuse the substitution principle with the "substitution of equals for equals." The substitution principle applies to tautologies only, while we may substitute equals for equals in any expression.

second. That is, $rq \equiv q(r+s)$ is not a tautology (its value is 0 if $s = q = 1$ and $r = 0$). ✥

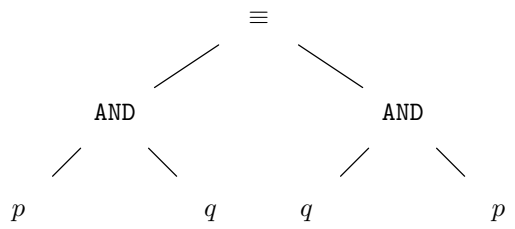


Fig. 12.20. Expression tree for the tautology $pq \equiv qp$.

The reason the substitution principle holds true can be seen if we think about expression trees. Imagine the expression tree for some tautology, such as the one discussed in Example 12.19, which we show in Fig. 12.20. Since the expression is a tautology, we know that, whatever assignment of truth values we make for the propositional variables at the leaves, the value at the root is true (as long as we assign the same value to each leaf that is labeled by a given variable).

Now suppose that we substitute for p an expression with tree T_p and that we substitute for q an expression with tree T_q; in general, we select one tree for each variable of the tautology, and replace all leaves for that variable by the tree selected for that variable.[3] Then we have a new expression tree similar to that suggested by Fig. 12.21. When we make an assignment of truth values for the variables of the new tree, the value of each node that is a root of a tree T_p gets the same value, because the same evaluation steps are performed underneath any such node.

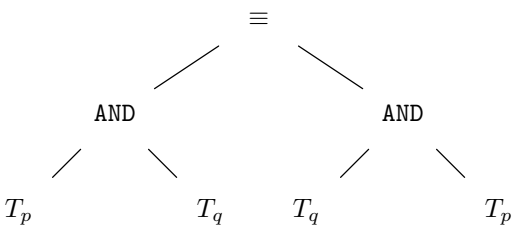


Fig. 12.21. A substitution for the variables of Fig. 12.20.

Once the roots of the trees like T_p and T_q in Fig. 12.21 are evaluated, we have a consistent assignment of values to the variables at the leaves of the original tree, which we illustrated in Fig. 12.20. That is, we take whatever value is computed for the occurrences of T_p, which must all be the same value, and assign it to all

[3] As a special case, the tree selected for some variable x can be a single node labeled x, which is the same as making no substitution for x.

the leaves labeled p in the original tree. We do the same for q, and in general, for any variable appearing in the original tree. Since the original tree represents a tautology, we know that evaluating that tree will result in the value `TRUE` at the root. But above the substituted trees, the new and original trees are the same, and so the new tree also produces value `TRUE` at the root. Since the above reasoning holds true no matter what substitution of values we make for the variables of the new tree, we conclude that the expression represented by the new tree is also a tautology.

The Tautology Problem

The tautology problem is to test whether a given logical expression is equivalent to `TRUE`, that is, whether it is a tautology. There is a straightforward way to solve this problem. Construct a truth table with one row for each possible assignment of truth values to the variables of the expression. Create one column for each interior node of the tree, and in a suitable bottom-up order, evaluate each node for each assignment of truth values to the variables. The expression is a tautology if and only if the value of the whole expression is 1 (`TRUE`) for every truth assignment. Example 12.18 illustrated this process.

Running Time of the Tautology Test

If the expression has k variables and n operator occurrences, then the table has 2^k rows, and there are n columns that need to be filled out. We thus expect a straightforward implementation of this algorithm to take $O(2^k n)$ time. That is not long for expressions with two or three variables, and even for, say, 20 variables, we can carry the test out by computer in a few seconds or minutes. However, for 30 variables, there are a billion rows, and it becomes far less feasible to carry out this test, even using a computer. These observations are typical of what happens when one uses an exponential-time algorithm. For small instances, we generally see no problem. But suddenly, as problem instances get larger, we find it is no longer possible to solve the problem, even with the fastest computers, in an amount of time we can afford.

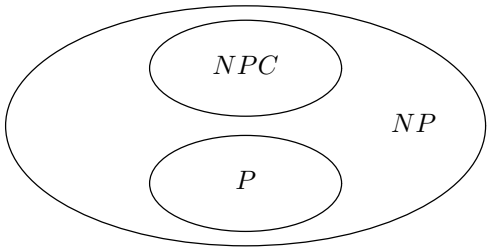


Fig. 12.22. P is the family of problems solvable in polynomial time, NP is the family solvable in nondeterministic polynomial time, and NPC is the family of NP-complete problems.

EXERCISES

12.7.1: Which of the following expressions are tautologies?

Inherent Intractability

The tautology problem, "Is E a tautology," is an important example of a problem that appears to be inherently exponential. That is, if k is the number of variables in expression E, all the known algorithms to solve the tautology problem have a running time that is an exponential function of k.

NP-complete problem

There is a family of problems, called *NP-complete*, which includes many important optimization problems that no one knows how to solve in less than exponential time. Many mathematicians and scientists have worked long and hard trying to find for at least one of these problems an algorithm that runs in less than exponential time, but no such algorithm has yet been found, and so many people now suspect that none exists.

Satisfiability problem

One of the classic NP-complete problems is the *satisfiability problem*: "Is there a truth assignment that makes logical expression E true?" Satisfiability is closely related to the tautology problem, and as with that problem, no significantly better solution to the satisfiability problem is known than cycling through all possible truth assignments.

Either all the NP-complete problems have less-than-exponential time solutions, or none do. The fact that each NP-complete problem appears to require exponential time thus reinforces our belief that all are inherently exponential-time problems. Thus we have strong evidence that the straightforward satisfiability test is about the best we can do.

Incidentally, "NP" stands for "nondeterministic polynomial." "Nondeterministic" informally means "the ability to guess right," as discussed in Section 10.3. A problem can be "solved in nondeterministic polynomial time" if, given a guess at a solution for some instance of size n, we can check that the guess is correct in polynomial time, that is, in time n^c for some constant c.

Satisfiability is an example of such a problem. If someone gave us an assignment of truth values to variables that they claimed, or guessed, made expression E evaluate to 1, we could evaluate E with that assignment to its operands, and check, in time at most quadratic in the length of E, that the expression is satisfiable.

The class of problems that — like satisfiability — can be "solved" by guessing followed by a polynomial time check is called NP. Some problems in NP are actually quite easy, and can be solved without the guessing, still taking only time that is polynomial in the length of the input. However, there are many problems in NP that can be proved to be as hard as any in NP, and these are the NP-complete problems. (Do not confuse "completeness" in this sense, meaning "hardest in the class," with "complete set of operators" meaning "able to express every Boolean function.")

The family of problems solvable in polynomial time with no guessing is often called P. Figure 12.22 shows the relationship between P, NP, and the NP-complete problems. If any NP-complete problem is in P, then $P = NP$, something we doubt very much is the case, because all the known NP-complete problems, and some other problems in NP, appear not to be in P. The tautology problem is not believed to be in NP, but it is as hard or harder than any problem in NP (called an *NP-hard* problem) and if the tautology problem is in P, then $P = NP$.

NP-hard problem

a) $pqr \rightarrow p + q$
b) $\big((p \rightarrow q)(q \rightarrow r)\big) \rightarrow (p \rightarrow r)$
c) $(p \rightarrow q) \rightarrow p$
d) $\big(p \equiv (q + r)\big) \rightarrow (\bar{q} \rightarrow pr)$

12.7.2*: Suppose we had an algorithm to solve the tautology problem for a logical expression. Show how this algorithm could be used to

a) Determine whether two expressions were equivalent.
b) Solve the satisfiability problem (see the box on "Inherent Intractability").

12.8 Some Algebraic Laws for Logical Expressions

In this section, we shall enumerate some useful tautologies. In each case, we shall state the law, leaving the tautology test to be carried out by the reader by constructing the truth table.

Laws of Equivalence

We begin with some observations about how equivalence works. The reader should notice the dual role played by equivalence. It is one of a number of operators that we use in logical expressions. However, it is also a signal that two expressions are "equal," and that one can be substituted for the other. Thus a tautology of the form $E_1 \equiv E_2$ tells us something about E_1 and E_2, namely that either can be substituted for the other within larger expressions, using the principle "equals can be substituted for equals."

Further, we can use equivalences to prove other equivalences. If we have a sequence of expressions $E_1, E_2, \ldots, E_k$, such that each is derived from the previous one by a substitution of equals for equals, then each of these expressions gives the same value when evaluated with the same truth assignment. As a consequence, $E_1 \equiv E_k$ must be a tautology.

12.1. *Reflexivity of equivalence*: $p \equiv p$.

As with all the laws we state, the principle of substitution applies, and we may replace p by any expression. Thus this law says that any expression is equivalent to itself.

12.2. *Commutative law for equivalence*: $(p \equiv q) \equiv (q \equiv p)$.

