12 Boolean Algebra

Introduction

In this chapter we will use Maple to model Boolean algebra. In the first section, we demonstrate the basic commands of the **Logic** package, which will be used extensively in this chapter. In the second section, we will focus on the disjunctive normal form of a logical expression. We will see how to use Maple's command for finding a disjunctive normal form expression for a Boolean function, and we will develop a procedure for finding such a representation for a function defined by a table of values. In Section 3, we will see how Maple can be used to model logical circuits, including how to go about transforming a circuit diagram into a Maple expression. We also provide a procedure that will transform a logical expression into a model of a circuit. In the final section of the chapter, we consider simplification of logical expressions, and we develop an implementation of the Quine-McCluskey method.

In this chapter we will be using the Maple package **Logic**. This package includes several commands related to Boolean algebra that will be useful. We load this package now.

```
> with(Logic):
```

12.1 Boolean Functions

In this section we will introduce Maple's **Logic** package, which can be used to explore Boolean algebra. In particular, we will see how Maple represents Boolean operators, how to work with Boolean expressions, and how to create Boolean functions. We will also use Maple to verify identities in Boolean algebra and to compute the dual of an expression.

Preliminaries

In Chapter 1 of this manual, we discussed Maple's logical expressions. The boolean values *true* and *false* are represented by the literal constants **true** and **false**. To Maple, these are constant values, like numbers 2 or **Pi**.

We also saw in Chapter 1 the logical operators, **and**, **or**, and **not**. These are similar to arithmetic operators like + and *. Combining boolean values with the logical operators causes Maple to evaluate the resulting expression.

```
> true or (not(false) and false);
```

$$\mathit{true} \tag{12.1}$$

These "ordinary" operators are a vital part of any programming language, as they are needed for controlling execution of procedures. Moreover, Maple recognizes a third value, **FAIL**, which is useful from the perspective of programming procedures. (The help page for boolean expressions provides tables showing how the operations are defined on **true**, **false**, and **FAIL**.)

The **Logic** package provides a second set of boolean operators. The basic operators are **&and**, **&or**, and **&not**. These are supplemented by **&nand** (not and), **&nor** (not or), **&xor** (exclusive or), **&implies** (implication), and **&iff** (biconditional).

These operators are different from the ordinary operators in three ways. First, they are inert, as opposed to active. This means that when you enter an expression in Maple using the **Logic** package operators, they are not immediately evaluated.

```
> true &or &not(false) &and false;
```

$$(true \ \&or \ \&not(false)) \ \&and \ false \tag{12.2}$$

This makes it easier to explore and analyze Boolean expressions symbolically since Maple won't perform simplification until you explicitly tell it to do so.

The second difference is that the operators in the **Logic** package do not recognize **FAIL** as a logical value.

The third difference is that the the **Logic** operators all have equal precedence, so you should fully parenthesize statements.

More on precedence

Precedence and parentheses require a bit of explanation. Recall that the usual order of operations for logical operators is **not**, then **and**, then **or**, and finally implication. When you enter an expression with the **Logic** operators, this order of precedence is not respected.

For example, if you enter *p* **or** *q* **and** *r*, using the **Logic** operators, Maple will consider the operations from left to right.

```
> p &or q &and r;
```

$$(p \ \&or \ q) \ \&and \ r \tag{12.3}$$

Note that in the output for that expression, Maple has put *p* **or** *q* in parentheses. This is consistent with the fact that the operators have equal precedence, meaning that the **&or** is applied first. That is, (*p* **or** *q*) **and** *r* is the interpretation of what was input.

Using the inert **Logic** operators, you must enforce the order of precedence yourself. To input *p* **or** *q* **and** *r* and have Maple interpret it in the correct order, you must use parentheses.

```
> p &or (q &and r);
```

$$p \ \&or \ q \ \&and \ r \tag{12.4}$$

You may be surprised that Maple removed the parentheses when it echoed the expression.

There is, of course, a reason that Maple added the parentheses in the first example and removed them in the second. When Maple "reads" your input, it sees two neutral operators, **&or** and **&and**. (Recall that a neutral operator is the means by which users are able to create infix operators. We created neutral operators for modular arithmetic in Section 4.1 of this manual.)

To Maple, neutral operators (except for **&***) have equal precedence, and thus it applies them left to right, unless parentheses direct it otherwise. To Maple,

```
> p &or q &and r;
```

$$(p \ \&or \ q) \ \&and \ r \tag{12.5}$$

is the same as

```
> &or(p,q) &and r;
```

$$(p \ \&or \ q) \ \&and \ r \tag{12.6}$$

which is the same as

```
> &and(&or(p,q),r);
```

$$(p \ \&or \ q) \ \&and \ r \tag{12.7}$$

In essence, that is how Maple approaches the statement **p &or q &and r**. When it encounters the **&or** operator, all it knows is that it is a neutral operator applied to **p** and **q**. Then it encounters **&and** which is applied to the result of **&or(p,q)** and to **r**.

But the operators themselves "know" that they have different precedence. Once Maple has processed (parsed) the entire statement, then it can use the precedence to determine whether or not parentheses are needed in displaying the result.

The upshot of all this is that Maple reads what you enter without respecting precedence, but its output does. You must always fully parenthesize your input with the **Logic** package or you may obtain erroneous results.

The 0-1 form of Boolean algebra
We conclude this subsection with a warning.

The textbook uses the objects 0 and 1 with operations $+$, $\cdot$, and $\bar{}$ instead of *true*, *false*, **or**, **and**, and **not** as we have done. Maple does, in fact, have commands available for using the 0-1 form. The **Logic** package contains commands, **Import** and **Export**, that can be used to change between expressions involving the **Logic** inert operators and either the active boolean operators or a $\{0, 1\}$ form, called **MOD2**.

However, the **MOD2** form is in opposition to the conventions used in the textbook. Specifically, Maple has **+** correspond to **&xor**, rather than **&or**.

To avoid the confusion that this difference could create, we will always use the logical form in this manual.

Boolean Expressions and Boolean Functions
We saw above how to create Boolean expressions using the inert **Logic** operators. Now we will look at how to evaluate boolean expressions and how to create boolean functions.

Evaluating Boolean Expressions
Because the operators in **Logic** are inert, you must explicitly tell Maple to evaluate expressions involving them by applying the **BooleanSimplify** command.

Consider Example 1 from the text, which asks that we compute the value of $1 \cdot 0 + \overline{(0 + 1)}$. To perform this computation in Maple, we must first translate it into a logical statement. We do this by changing 1 into *true*, 0 into *false*, the multiplication into **and**, the addition into **or**, and the bar into **not**.

This results in the Boolean expression (*true* **and** *false*) **or not** (*false* **or** *true*). Note that we added parentheses since the **Logic** arguments have equal precedence.

Using Maple's active logical operators, as in Chapter 1, we would just enter the statement to evaluate it.

```
> (true and false) or not(false or true);
```

false **(12.8)**

Using the inert operators, we apply the **BooleanSimplify** command. This command accepts only one argument, the logical expression that it is to simplify. In this example, it will simplify the expression down to a single truth value.

```
> BooleanSimplify((true &and false) &or &not(false &or true));
```

false **(12.9)**

As you might guess from the name, **BooleanSimplify** does more than evaluate an expression

to a truth value. It will also simplify more general expressions involving unassigned names. For example, consider the Boolean expression $(x + y)(x + z)$. In logical terms, this is $(x$ **or** $y)$ **and** $(x$ **or** $z)$.

```
> BooleanSimplify( (x &or y) &and (x &or z) );
```

$$x \;\&\text{or}\; y \;\&\text{and}\; z \qquad \textbf{(12.10)}$$

Note that Maple has simplified this in accordance with the distributive law. Also remember to interpret the output with precedence in mind. Since **and** has precedence over **or**, the output above is equivalent to x **or** $(y$ **and** $z)$.

Representing Boolean functions

Defining a Boolean function is identical to defining any other function in Maple.

Consider, for example, the boolean function shown below (written in the $\{0, 1\}$ notation).

$$F(x, y, z) = xy + yz + zx.$$

This can be modeled in Maple by the procedure **Fp** (p for procedure).

```
> Fp := proc(x,y,z)
    return (x &and y) &or (y &and z) &or (z &and x);
  end proc:
```

It is often more natural, however, to model such functions as functional operators.

```
> F := (x,y,z) -> (x &and y) &or (y &and z) &or (z &and x);
```

$$F := (x, y, z) \rightarrow \mathit{Logic}\text{:-}\&or(\mathit{Logic}\text{:-}\&or(\mathit{Logic}\text{:-}\&and(x, y), \mathit{Logic}\text{:-}\&and(y, z)), \mathit{Logic}\text{:-}\&and(z, x)) \qquad \textbf{(12.11)}$$

The output of the functional operator definition may be somewhat unexpected. The symbols *Logic:-&and* and *Logic:-&or* indicate that the commands **&and** and **&or** are part of the **Logic** package.

The output of the definition of **F** has revealed the way that Maple processed the Boolean expression. Even though we entered the expression using **&and** and **&or** as infix operators, Maple has internally translated our expression into prefix format with the procedure name followed by arguments in parentheses.

You can work with **F** (or **Fp**) in the usual way. The following applies **F** to **p**, **q**, and **r**.

```
> F(p,q,r);
```

$$(p \;\&\text{and}\; q \;\&\text{or}\; q \;\&\text{and}\; r) \;\&\text{or}\; r \;\&\text{and}\; p \qquad \textbf{(12.12)}$$

Observe what happens when **F** is applied to truth values.

```
> F(true,false,true);
```

$$(\mathit{true} \;\&\text{and}\; \mathit{false} \;\&\text{or}\; \mathit{false} \;\&\text{and}\; \mathit{true}) \;\&\text{or}\; \mathit{true} \;\&\text{and}\; \mathit{true} \qquad \textbf{(12.13)}$$

Because the operators **&and** and **&or** are inert, Maple does not automatically simplify to a truth value. We have to explicitly call **BooleanSimplify**.

```
> BooleanSimplify(F(true,false,true));
```

$$\mathit{true} \qquad \textbf{(12.14)}$$

It can be easier to embed the call to **BooleanSimplify** in the function definition itself. We redefine **F** to include **BooleanSimplify**.

```
> F := (x,y,z) -> BooleanSimplify((x &and y) &or (y &and z) &or
  (z &and x));
```

(12.15)

$$F := (x, y, z) \rightarrow Logic{:}\text{-}BooleanSimplify(Logic{:}\text{-}\&or(Logic{:}\text{-}\&or(Logic{:}\text{-}\&and(x, y), Logic{:}\text{-}\&and(y, z)), Logic{:}\text{-}\&and(z, x))) \quad \textbf{(12.15)}$$

Now applying F to p, q, and r produces the same result as before, since the expression cannot be simplified any further.

```
> F(p,q,r);
```

$$(p \ \&and \ q \ \&or \ p \ \&and \ r) \ \&or \ q \ \&and \ r \quad \textbf{(12.16)}$$

But applying F to truth values will return a single truth value.

```
> F(true,false,true);
```

$$true \quad \textbf{(12.17)}$$

You can also mix truth values and symbols. In this case, the inclusion of **BooleanSimplify** in the definition of **F** causes Maple to simplify the expression as much as possible, given the partial information.

```
> F(true,q,r);
```

$$q \ \&or \ r \quad \textbf{(12.18)}$$

The output indicates that if p is known to be true, then (p **and** q) **or** (q **and** r) **or** (r **and** p) is equivalent to q **or** r.

Values of Boolean functions

Examples 4 and 5 of Section 12.1 illustrate how the values of a Boolean function, in the {0, 1} format, can be displayed in a table. In the logical form, this is equivalent to a truth table for the Boolean function. In Chapter 1 of this manual, we created truth tables by looping through all the possible values. The **Logic** package has a command to create truth tables for us.

The **TruthTable** command requires two arguments. The first argument is a Boolean expression. The second is a list of the variable names that appear in the expression.

For example, we will display the table of values for the Boolean function **F** defined above. The first argument to the **TruthTable** command will be the Boolean expression obtained by **F(p,q,r)**. The second argument will be the list **[p,q,r]**.

```
> Ftable := TruthTable(F(p,q,r),[p,q,r]);
```

$$Ftable := table([(false, true, false) = false, (false, true, true) = true, (true, false, false) = false, (false, false, false) = false, (true, true, false) = true, (false, false, true) = false, (true, false, true) = true, (true, true, true) = true]) \quad \textbf{(12.19)}$$

Note that the result of **TruthTable** is a Maple **table** object. We can make it more readable by printing one entry at a time. Recall that **indices** produces a sequence of lists. In order to use the indices with the selection operation, we must apply **op** to remove the list structure.

```
> for i in indices(Ftable) do
    print(i,Ftable[op(i)]);
  end do;
```

$$[false, true, true], true$$
$$[true, true, true], true$$
$$[true, false, false], false$$
$$[true, false, true], true$$
$$[false, false, false], false$$

$$[true, true, false], true$$
$$[false, false, true], false$$
$$[false, true, false], false \qquad \textbf{(12.20)}$$

Operations on Boolean functions

As with functions on real numbers, Boolean functions can be combined using basic operations. The complement of a Boolean function and the Boolean sum and product of functions are defined in the text.

To compute complements, sums, and products of Boolean functions, you must define a new function in terms of the original. For example, consider the function $G(x, y) = x \cdot y$. In logical notation, this is $G(x, y) = x$ **and** y.

```
> G := (x,y) -> BooleanSimplify(x &and y);
```
$$G := (x, y) \rightarrow Logic\text{:-}BooleanSimplify(Logic\text{:-}\&and(x, y)) \qquad \textbf{(12.21)}$$

The complement of `G`, which we'll call `notG`, is created as follows. The arguments of `notG` are the same as `G`. The formula that defines `notG` is `&not G(x,y)`. That is, `&not` is applied to the result of evaluating `G` on the arguments. And once again, `BooleanSimplify` is called.

```
> notG := (x,y) -> BooleanSimplify(&not G(x,y));
```
$$notG := (x, y) \rightarrow Logic\text{:-}BooleanSimplify(Logic\text{:-}\&not(G(x, y))) \qquad \textbf{(12.22)}$$

Observe that if we evaluate `notG` at a pair of variables, the presence of `BooleanSimplify` ensures that De Morgan's law is applied.

```
> notG(x,y);
```
$$\&not(x)\ \&or\ \&not(y) \qquad \textbf{(12.23)}$$

Let us define another function, $H(x, y) = x \cdot \overline{y}$.

```
> H := (x,y) -> BooleanSimplify(x &and (&not y));
```
$$H := (x, y) \rightarrow Logic\text{:-}BooleanSimplify(Logic\text{:-}\&and(x, Logic\text{:-}\&not(y))) \qquad \textbf{(12.24)}$$

To compute the Boolean sum $G + H$, we combine the functions with the `&or` operator. More precisely, we define a functional operator `GpH` with the formula `G(x,y) &or H(x,y)`.

```
> GpH := (x,y) -> BooleanSimplify(G(x,y) &or H(x,y));
```
$$GpH := (x, y) \rightarrow Logic\text{:-}BooleanSimplify(Logic\text{:-}\&or(G(x, y), H(x, y))) \qquad \textbf{(12.25)}$$

Applying this to a pair of variables, we obtain the following formula for $G + H$.

```
> GpH(x,y);
```
$$x \qquad \textbf{(12.26)}$$

This result indicates that $x \cdot y + x \cdot \overline{y} = x$. This can also be verified using the identities in Table 5 of Section 12.1.

Identities of Boolean Algebra

In Chapter 1, we created a procedure, `AreEquivalent`, for checking whether two logical expressions are logically equivalent. The `Logic` package's `Equivalent` command makes checking identities and equivalence of Boolean expressions fairly straightforward. This command can also be used to check equality of Boolean functions.

Identities and equivalence of Boolean expressions

We will use the distributive law $x(y + z) = xy + xz$ as an example. First we must translate the

statement in the {0, 1} form into a statement of logic: x **and** $(y$ **or** $z) \equiv (x$ **and** $y)$ **or** $(x$ **and** $z)$.

Now we will assign the expressions on either side of the equivalence to names. This is not necessary, but it will make later statements easier to read.

```
> distributiveL := x &and (y &or z);
```

$$distributiveL := x \ \&\text{and} \ (y \ \&\text{or} \ z) \qquad \textbf{(12.27)}$$

```
> distributiveR := (x &and y) &or (x &and z);
```

$$distributiveR := x \ \&\text{and} \ y \ \&\text{or} \ x \ \&\text{and} \ z \qquad \textbf{(12.28)}$$

To confirm the equivalence of the two Boolean expressions, we use the **Equivalent** command. This command requires two arguments, the two Boolean expressions that are to be tested for equivalence. The command returns true if the expressions are equivalent and false if not.

```
> Equivalent(distributiveL, distributiveR);
```

$$true \qquad \textbf{(12.29)}$$

This verifies the given distributive law.

The **Equivalent** command also accepts an optional third argument. In the case that the two expressions are not equivalent, if you provide an unevaluated name (a name in single right quotes), then that name will be assigned to a set of assignments of truth values to the variables in the expression that demonstrate that the expressions are not equivalent.