Informally, p is equivalent to q if and only if q is equivalent to p. By the principle of substitution, if any expression E_1 is equivalent to another expression E_2, then E_2 is equivalent to E_1. Thus either of E_1 and E_2 may be substituted for the other.

12.3. *Transitive law for equivalence*: $\big((p \equiv q)\ \texttt{AND}\ (q \equiv r)\big) \rightarrow (p \equiv r)$.

Informally, if p is equivalent to q, and q is equivalent to r, then p is equivalent to r. An important consequence of this law is that if we have found both $E_1 \equiv E_2$ and $E_2 \equiv E_3$ to be tautologies, then $E_1 \equiv E_3$ is a tautology.

12.4. *Equivalence of the negations*: $(p \equiv q) \equiv (\bar{p} \equiv \bar{q})$.

Two expressions are equivalent if and only if their negations are equivalent.

Laws Analogous to Arithmetic

There is an analogy between the arithmetic operators +, ×, and unary minus on the one hand, and `OR`, `AND`, and `NOT` on the other. Thus the following laws should not be surprising.

12.5. *The commutative law for* `AND`: $pq \equiv qp$.

Informally, pq is true exactly when qp is true.

12.6. *The associative law for* `AND`: $p(qr) \equiv (pq)r$.

Informally, we can group the `AND` of three variables (or expressions) either by taking the `AND` of the first two initially, or by taking the `AND` of the last two initially. Moreover, with law 12.5, we can show that the `AND` of any collection of propositions or expressions can be permuted and grouped any way we wish — the result will be the same.

12.7. *The commutative law for* `OR`: $(p + q) \equiv (q + p)$.

12.8. *The associative law for* `OR`: $\big(p + (q + r)\big) \equiv \big((p + q) + r\big)$.

This law and law 12.7 tell us the `OR` of any set of expressions can be grouped as we like.

12.9. *The distributive law of* `AND` *over* `OR`: $p(q + r) \equiv (pq + pr)$.

That is, if we wish to take the `AND` of p and the `OR` of two propositions or expressions, we can either take the `OR` first, or take the `AND` of p with each expression first; the result will be the same.

12.10. 1 (`TRUE`) *is the identity for* `AND`: (p `AND` 1) $\equiv p$.

Notice that (1 `AND` p) $\equiv p$ is also a tautology. We did not need to state it, because it follows from the substitution principle and previous laws. That is, we may substitute simultaneously 1 for p and p for q in 12.5, the commutative law for `AND`, to get the tautology (1 `AND` p) $\equiv$ (p `AND` 1). Then, an application of 12.3, the transitivity of equivalence, tells us that (1 `AND` p) $\equiv p$.

12.11. 0 (`FALSE`) *is the identity for* `OR`: p `OR` 0 $\equiv p$.

Similarly, we can deduce that (0 `OR` p) $\equiv p$, using the same argument as for 12.10.

12.12. 0 *is the annihilator for* `AND`: (p `AND` 0) $\equiv 0$.[4]

Recall from Section 10.7 that an annihilator for an operator is a constant such that the operator, applied to that constant and any value, produces the annihilator as value. Note that in arithmetic, 0 is an annihilator for ×, but + has no annihilator. However, we shall see that 1 is an annihilator for `OR`.

12.13. *Elimination of double negations*: (`NOT` `NOT` p) $\equiv p$.

---

[4] Of course, (0 `AND` p) $\equiv 0$ holds as well. We shall not, in the future, mention all the consequences of the commutative laws.

Exploiting Analogies for Arithmetic and Logical Operators

When we use the shorthand notation for AND and OR, we can often pretend that we are dealing with multiplication and addition, as we use laws 12.5 through 12.12. That is an advantage, since we are quite familiar with the corresponding laws for arithmetic. Thus, for example, the reader should be able to replace $(p + q)(r + s)$ by $pr + ps + qr + qs$ or by $q(s + r) + (r + s)p$ quickly.

What is more difficult, and what requires practice, is applying the laws that are not analogous to arithmetic. Examples are DeMorgan's laws and the distribution of OR over AND. For example, replacing $pq+rs$ by $(p+r)(p+s)(q+r)(q+s)$ is valid, but requires some thought to see how it follows from three applications of 12.14, the distributive law of OR over AND, and commutative and associative laws.

Ways in Which AND and OR Differ from Plus and Times

There are also a number of laws that show the difference between AND and OR on the one hand, and the arithmetic operators $\times$ and $+$ on the other. We enumerate some of them here.

12.14. *The distributive law for* OR *over* AND: $(p + qr) \equiv \big((p + q)(p + r)\big)$.

Just as AND distributes over OR, OR distributes over AND. Note that the analogous arithmetic identity, $x + yz = (x + y)(x + z)$, is false in general.

12.15. 1 *is the annihilator for* OR: $(1 \texttt{ OR } p) \equiv 1$.

Note that the arithmetic analog $1 + x = 1$ is false in general.

12.16. *Idempotence of* AND: $pp \equiv p$.

Recall that an operator is idempotent if, when applied to two copies of the same value, it produces that value as result.

12.17. *Idempotence of* OR: $p + p \equiv p$.

Note that neither $\times$ nor $+$ is idempotent. That is, neither $x \times x = x$ nor $x + x = x$ is true in general.

12.18 *Subsumption.*

There are two versions of this law, depending on whether we remove a superfluous product or sum.

a) $(p + pq) \equiv p$.

b) $p(p + q) \equiv p$.

Note that if we substitute an arbitrary product of literals for p and another product of literals for q in (a), we are saying that in a sum of products, we can eliminate any product that has a superset of the literals of some other product. The smaller set is said to *subsume* the superset. In part (b) we are saying an analogous thing about a product of sums; we can eliminate a sum that is a superset of the literals of some other sum in the product.

12.19. *Elimination of certain negations.*

a) $p(\bar{p} + q) \equiv pq$.

b) $p + \bar{p}q \equiv p + q$.

Notice that (b) is the law that we used in Section 12.2 to explain why Sally's condition could replace Sam's.

DeMorgan's Laws

There are two laws that allow us to push NOT's through an expression of AND's and OR's, resulting in an expression in which all the negations apply to propositional variables. The resulting expression is an AND-OR expression applied to literals. Intuitively, if we negate an expression with AND's and OR's, we can push the negation down the expression tree, "flipping" operators as we go. That is, each AND becomes an OR, and vice versa. Finally, the negations reach the leaves, where they stay, unless they meet a negated literal, in which case we can remove two negations by law 12.13. We must be careful, when we construct the new expression, to place parentheses properly, because the precedence of operators changes when we exchange AND's and OR's.

The basic rules are called "DeMorgan's laws." They are the following two tautologies.

12.20. *DeMorgan's laws.*

a) $\texttt{NOT}\ (pq) \equiv \bar{p} + \bar{q}$.

b) $\texttt{NOT}\ (p + q) \equiv \bar{p}\bar{q}$.

Part (a) says that p and q are not both true exactly when at least one of them is false, and (b) says that neither p nor q is true if and only if they are both false. We can generalize these two laws to allow any number of propositional variables as follows.

c) $\big(\texttt{NOT}\ (p_1 p_2 \cdots p_k)\big) \equiv (\bar{p_1} + \bar{p_2} + \cdots + \bar{p_k})$.

d) $\big(\texttt{NOT}\ (p_1 + p_2 + \cdots + p_k)\big) \equiv (\bar{p_1}\bar{p_2} \cdots \bar{p_k})$.

For example, (d) says that none of some collection of expressions is true if and only if all of them are false.

✦ **Example 12.20.** We have seen in Sections 12.5 and 12.6 how to construct sum-of-products expressions for arbitrary logical expressions. Suppose we start with an arbitrary such expression E, which we may write as $E_1 + E_2 + \cdots + E_k$, where each E_i is the AND of literals. We can construct a product-of-sums expression for NOT E, by starting with

$$\texttt{NOT}\ (E_1 + E_2 + \cdots + E_k)$$

and applying DeMorgan's law (d) to get

$$\big(\texttt{NOT}\ (E_1)\big)\big(\texttt{NOT}\ (E_2)\big) \cdots \big(\texttt{NOT}\ (E_k)\big) \tag{12.8}$$

Now let E_i be the product of literals $X_{i1}X_{i2} \cdots X_{ij_i}$, where each X is either a variable or its negation. Then we can apply (c) to NOT (E_i) to turn it into

$$\bar{X}_{i1} + \bar{X}_{i2} + \cdots + \bar{X}_{ij_i}$$

If some literal X is a negated variable, say $\bar{q}$, then $\bar{X}$ should be replaced by q itself, using law 12.13, which says that double negations can be eliminated. When we make all these changes, (12.8) becomes a product of sums of literals.