Consider the non-equivalence: $x + xy \neq y$. In logical form, this is x **or** $(x$ **and** $y) \not\equiv y$. Apply the **Equivalent** command with third argument `'P'`.

```
> Equivalent(x &or (x &and y),y,'P');
```

$$false \qquad \textbf{(12.30)}$$

The name `P` now stores a set indicating assignments for x and y.

```
> P;
```

$$\{x = true, y = false\} \qquad \textbf{(12.31)}$$

This output means that setting x equal to true and y equal to false provides a demonstration, by counterexample, that x **or** $(x$ **and** $y) \not\equiv y$. Indeed, substituting $x = true$ and $y = false$ on the left hand side produces $true$ **or** $(true$ **and** $false) \equiv true$ **or** $false \equiv true$. That is not the same as the right hand side, y, which is assigned $false$.

Note that the single right quotes around the name in the third argument of **Equivalent** ensure that, should `P` have already stored a value, that value would be overwritten. If you omit the single quotes and `P` has a value already stored in it, an error will result.

Equality of Boolean functions

Equality of Boolean functions can also be checked with the **Equivalent** command. You do this by applying the command to the two functions evaluated on the same variables.

Consider the following Boolean functions.

$f_1(x, y) = \overline{(xy)}$

$f_2(x, y) = \overline{x} + \overline{y}$.

Define the corresponding functional operators:

```
> f1 := (x,y) -> BooleanSimplify(&not(x &and y));
```

$$f1 := (x, y) \rightarrow Logic{:}\text{-}BooleanSimplify(Logic{:}\text{-}\&not(Logic{:}\text{-}\&and(x, y))) \qquad \textbf{(12.32)}$$

```
> f2 := (x,y) -> BooleanSimplify(&not(x) &or &not(y));
```

(12.33)

$$f2 := (x, y) \rightarrow Logic\text{:-}BooleanSimplify(Logic\text{:-}\&or(Logic\text{:-}\&not(x), Logic\text{:-}\&not(y))) \quad \textbf{(12.33)}$$

We can test the assertion that $f_1(x, y) = f_2(x, y)$ by applying the **Equivalent** command with arguments **f1(x,y)** and **f2(x,y)**.

```
> Equivalent(f1(x,y),f2(x,y));
```

$$true \quad \textbf{(12.34)}$$

Duality

We conclude this section by introducing the **Dual** command. This command accepts only one argument, a Boolean expression, and produces the dual of that expression.

For example, consider the expression $x \cdot \overline{y} + y \cdot \overline{z} + \overline{x} \cdot z$. As a logical expression, this can be written as (x **and not** y) **or** (y **and not** z) **or** (**not** x **and** z). We calculate the dual by applying the **Dual** command.

```
> Dual((x &and &not(y)) &or (y &and &not(z)) &or (&not(x) &and
  z));
```

$$((x \,\&or\, \&not(y)) \,\&and\, (y \,\&or\, \&not(z))) \,\&and\, (\&not(x) \,\&or\, z) \quad \textbf{(12.35)}$$

Similarly, the dual of $\overline{x} \cdot y + \overline{y} \cdot z + x \cdot \overline{z}$ can be computed by

```
> Dual((&not(x) &and y) &or (&not(y) &and z) &or (x &and &not
  (z)));
```

$$((\&not(x) \,\&or\, y) \,\&and\, (\&not(y) \,\&or\, z)) \,\&and\, (x \,\&or\, \&not(z)) \quad \textbf{(12.36)}$$

Note that Exercise 13 of Section 12.1 asks you to prove that the expressions $x \cdot \overline{y} + y \cdot \overline{z} + \overline{x} \cdot z$ and $\overline{x} \cdot y + \overline{y} \cdot z + x \cdot \overline{z}$ are equivalent. The duality principle implies that the duals calculated above are also equivalent. This can be verified by the **Equivalent** command.

```
> Equivalent((12.35),(12.36));
```

$$true \quad \textbf{(12.37)}$$

12.2 Representing Boolean Functions

In this section we will see how to use Maple to express Boolean functions in the disjunctive normal form (also called sum-of-products expansion). We will first look at the Maple command for turning an expression in Boolean algebra into the disjunctive normal form. Then we will see how to write a procedure for finding an expression based on a table of values.

Disjunctive Normal Form from an Expression

Given an expression written in terms of the **Logic** operators, the **Canonicalize** command can be used to transform the expression into disjunctive normal form.

As an example, consider Example 3: $(x + y)\overline{z}$. In logical form, this is (x **or** y) **and not** z. We assign this logical expression to a name.

```
> Example3 := (x &or y) &and &not(z);
```

$$Example3 := (x \,\&or\, y) \,\&and\, \&not(z) \quad \textbf{(12.38)}$$

The **Canonicalize** command has several forms. To transform an expression into disjunctive normal form, two arguments are required. The first argument is the expression to be transformed. Second, you must provide a set or list containing the variables appearing in the expression.

```
> Canonicalize(Example3,{x,y,z});
```

(12.39)

$$((\&not(z)\ \&and\ x)\ \&and\ y\ \&or\ (\&not(z)\ \&and\ x)\ \&and\ \&not(y))\ \&or\ (\&not(z)\ \&and\ y)\ \&and\ \&not(x) \tag{12.39}$$

Note that this agrees with the solution to Example 3.

The **Canonicalize** command also accepts a third argument used to specify the type of canonical form desired. The default behavior is to produce disjunctive normal form, but this can be emphasized by including the option **form=DNF**.

```
> Canonicalize(Example3,{x,y,z},form=DNF);
```

$$((\&not(z)\ \&and\ x)\ \&and\ y\ \&or\ (\&not(z)\ \&and\ x)\ \&and\ \&not(y))\ \&or\ (\&not(z)\ \&and\ y)\ \&and\ \&not(x) \tag{12.40}$$

To produce conjunctive normal form instead, you use the **form=CNF** option.

```
> Canonicalize(Example3,{x,y,z},form=CNF);
```

$$((((x\ \&or\ y)\ \&or\ z)\ \&and\ ((x\ \&or\ y)\ \&or\ \&not(z)))\ \&and\ ((x\ \&or\ \&not(y))\ \&or\ \&not(z)))\ \&and\ ((y\ \&or\ \&not(x))\ \&or\ \&not(z)))\ \&and\ ((\&not(x)\ \&or\ \&not(y))\ \&or\ \&not(z)) \tag{12.41}$$

There is a third option, **form=MOD2**, which results in the $\{0, 1\}$ canonical form. However, as we mentioned earlier, Maple interprets $+$ as the exclusive or, and thus the result of this option will be different from what is described in the textbook.

The **Normalize** command can also be used to produce a disjunctive normal form expression. Note that the second argument is not accepted by **Normalize**. Like **Canonicalize**, the default is disjunctive normal form, but **form=DNF** and **form=CNF** are both accepted by **Normalize**.

Here is the result of **Normalize** applied to Example 3.

```
> Normalize(Example3);
```

$$\&not(z)\ \&and\ x\ \&or\ \&not(z)\ \&and\ y \tag{12.42}$$

Note that the result is not the same as before. In fact, if you compare this result to the solution of Example 3, you will see that this expression is equivalent to the second line in the step-by-step expansion in the solution. In particular, it is the result of applying the distributive law to the original expression.

Normalize produces an expression in disjunctive normal form. That is, the result is a disjunction of terms with each term consisting of a conjunction of variables and their negations. **Normalize** does not, however, produce canonical disjunctive normal form, like **Canonicalize** does.

The benefit of **Normalize** is that, as you can see, it is typically simpler than canonical disjunctive normal form. The benefit of **Canonicalize** is that the canonical disjunctive normal form is unique. That is, two equivalent expressions necessarily have the same canonical disjunctive normal form, while **Normalize** may express them differently. While the textbook does not emphasize this, the examples and techniques it describes are for producing the canonical sum-of-products expansion.

Disjunctive Normal Form from a Table

Example 1 of Section 12.2 describes how to find an expression for a Boolean function represented by a table of values. Here, we will show how to write a procedure to accomplish this task.

First we must decide how we will represent the table of values in Maple. Rather than creating a

Maple table object, we'll represent the table as the set of those assignments of truth values to variables for which the function returns true.