For example, $rs + \bar{r}\bar{s}$ is a sum-of-products expression that is true exactly when $r \equiv s$; that is, it can be thought of as a definition of equivalence using `AND`, `OR`, and `NOT`. The following formula, the negation of the one above, is true when r and s are inequivalent, that is, exactly one of r and s is true.

$$\texttt{NOT}\ (rs + \bar{r}\bar{s}) \tag{12.9}$$

Now let us apply a substitution to DeMorgan's law (b), in which p is replaced by rs and q is replaced by $\bar{r}\bar{s}$. Then the left-hand side of (b) becomes exactly (12.9), and we know by the principle of substitution that (12.9) is equivalent to the right-hand side of (b) with the same substitution, namely

$$\texttt{NOT}\ (rs)\ \texttt{AND}\ \texttt{NOT}\ (\bar{r}\bar{s}) \tag{12.10}$$

Now we can apply (a), with the substitution of r for p and s for q, replacing `NOT` (rs) by $\bar{r} + \bar{s}$. Similarly, (a) tells us that `NOT` $(\bar{r}\bar{s})$ is equivalent to `NOT` $(\bar{r})$+ `NOT` $(\bar{s})$. But `NOT` $(\bar{r})$ is the same as `NOT` $\big($`NOT` $(r)\big)$, which is equivalent to r, since double negations can be eliminated. Similarly, `NOT` $(\bar{s})$ can be replaced by s. Thus (12.10) is equivalent to $(\bar{r} + \bar{s})(r + s)$. This is a product-of-sums expression for "exactly one of r and s is true." Informally, it says, "At least one of r and s is false and at least one of r and s is true." Evidently, the only way that could happen is for exactly one of r and s to be true. ✥

The Principle of Duality

As we scan the laws of this section, we notice a curious phenomenon. The equivalences seem to come in pairs, in which the roles of `AND` and `OR` are interchanged. For example, parts (a) and (b) of law 12.19 are such a pair, and laws 12.9 and 12.14 are such a pair; the latter are the two distributive laws. When the constants 0 and 1 are involved, these two must be interchanged, as in the pair of laws about identities, 12.10 and 12.11.

The explanation for this phenomenon is found in DeMorgan's laws. Suppose we start with a tautology $E_1 \equiv E_2$, where E_1 and E_2 are expressions involving operators `AND`, `OR`, and `NOT`. By law 12.4, `NOT` $(E_1) \equiv$ `NOT` (E_2) is also a tautology. Now we apply DeMorgan's laws to push the negations through `AND`'s and `OR`'s. As we do, we "flip" each `AND` to an `OR` and vice versa, and we move the negation down to each of the operands. If we meet a `NOT` operator, we simply move the "traveling" `NOT` below it, until we come to another `AND` or `OR`. The exception occurs when we come to a negated literal, say $\bar{p}$. Then, we combine the traveling `NOT` with the one already there to leave operand p. As a special case, if a traveling `NOT` meets a constant, 0 or 1, we negate the constant; that is, (`NOT` 0) $\equiv$ 1 and (`NOT` 1) $\equiv$ 0.

✥ **Example 12.21.** Let us consider the tautology 12.19(b). We begin by negating both sides, which gives us the tree of Fig. 12.23(a). Then, we push the negations through the `OR`'s on each side of the equivalence, changing them to `AND`'s; `NOT` signs appear above each of the arguments of the two `OR`'s, as shown in Fig. 12.23(b). Three of the new `NOT`'s are above variables, and so their travels end. The one that

is above an AND flips it to an OR, and causes NOT's to appear on its two arguments. The right-hand argument becomes NOT q, while the left-hand argument, which was NOT p, becomes NOT NOT p, or simply p. The resulting tree is shown in Fig. 12.23(c).

The tree of Fig. 12.23(c) represents the expression $\bar{p}(p + \bar{q}) \equiv \bar{p}\bar{q}$. To get the expression into the form of law 12.19(a), we must negate the variables. That is, we substitute expression $\bar{p}$ for p and $\bar{q}$ for q. When we eliminate the double negations, we are left with exactly 12.19(a). ✦

Laws Involving Implication

There are several useful tautologies that give us properties of the $\rightarrow$ operator.

12.21. $\big((p \rightarrow q)$ AND $(q \rightarrow p)\big) \equiv (p \equiv q)$.

That is, two expressions are equivalent if and only if they each imply the other.

12.22. $(p \equiv q) \rightarrow (p \rightarrow q)$.

The equivalence of two expressions tells us that either one implies the other.

12.23. *Transitivity of implication*: $\big((p \rightarrow q)$ AND $(q \rightarrow r)\big) \rightarrow (p \rightarrow r)$.

That is, if p implies q, which implies r, then p implies r.

12.24. It is possible to express implication with AND and OR. The simplest form is:

a) $(p \rightarrow q) \equiv (\bar{p} + q)$.

We shall see that there are many situations in which we deal with an expression of the form "if this and this and $\cdots$, then that." For example, the programming language Prolog, and many "artificial intelligence" languages depend upon "rules" of that form. These rules are written formally as $(p_1 p_2 \cdots p_n) \rightarrow q$. They may be expressed with only AND and OR, by the equivalence

b) $(p_1 p_2 \cdots p_n \rightarrow q) \equiv (\bar{p_1} + \bar{p_2} + \cdots + \bar{p_n} + q)$.

That is, both the left-hand and the right-hand sides of the equivalence are true whenever q is true or one or more of the p's are false; both sides are false otherwise.

EXERCISES

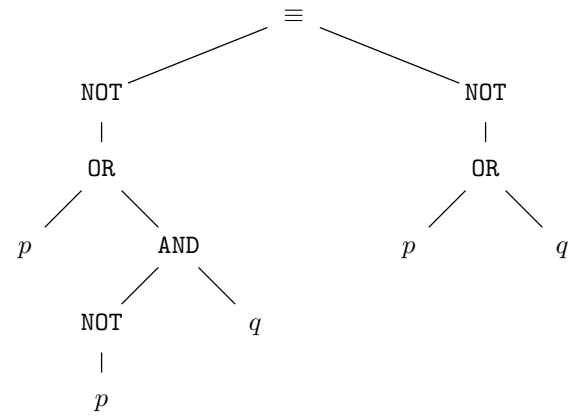
12.8.1: Check, by constructing the truth tables, that each of the laws 12.1 to 12.24 are tautologies.

12.8.2: We can substitute expressions for any propositional variable in a tautology and get another tautology. Substitute $x+y$ for p, yz for q, and $\bar{x}$ for r in each of the tautologies 12.1 to 12.24, to get new tautologies. Do not forget to put parentheses around the substituted expressions if needed.

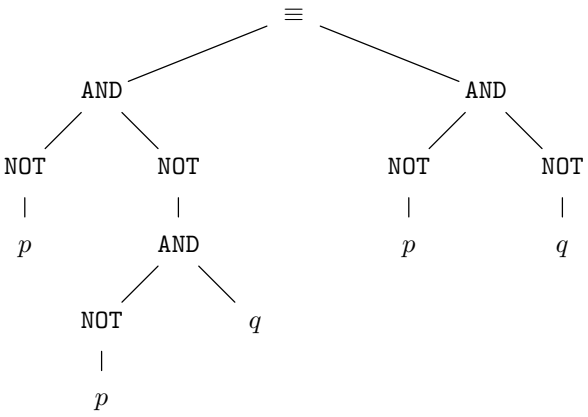
12.8.3: Prove that

a) $p_1 + p_2 + \cdots + p_n$ is equivalent to the sum (logical OR) of the p_i's in any order.
b) $p_1 p_2 \cdots p_n$ is equivalent to the product (logical AND) of the p_i's in any order.

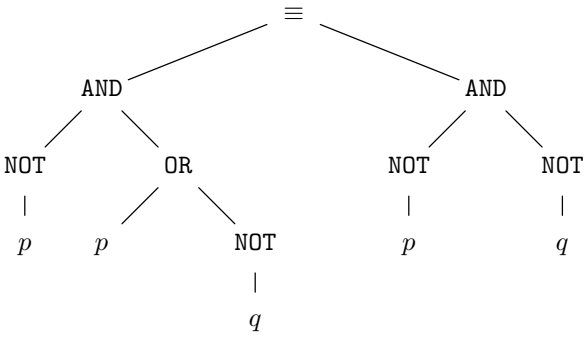
Hint: A similar result was shown for addition in Section 2.4.



(a) Initial expression tree



(b) First "pushes" of the negations



(c) Final expression

Fig. 12.23. Constructing the dual expression.

12.8.4*: Use laws given in this section to transform the first of each pair of expressions into the second. To save effort, you may omit steps that use laws 12.5 through 12.13, which are analogous to arithmetic. For example, commutativity and associativity of `AND` and `OR` may be assumed.

a) Transform $pq + rs$ into $(p + r)(p + s)(q + r)(q + s)$.
b) Transform $pq + p\bar{q}r$ into $p(q + r)$.
c) Transform $pq + p\bar{q} + \bar{p}q + \bar{p}\bar{q}$ into 1. (This transformation requires law 12.25 from the next section.)
d) Transform $pq \rightarrow r$ into $(p \rightarrow r) + (q \rightarrow r)$.
e) Transform `NOT` $(pq \rightarrow r)$ into $pq\bar{r}$.

12.8.5*: Show that the subsumption laws, 12.18(a) and (b), follow from previously given laws, in the sense that it is possible to transform $p + pq$ into p and transform $p(p + q)$ into p using only laws 12.1 through 12.17.