For example, consider the function defined by the following table.

x	y	z	$F(x, y, z)$
true	*true*	*true*	*false*
true	*true*	*false*	*true*
true	*false*	*true*	*false*
true	*false*	*false*	*true*
false	*true*	*true*	*false*
false	*true*	*false*	*true*
false	*false*	*true*	*true*
false	*false*	*false*	*false*

There are four rows in the table for which the function returns true. We represent this table by forming the set consisting of the four lists of values for x, y, and z corresponding to those rows.

```
> exampleTable := {[true,true,false],[true,false,false],[false,
  true,false],[false,false,true]};
```

$$exampleTable := \{[false, false, true], [false, true, false], [true, false, false], [true, true, false]\} \quad \textbf{(12.43)}$$

This will be the first argument to the procedure we create. The procedure will also need names for the variables, so we also require a list of names. This will be the second argument to the procedure.

```
> exampleVariables := [x,y,z];
```

$$exampleVariables := [x, y, z] \quad \textbf{(12.44)}$$

To form the Boolean expression that represents this function, we follow Example 1. For each row in the table for which the function returns true, *i.e.*, for each element in **exampleTable**, we produce the corresponding minterm.

To create a minterm associated with an element of **exampleTable**, we proceed as follows. Initialize the minterm to **NULL**. Then begin a for loop with loop variable ranging from 1 to the number of variables. Within the loop, test whether or not the entry in the list of truth values is true or not. If the entry is true, then update the minterm by forming the conjunction of it with the name of the corresponding variable. If the entry is false, then update the minterm by forming the conjunction with the negation of the variable.

The following procedure accepts a single list of truth values (a single row) and the list of variables, and produces the minterm.

```
> FormMinterm:=proc(row::list(truefalse),vars::list(symbol))
    local minterm, i;
    uses Logic;
    if nops(row) <> nops(vars) then
```

```
      error "Incorrect number of variables";
    end if;
    minterm := NULL;
    for i from 1 to nops(row) do
      if row[i] then
        minterm := minterm &and vars[i];
      else
        minterm := minterm &and &not(vars[i]);
      end if;
    end do;
    return minterm;
  end proc:
```

This procedure, applied to a member of the **exampleTable**, produces the corresponding minterm.

```
> FormMinterm([true,true,false],[x,y,z]);
```

$$(x \;\&\text{and}\; y) \;\&\text{and}\; \&not(z) \qquad \textbf{(12.45)}$$

To create the Boolean expression for the function, all that remains is to form the disjunction of the minterms produced by **FormMinterm**. The procedure below accepts the table representation and list of variables as arguments. It first checks to see if it was passed the empty set, and if so, returns the expression **false**. It initializes the result to **NULL**. For each element of the table representation, it calls **FormMinterm** and adds the output from that procedure to the result with the **&or** operator.

```
> BooleanFromTable := proc(T::set(list(truefalse)),
                           V::list(symbol))
    local B, R, mt;
    if T = {} then
      return false;
    end if;
    B := NULL;
    for R in T do
      mt := FormMinterm(R,V);
      B := B &or mt;
    end do;
    return B;
  end proc:
```

Note that the structured type **set(list(truefalse))** ensures that the first argument is a **set** whose members are each a **list** with members of type **truefalse** (namely **true** or **false**).

Applying the procedure to our example table produces the desired Boolean expression.

```
> BooleanFromTable(exampleTable,exampleVariables);
```

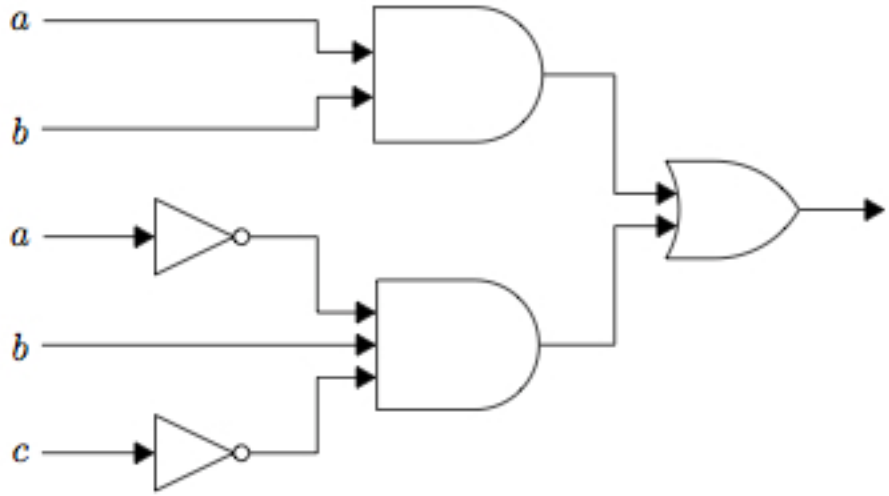
$$(((\&not(x) \;\&\text{and}\; \&not(y)) \;\&\text{and}\; z \;\&\text{or}\; (\&not(x) \;\&\text{and}\; y) \;\&\text{and}\; \&not(z)) \;\&\text{or}\; (x \;\&\text{and}\; \&not(y)) \;\&\text{and}\; \&not(z)) \;\&\text{or}\; (x \;\&\text{and}\; y) \;\&\text{and}\; \&not(z) \qquad \textbf{(12.46)}$$

12.3 Logic Gates

In this section, we will use Maple to work with logic gates, particularly circuit diagrams. First, we will see how to use Maple to translate a circuit diagram into an expression using the **Logic** operators. Then we will do the reverse and see how to transform a logical expression into a circuit diagram (modeled as a tree diagram).

Circuit Diagram to Logical Expression

Consider the circuit diagram shown below.



Our goal in this subsection is to use Maple to produce a logical expression for the output of this diagram. Each logic gate will be modeled as a procedure. We begin by creating procedures for the inverter, the OR gate, and the AND gate.

Modeling the gates

The inverter will accept only one argument (the other gates will be able to accept multiple arguments). It will return the logical negation of its input.

```
> Inverter := proc(a)
    return &not(a);
  end proc:
```

The AND and OR gates must be able to accept multiple input values, with, of course, a minimum of two. We could design the procedures to use a list or a set, but the syntax will be more natural if we use the **seq** modifier to the parameter with type **anything**. Refer to Section 6.5 for more information about the **seq** modifier.

For both procedures, we begin by forming a list from the parameter and using **nops** to ensure that there are at least two objects, otherwise an error is generated. We then initialize **result** to the first two arguments joined together by **&and** or **&or**, as appropriate. Using a for loop with loop variable ranging from 3 to the number of arguments, we add subsequent arguments to **result**.

```
> AndGate := proc(a::seq(anything))
    local result, i;
    if nops([a]) < 2 then
      error "AndGate requires at least two arguments.";
    end if;
    result := a[1] &and a[2];
    for i from 3 to nops([a]) do
      result := result &and a[i];
    end do;
    return result;
  end proc:
> OrGate := proc(a::seq(anything))
    local result, i;
    if nops([a]) < 2 then
      error "OrGate requires at least two arguments.";
    end if;
    result := a[1] &or a[2];
    for i from 3 to nops([a]) do
```

```
      result := result &or a[i];
    end do;
    return result;
  end proc:
```

For example, if a, b, and c are all input to an OR gate, we would enter the following.

```
> OrGate(a,b,c);
```

$$(a \ \&or\ b) \ \&or\ c \tag{12.47}$$

And if the result from that OR gate was sent to an inverter

```
> Inverter(%);
```

$$\&not((a \ \&or\ b) \ \&or\ c) \tag{12.48}$$

which is then joined to a with an AND gate, we obtain:

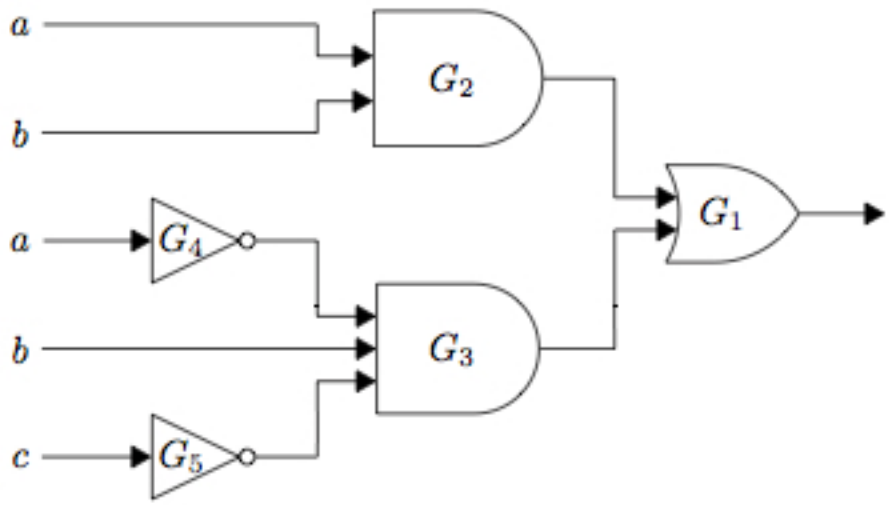
```
> AndGate(%,a);
```

$$\&not((a \ \&or\ b) \ \&or\ c) \ \&and\ a \tag{12.49}$$

Applying the procedures to a diagram

Now that we have the procedures in place, we will use them to find a logical expression for the output to the circuit diagram above.

First, give each gate in the diagram a label. The specific names are not important. We chose to label the gates using the capital letter G with subscripts numbered from the right to the left.



Interpret the labels as names for both the gates themselves and for their outputs. Also G_1 is the name for both the output of the final gate and also names the output of the circuit. The input of G_1 is the outputs from gates G_2 and G_3. That is to say, $G_1 = G_2$ **or** G_3. We can write that in Maple.

```
> G1 := OrGate(G2,G3);
```

$$G1 := G2 \ \&or\ G3 \tag{12.50}$$

For each gate, do the same. Note that the order in which the gates are specified is irrelevant. The output G_2 is a **and** b.

```
> G2 := AndGate(a,b);
```

$$G2 := a \ \&and\ b \tag{12.51}$$

The output of G_3 is the conjunction of G_4, b, and G_5.

```
> G3 := AndGate(G4,b,G5);
```

$$G3 := (G4 \ \&\text{and} \ b) \ \&\text{and} \ G5 \qquad \textbf{(12.52)}$$

And G_4 and G_5 are the results of inversion on a and c, respectively.

```
> G4 := Inverter(a);
```

$$G4 := \&not(a) \qquad \textbf{(12.53)}$$

```
> G5 := Inverter(c);
```

$$G5 := \&not(c) \qquad \textbf{(12.54)}$$

Once all of the gates have been specified, look at the value for the final gate, G_1.

```
> G1;
```

$$a \ \&\text{and} \ b \ \&\text{or} \ (\&not(a) \ \&\text{and} \ b) \ \&\text{and} \ \&not(c) \qquad \textbf{(12.55)}$$

This tells us that the circuit's result is (a **and** b) **or** (**not** a **and** b **and not** c). In {0, 1} form, this is $ab + \overline{a}b\overline{c}$.

The reason this works is that when we define the output of a gate in terms of unassigned names, such as **G2**, Maple accepts the definition. When **G2** is later assigned its own value and then the statement **G1;** is executed, Maple resolves all assigned names into their definitions so that the expression for **G1** is in terms of unassigned names (**a**, **b**, and **c**) only.

Logical Expression to Circuit Diagram

We have just seen how to use Maple to transform a circuit diagram into a logical expression for the result of the circuit. Now we consider the reverse. Given a logical expression, such as that for **G1**, we will use Maple to transform the expression into a circuit diagram.

We will model a circuit diagram as a binary tree. While circuit diagrams are generally not necessarily binary, this will serve for our purposes.

Recall that a binary tree has a number of vertices and directed edges. Vertices in the tree will correspond to gates in the circuit. One of the vertices is distinguished as the root, which will correspond to the output of the circuit. Each vertex has at most two children vertices. The edges between the vertex and its child correspond to the inputs to the gate. Each vertex other than the root has a parent, and the edge from the vertex to the parent corresponds to the output from the gate.

The assumption that a circuit can be modeled as a binary tree requires that the circuit satisfy the following properties. First, the circuit has only one output. Second, each gate has only one output. Third, each gate has at most two inputs.

Recall that in Chapter 11, we wrote the procedure **InfixToTree** for converting an algebraic expression in terms of inert versions of the binary arithmetic operators into a tree representation. We will make use of the procedures from Chapter 11 here, so we load them.

```
> with(Chapter11):
```

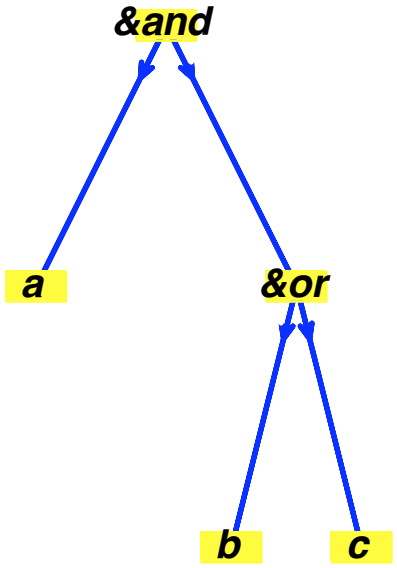
Recall that the procedures written in each chapter of this manual are collected in packages. Please refer to the Introduction if you need instructions on how to use the packages on your system.

Observe what happens if we apply the **InfixToTree** command to the logical expression a **and** (b **or** c) and then use the **DrawBTree** command to graph the tree.

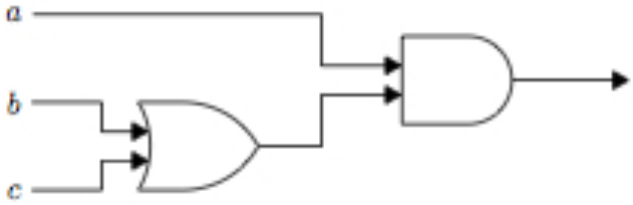
```
> InfixToTree(a &and (b &or c));
```

Graph 1: a directed unweighted graph with 5 vertices and 4 arc(s) **(12.56)**

```
> DrawBTree(%);
```



Compare the tree above to the circuit diagram below.



Observe that the diagram and the tree have the same structure. After reversing the arrows, rotating by 90°, and exchanging the symbols with the inert commands, the two are identical.

As you can see, we nearly have a procedure for turning logical expressions into trees that correspond to circuit diagrams. The only problem is that the **InfixToTree** command does not allow for unary operators such as **&not**.

Recall the definition of **InfixToTree**.

```
> eval(InfixToTree);
```

(12.57)

```
proc(e)
    local lhs, rhs, o, lhsTree, rhsTree, result;
    if type(e, {integer, symbol}) then
        result := Chapter11:-NewBTree(Chapter11:-Unique(e))
    else
        lhs := op(1, e);
        o := Chapter11:-Unique(op(0, e));
        rhs := op(2, e);
        lhsTree := Chapter11:-InfixToTree(lhs);
        rhsTree := Chapter11:-InfixToTree(rhs);
        result := Chapter11:-JoinTrees(o, lhsTree, rhsTree)
    end if;
```

```
    return result
end proc
```

In order to create a procedure like this that will work with expressions using the **Logic** operators, we need to allow for the possibility that there is only one operand.

The if-then-else statement will need an elif clause testing the number of operands of the expression, using **nops**. This will be similar to the else clause from before, except using the left hand side for the sole operand. We also will need to create a new procedure, **AddLeftBranch**, to take the place of **JoinTrees** for unary operators. Note that the else clause remains nearly identical.

```
> LogicToTree := proc(e)
  local lhs, rhs, o, lhsTree, rhsTree, result;
    uses GraphTheory, Chapter11;
    if type(e,{integer,symbol}) then
      result := NewBTree(Unique(e));
    elif nops(e) = 1 then
      lhs := op(1,e);
      o := Unique(op(0,e));
      lhsTree := LogicToTree(lhs);
      result := AddLeftBranch(o,lhsTree);
    else
      lhs := op(1,e);
      o := Unique(op(0,e));
      rhs := op(2,e);
      lhsTree := LogicToTree(lhs);
      rhsTree := LogicToTree(rhs);
      result := JoinTrees(o,lhsTree,rhsTree);
    end if;
    return result;
  end proc:
```

To implement **AddLeftBranch**, we essentially repeat the definition of **JoinTrees** with the commands related to the right child removed.

```
> AddLeftBranch := proc(newR,A::BTree)
    local newT, newVerts, Aroot, Broot, newEdges, v, e, p, w;
    uses GraphTheory;
    newVerts := [newR,op(Vertices(A))];
    Aroot := GetGraphAttribute(A,"root");
    newEdges := Edges(A) union {[newR,Aroot]};
    newT := Graph(newVerts,newEdges);
    for v in Vertices(A) do
      p := GetVertexAttribute(A,v,"order");
      SetVertexAttribute(newT,v,"order"=p);
    end do;
    SetVertexAttribute(newT,Aroot,"order"=1);
    SetVertexAttribute(newT,newR,"order"=0);
    SetGraphAttribute(newT,"root"=newR);
    return newT;
  end proc:
```