12.8.6: Apply DeMorgan's laws to turn the following expressions into expressions where the only `NOT`'s are applied to propositional variables (i.e., the `NOT`'s appear in literals only).

a) `NOT` $(pq + \bar{p}r)$
b) `NOT` (`NOT` $p + q$(`NOT` $(r + \bar{s})$))

12.8.7*: Prove the generalized DeMorgan's laws 12.20(c) and (d) by induction on k, using the basic laws 12.20(a) and (b). Then, justify the generalized laws informally by describing what the 2^k-row truth tables for each expression and their subexpressions look like.

12.8.8*: Find the pairs of laws in this section that are duals of one another.

12.8.9*: Prove law 12.24(b) by induction on n.

12.8.10*: Show that law 12.24(b) holds by describing the 2^n rows of the truth table for the expression and each of its subexpressions.

12.8.11: Simplify the following by using the subsumption laws and the commutative and associative laws for `AND` and `OR`.

a) $w\bar{x} + w\bar{x}y + \bar{z}\bar{x}w$
b) $(w + \bar{x})(w + y + \bar{z})(\bar{w} + \bar{x} + \bar{y})(\bar{x})$

12.8.12*: Show that the arithmetic analogs of laws 12.14 through 12.20 are false, by giving specific numbers for which the analogous equalities to not hold.

12.8.13*: If we start with logical expression whose only operators are `AND`, `OR`, and `NOT`, we can push all the `NOT`'s down the tree until the only `NOT`'s are immediately above propositions; that is, the expression is the `AND` and `OR` of literals. Prove that we can do so. *Hint*: Whenever we see a `NOT`, either it is immediately above another `NOT` (in which case we can eliminate them both by rule 12.13), or above a proposition (in which case the statement is satisfied), or it is above an `AND` and `OR` (in which case we can use DeMorgan's laws to push it down one level). However, a proof that we eventually reach an equivalent expression with all `NOT`'s above propositions cannot proceed by induction on an obvious "size" measure such as the sum of the heights of the nodes labeled `NOT`. The reason is that when we use DeMorgan's laws to push one `NOT` down, it is replaced by two `NOT`'s, and this sum might increase. In order

to prove that we eventually reach an expression with all NOT's above propositions, you need to find a suitable "size" measure that always decreases when DeMorgan's laws are applied in the direction where a NOT is pushed down below an AND or OR. Find such a size measure and prove the claim.

12.9 Tautologies and Methods of Proof

In the past three sections, we have seen one aspect of logic: its use as a design theory. In Section 12.6 we saw how to use Karnaugh maps to design expressions given a Boolean function, and in Chapter 13 we shall see how this methodology helps design the switching circuits from which computers and other digital devices are built. Sections 12.7 and 12.8 introduced us to tautologies, which can be used to simplify expressions, and therefore serve as another important tool when good expressions must be designed for a given Boolean function.

A second important use of logic will be seen in this section. When people reason or prove statements of mathematics, they use a variety of techniques to further their arguments. Examples of these techniques are

1. Case analysis,
2. Proof of the contrapositive,
3. Proof by contradiction, and
4. Proof by reduction to truth.

In this section we shall define these techniques, showing how each can be used in proofs. We also show how these techniques are justified by certain tautologies of propositional logic.

The Law of Excluded Middle

We begin with several tautologies that represent basic facts about how one reasons.

12.25. *The law of the excluded middle*: $(p + \bar{p}) \equiv 1$ is a tautology.

That is, something is either true or false; there is no middle ground.

Example 12.22. As an application of law 12.25, as well as several of the other laws seen so far, we can prove the law $(pq + \bar{p}q) \equiv q$ used in Section 12.6. Begin with

$$(1 \texttt{ AND } q) \equiv (1 \texttt{ AND } q)$$

which follows from law 12.1, reflexivity of equivalence, by substituting 1 AND q for p. Now, by law 12.25, we may replace 1 by $p + \bar{p}$ in the left-hand side above, substituting "equals for equals." Thus

$$\big((p + \bar{p})q\big) \equiv (1 \texttt{ AND } q)$$

is a tautology. On the right-hand side of the equivalence, use law 12.10 to replace 1 AND q by q. Then, on the left-hand side, we use 12.9, the distributivity of AND over OR, preceded and followed by law 12.5, the commutativity of AND, to show that the left-hand side is equivalent to $pq + \bar{p}q$. Thus we have

$$(pq + \bar{p}q) \equiv q$$

as desired. ✦

A generalization of the law of the excluded middle is a technique of proof called "case analysis," in which we wish to prove an expression E. We take some other expression F and its negation, `NOT` F, and prove that both F and `NOT` F imply E. Since F must be either true or false, we can conclude E. The formal basis for case analysis is the following tautology.

12.26. *Case analysis*: $\big((p \to q)$ `AND` $(\bar{p} \to q)\big) \equiv q$.

That is, the two cases occur when p is true and when p is false. If q is implied in both cases, then q must be true. We leave it as an exercise to show that 12.26 follows from 12.25 and other laws we have proved.

12.27. $p\bar{p} \equiv 0$.

A proposition and its negation cannot both be true simultaneously. This law is vital when we make a "proof by contradiction." We discuss this technique of proof shortly, in law 12.29, and also in Section 12.11, when we cover resolution proofs.

Proving the Contrapositive

Sometimes we want to prove an implication, like $p \to q$, but we find it easier to prove $\bar{q} \to \bar{p}$, which is an equivalent expression called the *contrapositive* of $p \to q$. This principle is formalized in the following law.

12.28. *The contrapositive law*: $(p \to q) \equiv (\bar{q} \to \bar{p})$.

✦ **Example 12.23.** Let us consider a simple example of a proof that shows how the contrapositive law may be used. This example also shows the limitations of propositional logic in proofs. Logic takes us part of the way, allowing us to reason about statements without reference to what the statements mean. However, to get a complete proof, we normally have to make some argument that refers to the meaning of our terms. For this example, we need to know what concepts about integers, like "prime," "odd," and "greater than" mean.

We shall consider three propositions about a positive integer x:

a	"$x > 2$"
b	"x is a prime"
c	"x is odd"

The theorem we want to prove is $ab \to c$, that is,

STATEMENT "If x is greater than 2 and a prime, then x is odd."

We begin by applying some of the laws we have studied to turn the expression $ab \rightarrow c$ into an equivalent expression that is more amenable to proof. First, we use law 12.28 to turn it into its contrapositive, $\bar{c} \rightarrow \texttt{NOT}\ (ab)$. Then we use DeMorgan's law 12.20(a) to turn $\texttt{NOT}\ (ab)$ into $\bar{a} + \bar{b}$. That is, we have transformed the theorem to be proved into $\bar{c} \rightarrow (\bar{a} + \bar{b})$. Put another way, we need to prove that

STATEMENT "If x is not odd, then x is not greater than 2 or x is not prime."

We can replace "not odd" by "even," "not greater than 2" by "equal to or less than 2," and "not prime" by "composite." Thus we want to prove

STATEMENT "If x is even, then $x \leq 2$ or x is composite."

Now we have gone as far as we can go with propositional logic, and we must start talking about the meaning of our terms. If x is even, then $x = 2y$ for some integer y; that is what it means for x to be even. Since x is assumed through this proof to be a positive integer, y must be 1 or greater.

Now we use case analysis, considering the cases where y is 1, and y is greater than 1, which are the only possibilities, since we just argued that $y \geq 1$. If $y = 1$, then $x = 2$, and so we have proved $x \leq 2$. If $y > 1$, then x is the product of two integers, 2 and y, both greater than 1, which means that x is composite. Thus we have shown that if x is even, then either $x \leq 2$ (in the case $y = 1$) or x is composite (in the case $y > 1$). ✦

Proof by Contradiction

Frequently, rather than make a "direct" proof of an expression E, we find it easier to start by assuming $\texttt{NOT}\ E$ and proving from that a *contradiction,* that is, the expression `FALSE`. The basis for such proofs is the following tautology.

12.29. *Proof by contradiction*: $(\bar{p} \rightarrow 0) \equiv p$.

Informally, if starting with $\bar{p}$ we can conclude 0, that is, conclude `FALSE` or derive a contradiction, then that is the same as proving p. This law actually follows from others we have stated. Start with law 12.24 with $\bar{p}$ in place of p, and 0 in place of q, to get the equivalence

$$(\bar{p} \rightarrow 0) \equiv \big(\texttt{NOT}\ (\bar{p}) + 0\big)$$

Law 12.13, the elimination of double negatives, lets us replace $\texttt{NOT}\ (\bar{p})$ by p, and so

$$(\bar{p} \rightarrow 0) \equiv (p + 0)$$

Now law 12.11 tells us that $(p + 0) \equiv p$, and so a further substitution gives us

$$(\bar{p} \rightarrow 0) \equiv p$$

✦ **Example 12.24.** Let us reconsider the propositions a, b, and c from Example 12.23, which talk about a positive integer x and assert, respectively, that $x > 2$, x is a prime, and x is odd. We want to prove the theorem $ab \rightarrow c$, and so we substitute this expression for p in 12.29. Then $\bar{p} \rightarrow 0$ becomes $(\texttt{NOT}\ (ab \rightarrow c)) \rightarrow 0$.