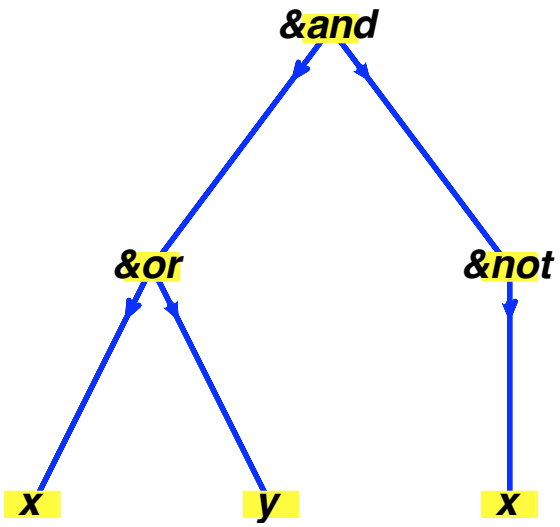
With these in place, let's look at the results for a couple of examples. First, consider $(x+y)\overline{x}$, the subject of Example 1(a) from the text.

```
> LogicToTree((x &or y) &and &not(x));
```

Graph 2: a directed unweighted graph with 6 vertices and 5 arc(s) **(12.58)**

We use **DrawORTree** to display the tree rather than **DrawBTree**, since **DrawBTree** was created expressly to ensure that "only children" would display on the left or right rather than directly below their parent. That is not needed here.

```
> DrawORTree(%);
```



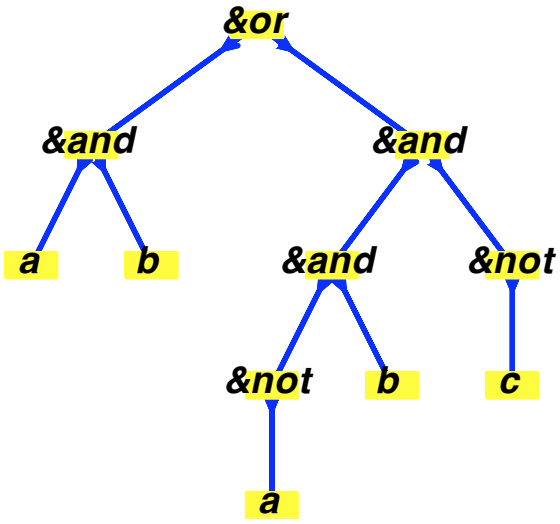
Compare this diagram to Figure 4(a) in the text.

As a second example, we apply the **LogicToTree** procedure to **G1**, the logical expression we obtained from the circuit diagram earlier.

```
> LogicToTree(G1);
```

Graph 3: a directed unweighted graph with 11 vertices and 10 arc(s) **(12.59)**

```
> DrawORTree(%);
```



Observe that this does not produce quite the same diagram as the original circuit. In particular, where the circuit had three inputs to a single AND gate, this diagram uses two binary **&and** vertices.

That suggests a way in which we could improve our procedure. In the tree above, the presence of

an `&and` as the child of another `&and` suggests that they could be combined into a single `&and` with inputs consisting of the two inputs to the lower `&and` and the right input of the higher `&and`. The result would no longer be a binary tree, as the result would have three children, but it would remain an ordered rooted tree.

It is left as an exercise to write a procedure to turn the tree above into a tree that has one `&and` with three inputs. The procedure should check each vertex and determine if it is identical to its parent. If so, the two vertices should be merged as described in the previous paragraph. Be sure to continue testing vertices until all such simplifications are complete.

Note that you will not be able to compare vertices directly, as the `Unique` procedure has guaranteed that Maple does not recognize two vertices as equal. You can determine when two vertices have the same name by converting the name into a string with the syntax `convert(name,'string')`.

12.4 Minimization of Circuits

In this section we will discuss the use of the `BooleanSimplify` command for minimizing circuits. Then we will create a brute force algorithm for handling don't care conditions. Finally, we will provide an implementation of the Quine-McCluskey method.

The BooleanSimplify Command

We described the `BooleanSimplify` command in the first section of this chapter. The command accepts only one argument, a boolean expression.

For example, we apply `BooleanSimplify` to G1, the expression we obtained for the output of the circuit diagram at the beginning of Section 12.3.

```
> BooleanSimplify(G1);
```

$$a \ \&\text{and}\ b \ \&\text{or}\ b \ \&\text{and}\ \&not(c) \qquad \textbf{(12.60)}$$

The result indicates that **not** *a* can be removed as an input to the second AND gate.

Note that the result of `BooleanSimplify` is guaranteed to be minimal. That is, it is not possible to reduce it further. It is not, however, guaranteed that the result is a minimum sum of prime implicants. That is, while no simplification of the output from `BooleanSimplify` is possible, it may be the case that there is a simpler expression equivalent to the original input.

The reason for this lies in the final step of the Quine-McCluskey method. Once the essential prime implicants have been identified, you are left with prime implicants that are not essential and you must identify the best choice of those prime implicants that will complete the cover. To find the minimum expression, you use a backtracking approach (a depth-first search). Unfortunately, this requires exponential time.

The alternative is to use a heuristic approach, choosing the prime implicants that cover the most minterms. This is considerably more efficient, but does not guarantee that the resulting expression is the minimum. At the conclusion of this section, we will design an implementation of the Quine-McCluskey method using such a heuristic approach. Implementing a backtracking approach is left as an exercise.

Don't Care Conditions

Informally, a set of don't care conditions for a boolean function *F* is a set of points in the domain of *F* whose images do not concern us.

If F is a function on n variables, then its domain is $\{true, false\}^n$. Let A be the subset of $\{true, false\}^n$ for which the value of F is specified. If we think of F as fully defined on this subset A, then we are interested in the family of all extensions of F to all of $\{true, false\}^n$. In other words, the set of all G defined on $\{true, false\}^n$ that agree with F on A. The goal is to choose the particular G that is "simplest." That is, the G that has the smallest sum of products expansion.

We should pause to consider the size of this problem. If there are d don't care points, then there are 2^d possible extensions G. Considering every possible extension can become rather time consuming.

The procedure we write will make use of the **BooleanFromTable** procedure from Section 12.2. Recall that the **BooleanFromTable** procedure accepted a set consisting of those points for which the function returns true. The points are represented by lists of trues and falses. It also required a list of the names of the variables. **BooleanFromTable** returns the conjunctive normal form of the function that returns true on the specified points and false on all others.

In **DontCare**, we will loop through every possible extension G of the input function F. Specifically, the procedure will accept two sets of points. One representing those points for which the function must return true, and the second set of points representing the don't care conditions. it is understood that the function must return false on all points in neither set. **DontCare** will also accept a list of variable names.

Each extension G corresponds to a subset of the don't care conditions. We will use the **subsets** command, described in Section 6.1 of this manual. Recall that **subsets**, when applied to a set, returns a table with two elements. The **finished** element is a boolean value set to true once all of the subsets of the given set have been listed. The **nextvalue** entry is a procedure that, when executed, produces the next subset.

For each subset of the don't care conditions, **DontCare** will apply **BooleanFromTable** to the union of the subset and the set of points for which F must be true. It will then apply **BooleanSimplify** to the result.

To determine the minimal expression, we will compare expressions using the **length** command, which returns an integer representing the length of the expression. (This is a crude comparison, but it will suffice.) We use the standard approach of storing the simplest expression found so far, and replacing it each time a shorter expression is located. Note that the first set produced by **subsets** is always the empty set, so we initialize the temporary minimum to the function in which all don't care conditions are taken to be false.

```
> DontCare := proc(T::set(list(truefalse)),
                   DC::set(list(truefalse)),V::list(symbol))
    local minExpr, minLength, S, s, nextExpr;
    S := combinat[subsets](DC);
    s := S[nextvalue]();
    minExpr := BooleanSimplify(BooleanFromTable(T,V));
    minLength := length(minExpr);
    while not S[finished] do
      s := S[nextvalue]();
      nextExpr:=BooleanSimplify(BooleanFromTable(T union s,V));
      if length(nextExpr) < minLength then
        minExpr := nextExpr;
        minLength := length(nextExpr);
```

```
      end if;
    end do;
    return minExpr;
  end proc:
```

Consider the boolean function F defined by the following table of values, in which "d" in the final column indicates a don't care condition.

x	y	z	$F(x, y, z)$
false	*false*	*false*	*true*
false	*false*	*true*	*false*
false	*true*	*false*	*d*
false	*true*	*true*	*d*
true	*false*	*false*	*true*
true	*false*	*true*	*false*
true	*true*	*false*	*false*
true	*true*	*true*	*true*

The points that must evaluate to true are:

$$\{[false, false, false], [true, false, false], [true, true, true]\}.$$

And the don't care conditions are:

$$\{[false, true, false], [false, true, true]\}.$$

We apply the procedure **DontCare** with these two sets as inputs along with the list of variables.

```
> DontCare({[false,false,false],[true,false,false],[true,true,
  true]},{[false,true,false],[false,true,true]},['x','y','z']);
```

$$y \,\&\text{and}\, z \,\&\text{or}\, \&not(y) \,\&\text{and}\, \&not(z) \qquad \textbf{(12.61)}$$

Before leaving don't care conditions, we should mention that the Quine-McCluskey method, which is the subject of the next subsection, provides a much more efficient solution than the procedure **DontCare**.

To take don't care conditions into account with the Quine-McCluskey method, you include them in the list of minterms that are used to generate prime implicants, but you do not include them in the list of minterms that need to be covered by the prime implicants. In terms of Example 9 of the text, don't care conditions appear in the first column of Table 3, but are omitted from the top row of Table 4.

Quine-McCluskey

We conclude with an implementation of the Quine-McCluskey method. This method is fairly involved and it will take considerable effort to implement it correctly.

It will be helpful to have an example that we can use to illustrate the method as we build the procedure. The expression we use for the example is

$$\overline{wxyz} + w\overline{x}yz + w\overline{x}\,\overline{yz} + \overline{wxyz} + \overline{wxyz} + \overline{wx}\,\overline{y}z + \overline{wxyz} + \overline{wxyz} + \overline{wxyz} + \overline{wxyz}.$$

We assign this to the name **F**.

```
> F := (w &and x &and &not(y) &and &not(z)) &or
       (w &and &not(x) &and y &and z) &or
       (w &and &not(x) &and y &and &not(z)) &or
       (w &and &not(x) &and &not(y) &and &not(z)) &or
       (&not(w) &and x &and y &and z) &or
       (&not(w) &and x &and &not(y) &and z) &or
       (&not(w) &and x &and &not(y) &and &not(z)) &or
       (&not(w) &and &not(x) &and y &and z) &or
       (&not(w) &and &not(x) &and &not(y) &and z) &or
       (&not(w) &and &not(x) &and &not(y) &and &not(z)):
```

Let us begin by (very) briefly outlining the approach. More details will be given as we proceed.

1. Transform the minterms into bit strings.
2. Group the bit strings by the number of 1s.
3. Combine bit strings that differ in exactly one location.
4. Repeat steps 2 and 3 until no additional combinations are possible.
5. Identify the prime implicants (those bit strings not involved in a simplification) and form the coverage table.
6. Identify the essential prime implicants and update the table.
7. Process the remaining prime implicants using a heuristic approach in order to achieve complete coverage.

Implementing this will require several different procedures that will come together to achieve the goal of minimizing the expression for *F*.

Modifying arguments

Before we begin implementing the method, we take a moment to reiterate the use of **evaln** as a parameter modifier, which we first described in Section 11.2. This will be used later in the implementation to avoid the need to copy data structures that must be modified by a procedure.

Modifying a parameter with **evaln** means that, instead of sending a copy of the object to be worked on by the procedure, you send the name of the object. This allows the argument to be modified directly within the procedure. However, every time you want to access the value of the object, rather than the name, you must apply **eval** to the parameter.

The procedure below adds 3 to its argument, changing the value stored in the name it is passed.

```
> add3 := proc(x::evaln(integer))
    x := eval(x) + 3;
  end proc:
```

Applying **add3** to a name that stores an integer will now alter the value stored in the name.

```
> two := 2;
```

$$two := 2 \tag{12.62}$$

```
> add3(two);
```

$$5 \tag{12.63}$$

```
> two;
```

$$5 \tag{12.64}$$

Transforming minterms into bit strings

The first task is to process the input. That is, *F* must be transformed into a list of bit strings. This is not strictly necessary, but it makes working with the minterms more convenient. We represent bit

strings as lists of 0s and 1s.

We begin by creating a procedure to transform a single minterm into a bit string. We assume that the input to this procedure will be a properly formed minterm, that is, a conjunction of variables and negations of variables. We require that a list of variables be provided to the procedure, so that the bit string can be formed in the proper order.

Consider the following minterm, which is the fourth minterm in our example *F*.

```
> minterm := w &and &not(x) &and y &and &not(z);
```

$$minterm := ((w \,\&and\, \&not(x)) \,\&and\, y) \,\&and\, \&not(z) \qquad \textbf{(12.65)}$$

In order to transform this into the bit string [1, 0, 1, 0], we must first determine the variables and negations of variables that are conjoined. Our first goal, therefore, is to transform the minterm into the list [*w*,**not** *x*, *y*,**not** *z*].

For this, recall that the **op** command can be used to extract the operands of an expression. In this case, **op** will return the two operands of the final **&and**.

```
> [op(minterm)];
```

$$[(w \,\&and\, \&not(x)) \,\&and\, y, \&not(z)] \qquad \textbf{(12.66)}$$

In order to obtain a list of the conjoined variables and negations, we will repeatedly apply **op**.

We begin by initializing a list consisting of **minterm** as the only object and we set an index variable equal to 1.

```
> MTList := [minterm]; i := 1;
```

$$MTList := [(w \,\&and\, \&not(x)) \,\&and\, y) \,\&and\, \&not(z)]$$

$$i := 1 \qquad \textbf{(12.67)}$$

We create a while loop that will continue as long as the index variable is not greater than the length of **MTList**. Within the loop, we consider **MTList[i]**. Comparing **op(0,MTList[i])** to the operator **`&and`** will tell us whether or not the *i*th member of the list is a conjunction. If so, we apply **op** to it, using **subsop** to replace the **i** location in the list with **op(MTList[i])**. On the other hand, if **op(0,MTList[i])** is not **`&and`**, then we increment **i**.

```
> while i <= nops(MTList) do
    if op(0,MTList[i]) = `&and` then
      MTList := subsop(i=op(MTList[i]),MTList);
    else
      i := i + 1;
    end if;
  end do;
```

This has transformed **MTList** into a list of the conjoined variables and negations of variables.

```
> MTList;
```

$$[w, \&not(x), y, \&not(z)] \qquad \textbf{(12.68)}$$

To complete the transformation into a bit string, we only need to check, for each variable, whether the variable or its negation is in the list. Recall that we will insist that the procedure be given the list of variables as an argument to maintain the proper order of the variables.

We first assign the list of variables to a name.

```
> variableList := [w,x,y,z];
```

$$variableList := [w, x, y, z] \qquad \textbf{(12.69)}$$

Now create a list, initialized to the proper length, for the bit string.

```
> Bitstring := [0 $ nops(variableList)];
```

$$Bitstring := [0, 0, 0, 0] \quad \textbf{(12.70)}$$

Finally, we use a for loop to check, for each variable, whether the variable is in **MTList**. If the variable is a member of **MTList**, then we change the bit to 1. Otherwise, we assume that the negation is in the minterm and we leave the value in the bit string as 0.

```
> for i from 1 to nops(variableList) do
    if variableList[i] in MTList then
      Bitstring[i] := 1;
    end if;
  end do;
```

This has created the bit string associated to **minterm**.

```
> Bitstring;
```

$$[1, 0, 1, 0] \quad \textbf{(12.71)}$$

We condense this process into a single procedure.

```
> MTtoBitString := proc(minterm,variableList::list(symbol))
    local MTList, i, Bitstring;
    MTList := [minterm];
    i := 1;
    while i <= nops(MTList) do
      if op(0,MTList[i]) = `&and` then
        MTList := subsop(i=op(MTList[i]),MTList);
      else
        i := i + 1;
      end if;
    end do;
    Bitstring := [0 $ nops(variableList)];
    for i from 1 to nops(variableList) do
      if variableList[i] in MTList then
        Bitstring[i] := 1;
      elif `&not`(variableList[i]) in MTList then
        Bitstring[i] := 0;
      else
        error "Unrecognized object in MTList.";
      end if;
    end do;
    return Bitstring;
  end proc:
> MTtoBitString(minterm,[w,x,y,z]);
```

$$[1, 0, 1, 0] \quad \textbf{(12.72)}$$

Transforming the original expression into bit strings

Now that we have the means for transforming a single minterm into a bit string, we are ready to transform an expression in disjunctive normal form into a list of bit strings.

This works in nearly the same way as **MTtoBitString** did. Given an expression in disjunctive normal form, we break it into a list, **DNFList**, but this time, instead of looking for the operator to be **`&and`**, it must be **`&or`**. Once the list is formed, we apply **MTtoBitString** on each element of the list.