If we use 12.24 on the first of these implications, we get

$$\Big(\texttt{NOT}\ \big(\texttt{NOT}\ (ab) + c\big)\Big) \rightarrow 0$$

DeMorgan's law applied to the inner NOT gives $\big(\texttt{NOT}\ (\bar{a} + \bar{b} + c)\big) \rightarrow 0$. Another use of DeMorgan's law followed by 12.13 twice to eliminate the double negatives turns this expression into $(ab\bar{c}) \rightarrow 0$.

That is as far as propositional logic takes us; now we must reason about integers. We must start with a, b, and $\bar{c}$ and derive a contradiction. In words, we start by assuming that $x > 2$, x is a prime, and x is even, and from these we must derive a contradiction.

Since x is even, we can say $x = 2y$ for some integer y. Since $x > 2$, it must be that $y \geq 2$. But then x, which equals $2y$, is the product of two integers, each greater than 1, and therefore x is composite. Thus we have proven that x is not a prime, that is, the statement $\bar{b}$. Since we were given b, that x is a prime, and we now have $\bar{b}$ as well, we have $b\bar{b}$, which by 12.27 is equivalent to 0, or FALSE.

We have thus proved $\big(\texttt{NOT}\ (ab \rightarrow c)\big) \rightarrow 0$, which is equivalent to $ab \rightarrow c$ by 12.29. That completes our proof by contradiction. ✦

Equivalence to Truth

Our next proof method allows us to prove an expression to be a tautology by transforming it by substitution of equals for equals until the expression is reduced to 1 (TRUE).

12.30. *Proof by equivalence to truth*: $(p \equiv 1) \equiv p$.

✦ **Example 12.25.** The expression $rs \rightarrow r$ says the AND of two expressions implies the first of them (and by commutativity of AND, also implies the second). We can show that $rs \rightarrow r$ is a tautology by the following sequence of equivalences.

$$\begin{array}{rcl} & & rs \rightarrow r \\ 1) & \equiv & \texttt{NOT}\ (rs) + r \\ 2) & \equiv & (\bar{r} + \bar{s}) + r \\ 3) & \equiv & 1 + \bar{s} \\ 4) & \equiv & 1 \end{array}$$

(1) follows by applying law 12.24, the definition of $\rightarrow$ in terms of AND and OR. (2) is an application of DeMorgan's law. (3) follows when we use 12.7 and 12.8 to reorder terms and then replace $r + \bar{r}$ by 1 according to law 12.25. Finally, (4) is an application of law 12.12, the fact that 1 is an annihilator for OR. ✦

EXERCISES

12.9.1: Show that laws 12.25 and 12.27 are duals of each other.

12.9.2*: We would like to prove the theorem "If x is a perfect square and x is even, then x is divisible by 4."

a) Designate propositional variables to stand for the three conditions about x mentioned in the theorem.
b) Write the theorem formally in terms of these propositions.
c) State the contrapositive of your answer to (b), both in terms of your propositional variables and in colloquial terms.
d) Prove the statement from (c). *Hint*: It helps to notice that if x is not divisible by 4, then either x is odd, or $x = 2y$ and y is odd.

12.9.3*: Give a proof by contradiction of the theorem from Exercise 12.9.2.

12.9.4*: Repeat Exercises 12.9.2 and 12.9.3 for the statement "If x^3 is odd, then x is odd" about integers x. (But in 12.9.2(a) there are only two conditions discussed.)

12.9.5*: Prove the following are tautologies by showing they are equivalent to 1 (`TRUE`).

a) $pq + r + \bar{q}\bar{r} + \bar{p}\bar{r}$
b) $p + \bar{q}\bar{r} + \bar{p}r + q\bar{r}$

12.9.6*: Prove law 12.26, case analysis, by substitution of equals for equals in (instances of) previously proved laws.

12.9.7*: Generalize the case analysis law to the situation where the cases are defined by k propositional variables, which may be true or false in all 2^k combinations. What is the justifying tautology for the case $k = 2$? For general k? Show why this tautology must be true.

12.10 Deduction

We have seen logic as a design theory in Sections 12.6 to 12.8, and as a formalization of proof techniques in Section 12.9. Now, let us see a third side of the picture: the use of logic in *deduction,* that is, in sequences of statements that constitute a complete proof. Deduction should be familiar to the reader from the study of plane geometry in high school, where we learn to start with certain *hypotheses* (the "givens"), and to prove a *conclusion* by a sequence of steps, each of which follows from previous steps by one of a limited number of reasons, called *inference rules.* In this section, we explain what constitutes a deductive proof and give a number of examples.

Hypothesis and conclusion

Inference rule

Unfortunately, discovering a deductive proof for a tautology is difficult. As we mentioned in Section 12.7, it is an example of an "inherently intractable" problem, in the NP-hard class. Thus we cannot expect to find deductive proofs except by luck or by exhaustive search. In Section 12.11, we shall discuss resolution proofs, which appear to be a good heuristic for finding proofs, although in the worst case this technique, like all others, must take exponential time.

What Constitutes a Deductive Proof?

Suppose we are given certain logical expressions $E_1, E_2, \ldots, E_k$ as hypotheses, and we wish to draw a conclusion in the form of another logical expression E. In general, neither the conclusion nor any of the hypotheses will be tautologies, but what we want to show is that

$$(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow E \tag{12.11}$$

Applications of Deduction

Automated theorem proving

In addition to being the stuff of which all proofs in mathematics are ultimately made, deduction or formal proof has many uses in computer science. One application is *automated theorem proving.* There are systems that find proofs of theorems by searching for sequences of steps that proceed from hypotheses to conclusion. Some systems search for proofs on their own, and others work interactively with the user, taking hints and filling in small gaps in the sequence of steps that form a proof. Some believe that these systems will eventually be useful for proving the correctness of programs, although much progress must be made before such facilities are practical.

A second use of deductive proofs is in programming languages that relate deduction to computation. As a very simple example, a robot finding its way through a maze might represent its possible states by a finite set of positions in the centers of hallways. We could draw a graph in which the nodes represent the positions, and an arc $u \rightarrow v$ means that it is possible for the robot to move from position u to position v by a simple move because u and v represent adjacent hallways.

We could also think of the positions as propositions, where u stands for "The robot can reach position u." Then $u \rightarrow v$ can be interpreted not only as an arc, but as a logical implication, that is, "If the robot can reach u, then it can reach v." (Note the "pun"; the arrow can represent an arc or an implication.) A natural question to ask is: "What positions can the robot reach from position a."

We can phrase this question as a deduction if we take the expression a, and all expressions $u \rightarrow v$ for adjacent positions u and v, as hypotheses, and see which propositional variables x we can prove from these hypotheses. In this case, we don't really need a tool as powerful as deduction, because depth-first search works, as discussed in Section 9.7. However, there are many related situations where graph-theoretic methods are not effective, yet the problem can be couched as a deduction and a reasonable solution obtained.

is a tautology. That is, we want to show that if we assume $E_1, E_2, \ldots, E_k$ are true, then it follows that E is true.

One way to show (12.11) is to construct its truth table and test whether it has value 1 in each row — the routine test for a tautology. However, that may not be sufficient for two reasons.

1. As we mentioned, tautology testing becomes infeasible if there are too many variables in the expression.

2. More importantly, while tautology testing works for propositional logic, it cannot be used to test tautologies in more complex logical systems, such as predicate logic, discussed in Chapter 14.

We can often show that (12.11) is a tautology by presenting a *deductive proof.* A deductive proof is a sequence of lines, each of which is either a given hypothesis or is constructed from one or more previous lines by a rule of inference. If the last line is E, then we say that we have a proof of E from $E_1, E_2, \ldots, E_k$.

There are many rules of inference we might use. The only requirement is that if an inference rule allows us to write expression F as a line whenever expressions $F_1, F_2, \ldots, F_n$ are lines, then

The Sound of No Hands Clapping

Frequently, we need to understand the limiting case of an operator applied to no operands, as we did in inference rule (a). We asserted that it makes sense to regard the AND of zero expressions (or lines of a proof) as having truth value 1. The motivation is that F_1 AND F_2 AND $\cdots$ AND F_n is true unless there is at least one of the F's that is false. But if $n = 0$, that is, there are no F's, then there is no way the expression could be false, and thus it is natural to take the AND of zero expressions to be 1.

We adopt the convention that whenever we apply an operator to zero operands, the result is the identity for that operator. Thus we expect the OR of zero expressions to be 0, since an OR of expressions is true only if one of the expressions is true; if there are no expressions, then there is no way to make the OR true. Likewise, the sum of zero numbers is taken to be 0, and the product of zero numbers is taken to be 1.

$$(F_1 \texttt{ AND } F_2 \texttt{ AND } \cdots \texttt{ AND } F_n) \rightarrow F$$

must be a tautology. For example,

a) Any tautology may be used as a line in a proof, regardless of previous lines. The justification for this rule is that if F is a tautology, then the logical AND of zero lines of the proof implies F. Note that the AND of zero expressions is 1, conventionally, and $1 \rightarrow F$ is a tautology when F is a tautology.

Modus ponens

b) The rule of *modus ponens* says that if E and $E \rightarrow F$ are lines of a proof, then F may be added as a line of the proof. Modus ponens follows from the tautology $\big(p \texttt{ AND } (p \rightarrow q)\big) \rightarrow q$; here expression E is substituted for p and F for q. The only subtlety is that we do not need a line with E AND $(E \rightarrow F)$, but rather two separate lines, one with E and one with $E \rightarrow F$.

c) If E and F are two lines of a proof, then we can add the line E AND F. The justification is that $(p \texttt{ AND } q) \rightarrow (p \texttt{ AND } q)$ is a tautology; we may substitute any expression E for p and F for q.

d) If we have lines E and $E \equiv F$, then we may add line F. The justification is similar to modus ponens, since $E \equiv F$ implies $E \rightarrow F$. That is,

$$\big(p \texttt{ AND } (p \equiv q)\big) \rightarrow q$$

is a tautology, and inference rule (d) is a substituted instance of this tautology.

◆ **Example 12.26.** Suppose we have the following propositional variables, with intuitive meanings as suggested.

r	"It is raining."
u	"Joe brings his umbrella."
w	"Joe gets wet."

We are given the following hypotheses.

$r \rightarrow u$	"If it rains, Joe brings his umbrella."
$u \rightarrow \bar{w}$	"If Joe has an umbrella, he doesn't get wet."
$\bar{r} \rightarrow \bar{w}$	"If it doesn't rain, Joe doesn't get wet."

We are asked to prove $\bar{w}$, that is, Joe never gets wet. In a sense, the matter is trivial, since the reader may check that

$$\big((r \rightarrow u)\ \texttt{AND}\ (u \rightarrow \bar{w})\ \texttt{AND}\ (\bar{r} \rightarrow \bar{w})\big) \rightarrow \bar{w}$$

is a tautology. However, it is also possible to prove $\bar{w}$ from the hypotheses, using some of the algebraic laws from Section 12.8 and some of the inference rules just discussed. The approach of finding a proof is the one that we would have to take if we were dealing with a more complex form of logic than propositional calculus or with a logical expression involving many variables. One possible proof, along with the justification for each step, is shown in Fig. 12.24.

The rough idea of the proof is that we use case analysis, considering both the cases where it is raining and it is not raining. By line (5) we have proved that if it is raining, Joe doesn't get wet, and line (6), a given hypothesis, says that if it is not raining Joe doesn't get wet. Lines (7) through (9) combine the two cases to draw the desired conclusion. ❖

1)	$r \rightarrow u$	Hypothesis
2)	$u \rightarrow \bar{w}$	Hypothesis
3)	$(r \rightarrow u)\ \texttt{AND}\ (u \rightarrow \bar{w})$	(c) applied to (1) and (2)
4)	$\big((r \rightarrow u)\ \texttt{AND}\ (u \rightarrow \bar{w})\big) \rightarrow (r \rightarrow \bar{w})$	Substitution into law (12.23)
5)	$r \rightarrow \bar{w}$	Modus ponens, with (3) and (4)
6)	$\bar{r} \rightarrow \bar{w}$	Hypothesis
7)	$(r \rightarrow \bar{w})\ \texttt{AND}\ (\bar{r} \rightarrow \bar{w})$	(c) applied to (5) and (6)
8)	$\big((r \rightarrow \bar{w})\ \texttt{AND}\ (\bar{r} \rightarrow \bar{w})\big) \equiv \bar{w}$	Substitution into law (12.26)
9)	$\bar{w}$	(d) with (7) and (8)

Fig. 12.24. Example of a deductive proof.

Why a Deductive Proof "Works"

A deductive proof, recall, starts with hypotheses $E_1, E_2, \ldots, E_k$ and adds additional lines (i.e., expressions), each of which is implied by $E_1\ \texttt{AND}\ E_2\ \texttt{AND} \cdots \texttt{AND}\ E_k$. Each line we add is implied by the AND of zero or more previous lines or is one of the hypotheses. We can show that $E_1\ \texttt{AND}\ E_2\ \texttt{AND} \cdots \texttt{AND}\ E_k$ implies each line of the proof, by induction on the number of lines added so far. To do so, we need two families of tautologies involving implication. The first family is a generalized transitive law for $\rightarrow$. For any n:

$$\Big((p \rightarrow q_1)\ \texttt{AND}\ (p \rightarrow q_2)\ \texttt{AND} \cdots \texttt{AND}\ (p \rightarrow q_n) \\ \texttt{AND}\ \big((q_1 q_2 \cdots q_n) \rightarrow r\big)\Big) \rightarrow (p \rightarrow r) \tag{12.12}$$

That is, if p implies each of the q_i's, and the q_i's together imply r, then p implies r.

We find that (12.12) is a tautology by the following reasoning. The only way that (12.12) could be false is if $p \rightarrow r$ were false and the left-hand side true. But $p \rightarrow r$ can only be false if p is true and r is false, and so in what follows we shall assume p and $\bar{r}$. We must show that the left-hand side of (12.12) is then false.

If the left-hand side of (12.12) is true, then each of its subexpressions connected by AND is true. For example, $p \rightarrow q_1$ is true. Since we assume p is true, the only way $p \rightarrow q_1$ can be true is if q_1 is true. Similarly, we can conclude that $q_2, \ldots, q_n$ are all true. Thus $q_1 q_2 \cdots q_n \rightarrow r$, must be false, since we assume r is false and we have just discovered that all the q_i's are true.

We started by assuming that (12.12) was false and observed that the right-hand side must therefore be true, and thus p and $\bar{r}$ must be true. We then concluded that the left-hand side of (12.12) is false when p is true and r is false. But if the left-hand side of (12.12) is false, then (12.12) itself is true, and we have a contradiction. Thus (12.12) can never be false and is therefore a tautology.

Note that if $n = 1$ in (12.12), then we have the usual transitive law of $\rightarrow$, which is law 12.23. Also, if $n = 0$, then (12.12) becomes $(1 \rightarrow r) \rightarrow r$, which is a tautology. Recall that when $n = 0$, $q_1 q_2 \cdots q_n$ is conventionally taken to be the identity for AND, which is 1.

We also need a family of tautologies to justify the fact that we can add the hypotheses to the proof. It is a generalization of a tautology discussed in Example 12.25. We claim that for any m and i such that $1 \leq i \leq m$,

$$(p_1 p_2 \cdots p_m) \rightarrow p_i \tag{12.13}$$

is a tautology. That is, the AND of one or more propositions implies any one of them.

The expression (12.13) is a tautology because the only way it could be false is if the left-hand side is true and the right-hand side, p_i, is false. But if p_i is false, then the AND of p_i and other p's is surely false, so the left-hand side of (12.13) is false. But (12.13) is true whenever its left-hand side is false.

Now we can prove that, given

1. Hypotheses $E_1, E_2, \ldots, E_k$, and
2. A set of inference rules such that, whenever they allow us to write a line F, this line is either one of the E_i's, or there is a tautology

   $$(F_1 \texttt{ AND } F_2 \texttt{ AND } \cdots \texttt{ AND } F_n) \rightarrow F$$

   for some set of previous lines $F_1, F_2, \ldots, F_n$,

it must be that $(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow F$ is a tautology for each line F. The induction is on the number of lines added to the proof.

BASIS. For a basis, we take zero lines. The statement holds, since it says something about every line F of a proof, and there are no such lines to discuss. That is, our inductive statement is really of the form "if F is a line then $\cdots$," and we know such an if-then statement is true if the condition is false.

INDUCTION. For the induction, suppose that for each previous line G,

$$(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow G$$

Deductive Proofs Versus Equational Proofs

The kinds of proofs we saw in Sections 12.8 and 12.9 differ in flavor from the deductive proofs studied in Section 12.10. However, proofs of both kinds involve the creation of a sequence of tautologies, leading to the desired tautology.

In Sections 12.8 and 12.9 we saw equational proofs, where starting with one tautology we made substitutions to derive other tautologies. All the tautologies derived have the form $E \equiv F$ for some expressions E and F. This style of proof is used in high-school trigonometry, for example, where we learn to prove "trigonometric identities."

Deductive proofs also involve discovery of tautologies. The only difference is that each is of the form $E \rightarrow F$, where E is the AND of the hypotheses, and F is the line of the proof that we actually right down. The fact that we do not write the full tautology is a notational convenience, not a fundamental distinction. This style of proof should also be familiar from high-school; it is the style of proofs in plane geometry, for example.

is a tautology. Let F be the next line added. There are two cases.

Case 1: F is one of the hypotheses. Then $(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow F$ is a tautology because it comes from (12.13) with $m = k$ when we substitute E_j for each p_j, for $j = 1, 2, \ldots, k$.

Case 2: F is added because there is a rule of inference

$$(F_1 \texttt{ AND } F_2 \texttt{ AND } \cdots \texttt{ AND } F_n) \rightarrow F$$

where each of the F_j's is one of the previous lines. By the inductive hypothesis,

$$(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow F_j$$

is a tautology for each j. Thus, if we substitute F_j for q_j in (12.12), substitute

$$E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k$$

for p, and substitute F for r, we know that any substitution of truth values for the variables of the E's and F's makes the left-hand side of (12.12) true. Since (12.12) is a tautology, every assignment of truth values must also make the right-hand side true. But the right-hand side is $(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow F$. We conclude that this expression is true for every assignment of truth values; that is, it is a tautology.

We have now concluded the induction, and we have shown that

$$(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow F$$

for every line F of the proof. In particular, if the last line of the proof is our goal E, we know $(E_1 \texttt{ AND } E_2 \texttt{ AND } \cdots \texttt{ AND } E_k) \rightarrow E$.

EXERCISES

12.10.1*: Give proofs of the following conclusions from the following hypotheses. You may use inference rules (a) through (d). For tautologies you may use only the laws stated in Sections 12.8 and 12.9 and tautologies that follow by using instances of these laws to "substitute equals for equals."

a) Hypotheses: $p \to q$, $p \to r$; conclusion: $p \to qr$.
b) Hypotheses: $p \to (q + r)$, $p \to (q + \bar{r})$; conclusion: $p \to q$.
c) Hypotheses: $p \to q$, $qr \to s$; conclusion: $pr \to s$.

12.10.2: Justify why the following is a rule of inference. If $E \to F$ is a line, and G is any expression whatsoever, then we can add $E \to (F \texttt{ OR } G)$ as a line.

12.11 Proofs by Resolution

As we mentioned earlier in this chapter, finding proofs is a hard problem, and since the tautology problem is very likely to be inherently exponential, there is no general way to make finding proofs easy. However, there are many techniques known that for "typical" tautologies appear to help with the exploration needed in the search for a proof. In this section we shall study a useful inference rule, called *resolution,* that is perhaps the most basic of these techniques. Resolution is based on the following tautology.

$$\big((p + q)(\bar{p} + r)\big) \to (q + r) \tag{12.14}$$

The validity of this rule of inference is easy to check. The only way it could be false is if $q + r$ were false, and the left-hand side were true. If $q + r$ is false, then both q and r are false. Suppose p is true, so that $\bar{p}$ is false. Then $\bar{p} + r$ is false, and the left-hand side of (12.14) must be false. Similarly, if p is false, then $p + q$ is false, which again tells us that the left-hand side is false. Thus it is impossible for the right-hand side to be false while the left-hand side is true, and we conclude that (12.14) is a tautology.

Clause

The usual way resolution is applied is to convert our hypotheses into *clauses,* which are sums (logical OR's) of literals. We convert each of our hypotheses into a product of clauses. Our proof begins with each of these clauses as a line of the proof, and the justification is that each is "given." We then apply the resolution rule to construct additional lines, which will always turn out to be clauses. That is, if q and r in (12.14) are each replaced by any sum of literals, then $q + r$ will also be a sum of literals.

In practice, we shall simplify clauses by removing duplicates. That is, both q and r could include a literal X, in which case we shall remove one copy of X from $q + r$. The justification is found in laws 12.17, 12.7, and 12.8, the idempotence, commutativity, and associativity of OR. In general, a useful point of view is that a clause is a set, rather than a list, of literals. The associative and commutative laws allow us to order the literals any way we please, and the idempotent law allows us to eliminate duplicates.

We also eliminate clauses that contain contradictory literals. That is, if both X and $\bar{X}$ are found in one clause, then by laws 12.25, 12.7, 12.8, and 12.15, the clause

is equivalent to 1, and there is no need to include it in a proof. That is, by law 12.25, $(X + \bar{X}) \equiv 1$, and by the annihilator law 12.15, 1 `OR` anything is equivalent to 1.

✥ **Example 12.27.** Consider the clauses $(a + \bar{b} + c)$ and $(\bar{d} + a + b + e)$. We may let b play the role of p in (12.14). Then q is $\bar{d} + a + e$, and r is $a + c$. Notice that we have done some rearrangement to match our clauses with (12.14). First, our second clause has been matched with the first, $p + q$ in (12.14), and our first clause is matched with the second of (12.14). Moreover, the variable that plays the role of p does not appear first in our two clauses, but that is no matter, because the commutative and associative laws for `OR` justify our rearranging the clauses in any order we choose.

The new clause $q + r$, which may appear as a line in a proof if our two clauses are already in that proof, is $(\bar{d} + a + e + a + c)$. We may simplify this clause by eliminating the duplicate a, leaving $(\bar{d} + a + e + c)$.

As another example, consider the clauses $(a + b)$ and $(\bar{a} + \bar{b})$. We may let a play the roll of p in (12.14); q is b, and r is $\bar{b}$, giving us the new clause $(b + \bar{b})$. That clause is equivalent to 1, and therefore need not be generated. ✥

Putting Logical Expressions in Conjunctive Normal Form

In order to make resolution work, we need to put all hypotheses, and the conclusion, into product-of-sums form, or "conjunctive normal form." There are several approaches that may be taken. Perhaps the simplest is the following.

1. First, we get rid of any operators except `AND`, `OR`, and `NOT`. We replace $E \equiv F$ by $(E \rightarrow F)(F \rightarrow E)$, by law 12.21. Then, we replace $G \rightarrow H$ by

   $$\texttt{NOT}\ (G) + (H)$$

   according to law 12.24. `NAND` and `NOR` are easily replaced by `AND` or `OR`, respectively, followed by `NOT`. In fact, since `AND`, `OR`, and `NOT` are a complete set of operators, we know that any logical operator whatsoever, including those not introduced in this book, can be replaced by expressions involving only `AND`, `OR`, and `NOT`.

2. Next, apply DeMorgan's laws to push all negations down until they either cancel with other negations by law 12.13 in Section 12.8, or they apply only to propositional variables.

3. Now we apply the distributive law for `OR` over `AND` to push all `OR`'s below all `AND`'s. The result is an expression in which there are literals, combined by `OR`'s, which are then combined by `AND`'s; this is a conjunctive normal form expression.

✥ **Example 12.28.** Let us consider the expression

$$p + \Big(q\ \texttt{AND}\ \texttt{NOT}\ \big(r\ \texttt{AND}\ (s \rightarrow t)\big)\Big)$$

Note that to balance conciseness and clarity, we are using overbar, +, and juxtaposition mixed with their equivalents — NOT, OR, and AND — in this and subsequent expressions.

Step (1) requires us to replace $s \rightarrow t$ by $\bar{s} + t$, giving the AND-OR-NOT expression

$$p + \Big(q \texttt{ AND NOT } \big(r(\bar{s} + t)\big)\Big)$$

In step (2), we must push the first NOT down by DeMorgan's laws. The sequence of steps, in which the NOT reaches the propositional variables is

$$p + \Big(q\big(\bar{r} + \texttt{ NOT } (\bar{s} + t)\big)\Big)$$
$$p + \Big(q\big(\bar{r} + (\texttt{NOT } \bar{s})(\bar{t}\ )\big)\Big)$$
$$p + \Big(q\big(\bar{r} + (s\bar{t}\ )\big)\Big)$$

Now we apply law 12.14 to push the first OR below the first AND.

$$(p + q)\Big(p + \big(\bar{r} + (s\bar{t})\big)\Big)$$

Next, we regroup, using law 12.8 of Section 12.8, so that we can push the second and third OR's below the second AND.

$$(p + q)\big((p + \bar{r}) + (s\bar{t})\big)$$

Finally, we use law 12.14 again, and all OR's are below all AND's. The resulting expression,

$$(p + q)(p + \bar{r} + s)(p + \bar{r} + \bar{t})$$

is in conjunctive normal form. ✦

Inference Using Resolution

We now see the outline of a way to find a proof of E from hypotheses $E_1, E_2, \ldots, E_k$. Convert E and each of $E_1, \ldots, E_k$ into conjunctive normal form expressions F and $F_1, F_2, \ldots, F_k$, respectively. Our proof is a list of clauses, and we start by writing down all the clauses of the hypotheses $F_1, F_2, \ldots, F_k$. We apply the resolution rule to pairs of clauses, and thus we add new clauses as lines of our proof. Then, if we add all the clauses of F to our proof, we have proved F, and we therefore have also proved E.

✦ **Example 12.29.** Suppose we take as our hypothesis the expression

$$(r \rightarrow u)(u \rightarrow \bar{w})(\bar{r} \rightarrow \bar{w})$$

Note that this expression is the AND of the hypotheses used in Example 12.26.[5] Let the desired conclusion be $\bar{w}$, as in Example 12.26. We convert the hypothesis to conjunctive normal form by replacing the →'s according to law 12.24. At this point, the result is already in conjunctive normal form and needs no further manipulation. The desired conclusion, $\bar{w}$, is already in conjunctive normal form, since any single

[5] You should have observed by now that it doesn't matter whether we write many hypotheses or connect them all with AND's and write one hypothesis.

Why Resolution is Effective

In general, the discovery of a proof requires luck or skill to put together the sequence of lines that lead from hypotheses to conclusion. You will by now have noted, while it is easy to check that the proofs given in Sections 12.10 and 12.11 are indeed valid proofs, solving exercises that require the discovery of a proof is much harder. Guessing the sequence of resolutions to perform in order to produce some clause or clauses, as in Example 12.29, is not significantly easier than discovering a proof in general.

However, when we combine resolution with proof by contradiction, as in Example 12.30, we see the magic of resolution. Since our goal clause is 0, the "smallest" clause, we suddenly have a notion of a "direction" in which to search. That is, we try to prove progressively smaller clauses, hoping thereby to prove 0 eventually. Of course, this heuristic does not guarantee success. Sometimes, we must prove some very large clauses before we can start shrinking clauses and eventually prove 0.

Complete proof procedure

In fact, resolution is a *complete* proof procedure for propositional calculus. Whenever $E_1E_2\cdots E_k \rightarrow E$ is a tautology, we can derive 0 from $E_1, E_2, \ldots, E_k$ and `NOT` E, expressed in clause form. (Yes, this is a third meaning that logicians give to the word "complete." Recall the others are "a set of operators capable of expressing any logical function," and "a hardest problem within a class of problems," as in "NP-complete".) Again, just because the proof exists doesn't mean it is easy to find the proof.

literal is a clause, and a single clause is a product of clauses. Thus we begin with clauses

$$(\bar{r} + u)(\bar{u} + \bar{w})(r + \bar{w})$$

Now, suppose we resolve the first and third clauses, using r in the role of p. The resulting clause is $(u + \bar{w})$. This clause may be resolved with the second clause in the hypothesis, with u in the role of p, to get clause $(\bar{w})$. Since this clause is the desired conclusion, we are done. Figure 12.25 shows the proof as a series of lines, each of which is a clause. ✦

1)	$(\bar{r} + u)$	Hypothesis
2)	$(\bar{u} + \bar{w})$	Hypothesis
3)	$(r + \bar{w})$	Hypothesis
4)	$(u + \bar{w})$	Resolution of (1) and (3)
5)	$(\bar{w})$	Resolution of (2) and (4)

Fig. 12.25. Resolution proof of $\bar{w}$.

Resolution Proofs by Contradiction

The usual way resolution is used as a proof mechanism is somewhat different from that in Example 12.29. Instead of starting with the hypothesis and trying to prove the conclusion, we start with both the hypotheses and the negation of the conclusion and try to derive a clause with no literals. This clause has the value 0, or `FALSE`.

For example, if we have clauses (p) and $(\bar{p})$, we may apply (12.14) with $q = r = 0$, to get the clause 0.

The reason this approach is valid stems from the contradiction law 12.29 of Section 12.9, or $(\bar{p} \rightarrow 0) \equiv p$. Here, let p be the statement we want to prove: $(E_1 E_2 \cdots E_k) \rightarrow E$, for some hypotheses $E_1, E_2, \ldots, E_k$ and conclusion E. Then $\bar{p}$ is NOT $(E_1 E_2 \cdots E_k \rightarrow E)$, or NOT $\big($NOT $(E_1 E_2 \cdots E_k) + E\big)$, using law 12.24. Several applications of DeMorgan's laws tell us that p is equivalent to $E_1 E_2 \cdots E_k \bar{E}$. Thus, to prove p we can instead prove $\bar{p} \rightarrow 0$, or $(E_1 E_2 \cdots E_k \bar{E}) \rightarrow 0$. That is, we prove that the hypotheses and the negation of the conclusion together imply a contradiction.

✦ **Example 12.30.** Let us reconsider Example 12.29, but start with both the three hypothesis clauses and the negation of the desired conclusion, that is, with clause (w) as well. The resolution proof of 0 is shown in Fig. 12.26. Using the law of contradiction, we can conclude that the hypotheses imply $\bar{w}$, the conclusion. ✦

1)	$(\bar{r} + u)$	Hypothesis
2)	$(\bar{u} + \bar{w})$	Hypothesis
3)	$(r + \bar{w})$	Hypothesis
4)	(w)	Negation of conclusion
5)	$(u + \bar{w})$	Resolution of (1) and (3)
6)	$(\bar{w})$	Resolution of (2) and (5)
7)	0	Resolution of (4) and (6)

Fig. 12.26. Resolution proof by contradiction.

EXERCISES

12.11.1: Use the truth table method to check that expression (12.14) is a tautology.

a	A person has blood type A.
b	A person has blood type B.
c	A person has blood type AB.
o	A person has blood type O.
t	Test T is positive on a person's blood sample.
s	Test S is positive on a person's blood sample.

Fig. 12.27. Propositions for Exercise 12.11.2.

12.11.2: Let the propositions have the intuitive meanings given in Fig. 12.27. Write a clause or product of clauses that express the following ideas.

a) If test T is positive, then that person has blood type A or AB.
b) If test S is positive, then that person has blood type B or AB.
c) If a person has type A, then test T will be positive.
d) If a person has type B, then test S will be positive.
e) If a person has type AB, then both tests T and S will be positive. *Hint*: Note that $(\bar{c} + st)$ is not a clause.
f) A person has type A, B, AB, or O blood.

12.11.3: Use resolution to discover all nontrivial clauses that follow from your clauses in Exercise 12.11.2. You should omit trivial clauses that simplify to 1 (`TRUE`), and also omit a clause C if its literals are a proper superset of the literals of some other clause D.

12.11.4: Give proofs using resolution and proof by contradiction for the implications in Exercise 12.10.1.

12.12 Summary of Chapter 12

In this chapter, we have seen the elements of propositional logic, including:

- The principal operators, `AND`, `OR`, `NOT`, $\rightarrow$, $\equiv$, `NAND`, and `NOR`.
- The use of truth tables to represent the meaning of a logical expression, including algorithms to construct a truth table from an expression and vice versa.
- Some of the many algebraic laws that apply to the logical operators.

We also discussed logic as a design theory, seeing:

- How Karnaugh maps help us design simple expressions for logical functions that have up to four variables.
- How algebraic laws can be used sometimes to simplify expressions of logic.

Then, we saw that logic helps us express and understand the common proof techniques such as:

- Proof by case analysis,
- Proof of the contrapositive,
- Proof by contradiction, and
- Proof by reduction to truth.

Finally, we studied deduction, that is, the construction of line-by-line proofs, seeing:

- There are a number of inference rules, such as "modus ponens," that allow us to construct one line of a proof from previous lines.
- The resolution technique often helps us find proofs quickly by representing lines of a proof as sums of literals and combining sums in useful ways.
- However, there is no known algorithm that is guaranteed to find a proof of an expression in time less than an exponential in the size of the expression.

- Moreover, since the tautology problem is "NP-hard," it is strongly believed that no less-than-exponential algorithm for this problem exists.

12.13 Bibliographic Notes for Chapter 12

The study of deduction in logic dates back to Aristotle. Boole [1854] developed the algebra of propositions, and it is from this work that Boolean algebra comes.

Lewis and Papadimitriou [1979] is a somewhat more advanced treatment of logic. Enderton [1972] and Mendelson [1987] are popular treatments of mathematical logic. Manna and Waldinger [1990] present the subject from the point of view of proving correctness of programs.

Genesereth and Nilsson [1987] treat logic from the point of view of applications to artificial intelligence. There, you can find more on the matter of heuristics for discovering proofs, including resolution-like techniques. The original paper on resolution as a method of proof is Robinson [1965].

For more on the theory of intractable problems, read Garey and Johnson [1979]. The concept of NP-completeness is by Cook [1971], and the paper by Karp [1972] made clear the importance of the concept for commonly encountered problems.

Boole, G. [1854]. *An Investigation of the Laws of Thought*, McMillan; reprinted by Dover Press, New York, in 1958.

Cook, S. A. [1971]. "The complexity of theorem proving procedures," *Proc. Third Annual ACM Symposium on the Theory of Computing,* pp. 151–158.

Enderton, H. B. [1972]. *A Mathematical Introduction to Logic*, Academic Press, New York.

Garey, M. R. and D. S Johnson [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York.

Genesereth, M. R. and N. J. Nilsson [1987]. *Logical Foundations for Artificial Intelligence*, Morgan-Kaufmann, San Mateo, Calif.

Karp, R. M. [1972]. "Reducibility among combinatorial problems," in *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), Plenum, New York, pp. 85–103.

Lewis, H. R. and C. H. Papadimitriou [1981]. *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, New Jersey.

Manna, Z. and R. Waldinger [1990]. *The Logical Basis for Computer Programming* (two volumes), Addison-Wesley, Reading, Mass.

Mendelson, E. [1987]. *Introduction to Mathematical Logic*, Wadsworth and Brooks, Monterey, Calif.

Robinson, J. A. [1965]. "A machine-oriented logic based on the resolution principle," *J. ACM* **12**:1, pp. 23–41.