Here is the procedure. Notice that the first while loop is very similar to **MTtoBitString**, but

after the while loop, we complete the procedure with an application of **map**. We use **variableList** as a third argument to **map**. When **map** is given more than two arguments, subsequent arguments are treated as additional arguments to the procedure. That is, **map(P,L,a)** applies the procedure **P** to **(l,a)** for each **l** in the list **L**. In this case, **MTtoBitString** requires that the list of variables be passed to it each time it is called.

```
> DNFtoBitList := proc(dnfExpr,variableList::list(symbol))
    local DNFList, i;
    DNFList := [dnfExpr];
    i := 1;
    while i <= nops(DNFList) do
      if op(0,DNFList[i]) = `&or` then
        DNFList := subsop(i=op(DNFList[i]),DNFList);
      else
        i := i + 1;
      end if;
    end do;
    return map(MTtoBitString,DNFList,variableList);
  end proc:
```

Apply this procedure to the example expression.

```
> Fbits := DNFtoBitList(F,[w,x,y,z]);
```

$$Fbits := [[1, 1, 0, 0], [1, 0, 1, 1], [1, 0, 1, 0], [1, 0, 0, 0], [0, 1, 1, 1], [0, 1, 0, 1], [0, 1, 0, 0], [0, 0, 1, 1], [0, 0, 0, 1], [0, 0, 0, 0]] \quad \textbf{(12.73)}$$

Transforming bit strings into minterms

At the conclusion of the Quine-McCluskey process, we will want to display the result in disjunctive normal form. This will require that we turn bit strings back into minterms.

Note that since this procedure will be applied at the end of the process, it may be that some of the variables have been removed. We will be using the string "-" in a bit string to indicate the elimination of a variable.

This procedure will require the bit string and a list of variable names as its input. It operates in two stages. First, it processes the variable list based on the content of the bit string. We make a copy of the variable list and then modify it by applying **&not** when the entry in the bit string is 0 and by replacing the variable with "-" when that is the entry in the bit string.

```
> varList := [w,x,y,z];
```

$$varList := [w, x, y, z] \quad \textbf{(12.74)}$$

```
> bitstr := [0,1,"-",0];
```

$$bitstr := [0, 1, "-", 0] \quad \textbf{(12.75)}$$

```
> for i from 1 to nops(varList) do
    if bitstr[i] = 0 then
      varList[i] := &not(varList[i]);
    elif bitstr[i] = "-" then
      varList[i] := "-";
    end if;
  end do;
> varList;
```

$$[\&not(w), x, "-", \&not(z)] \quad \textbf{(12.76)}$$

Once this processing has been done, we remove any occurrences of "-" by replacing them with

NULL. The **subs** command accepts as its arguments a sequence of equations and a final expression. For example **subs(x=a,expr)**. The result of this statement is that every occurrence of **x** in **expr** is replaced by **a**. With the equation **"-"=NULL** and the processed variable list as the arguments, **subs** will return the list with each **"-"** removed.

```
> varList := subs("-"=NULL,varList);
```

$$varList := [\&not(w), x, \&not(z)] \quad \textbf{(12.77)}$$

To form the conjunction, we recreate the **AndGate** procedure from earlier, but modify the behavior when the input has fewer than two elements.

```
> AndList := proc(L::list)
    local result, i;
    if nops(L) = 0 then
      return NULL;
    elif nops(L) = 1 then
      return op(L);
    else
      result := L[1] &and L[2];
      for i from 3 to nops(L) do
        result := result &and L[i];
      end do;
      return result;
    end if;
  end proc:
```

Applying this to the modified **varList** produces the minterm.

```
> AndList(varList);
```

$$(\&not(w)\ \&and\ x)\ \&and\ \&not(z) \quad \textbf{(12.78)}$$

We combine this into a procedure.

```
> BitStringtoMT := proc(bitstring,variableList::list(symbol))
    local varList, i;
    varList := variableList;
    for i from 1 to nops(varList) do
      if bitstring[i] = 0 then
        varList[i] := &not(varList[i]);
      elif bitstring[i] = "-" then
        varList[i] := "-";
      end if;
    end do;
    varList := subs("-"=NULL,varList);
    return AndList(varList);
  end proc:
```

Applied to [0, 1, 0, 1] and [*x*, *y*, *z*, *w*], we see that **BitStringtoMT** reproduces the original minterm.

```
> BitStringtoMT([0,1,0,1],[w,x,y,z]);
```

$$((\&not(w)\ \&and\ x)\ \&and\ \&not(y))\ \&and\ z \quad \textbf{(12.79)}$$

And applied to [0, 1, "-", 1], it removes the *y*.

```
> BitStringtoMT([0,1,"-",1],[w,x,y,z]);
```

$$(\&not(w)\ \&and\ x)\ \&and\ z \quad \textbf{(12.80)}$$

The final result of our Quine-McCluskey process will be a list of bit strings. To produce the

associated disjunctive normal form expression, we only need to join the individual minterms produced by **BitStringtoMT** with **&or**s. We define the needed **OrList**.

```
> OrList := proc(L::list)
    local result, i;
    if nops(L) = 0 then
      return NULL;
    elif nops(L) = 1 then
      return op(L);
    else
      result := L[1] &or L[2];
      for i from 3 to nops(L) do
        result := result &or L[i];
      end do;
      return result;
    end if;
  end proc:
```

This is identical to **AndList**, except with **&or** replacing **&and**.

Initializing the source table

In order to form the coverage table in the second part of the method, we need to know which of the original minterms are covered by which of the prime implicants. Refer to Tables 3 and 6 in the text. Notice that each bit string in those tables is associated with either a single number, in the case of the original minterms, or lists of numbers, for the derived products.

We will store this information in a table whose indices are the bit strings and whose entries are sets of integers. Given the **Fbits** list, we initialize this table with the elements of **Fbits** as the indices. The corresponding entries will be the set consisting of the bit string's position in **Fbits**.

We will refer to this as the "coverage dictionary," since it allows us to look up any bit string and determine all of the original minterms covered by it. The following procedure accepts the **Fbits** list as an argument and returns the coverage dictionary.

```
> initCoverDict := proc(L::list)
    local coverDict, i;
    coverDict := table();
    for i from 1 to nops(L) do
      coverDict[L[i]] := {i};
    end do;
    return coverDict;
  end proc:
```

Applying this procedure to **Fbits** produces the initial coverage dictionary. In order to inspect the entries, we must apply **eval** to the name of the table.

```
> coverageDict := initCoverDict(Fbits);
```

$$coverageDict := coverDict \tag{12.81}$$

```
> eval(coverageDict);
```

$$table([[0,1,0,0]={7},[0,0,0,0]={10},[1,0,1,1]={2},[0,0,1,1]={8},[0,1,1,1]={5},[1,0,1,0]={3},[0,0,0,1]={9},[0,1,0,1]={6},[1,1,0,0]={1},[1,0,0,0]={4}]) \tag{12.82}$$

Grouping by the number of 1s

Step 2 in our outline is to group the bit strings by the number of 1s.

The reason for this step is to improve the efficiency of finding simplifications to make. Since two bit strings can be combined only when they are identical except for one location, the only possible combinations are when one bit string has n 1s and the other has $n - 1$.

After step 1 is concluded, we have a list of bit strings. That will be the starting point for the procedure we create for this step. The result of this step will be to turn the list of bit strings into a list of sets of bit strings, which we'll call **groups**. In location i of **groups** will be the set of all bit strings with $i - 1$ 1s.

We know that the number of 1s in any bit string must be between 0 and the length of the bit string. We initialize **groups** to be the list of empty sets. The maximum number of 1s in equal to the length of a bit string, which we can obtain from the size of the first element of **Fbits**.

```
> groups := [ {} $ nops(Fbits[1])+1 ];
```

$$groups := [\{\}, \{\}, \{\}, \{\}, \{\}] \quad \textbf{(12.83)}$$

Since the bit strings had four entries, **groups** now consists of five copies of the empty set.

For each member of **Fbits**, we need to count the number of 1s. We will create a small procedure to do this.

```
> count1s := proc(L::list)
    local c, i;
    c := 0;
    for i from 1 to nops(L) do
      if L[i] = 1 then
        c := c + 1;
      end if;
    end do;
    return c;
  end proc:
> count1s([1,0,1,1,0,0,1]);
```

$$4 \quad \textbf{(12.84)}$$

It is tempting to use the **add** command to add the bits in the list. We cannot do this, however, because after the first simplification, our bit strings will contain symbols that are not 1s or 0s.

We use **count1s** to sort the members of **Fbits** into **groups**. Using a for loop to step through the **Fbits** list, we apply **count1s** and add 1 to the result (since the bit strings with no 1s are in the first position). We then update that set in **groups** using **union**.

Here is the procedure implementing this.

```
> sortGroups := proc(bitstringList)
    local groups, i, c;
    groups := [ {} $ nops(bitstringList[1])+1 ];
    for i from 1 to nops(bitstringList) do
      c := count1s(bitstringList[i]);
      groups[c+1] := groups[c+1] union {bitstringList[i]};
    end do;
    return groups;
  end proc:
```

Here is the result of sorting **Fbits**.

```
> groups := sortGroups(Fbits);
```

$$groups := [\{[0,0,0,0]\}, \{[0,0,0,1], [0,1,0,0], [1,0,0,0]\}, \{[0,0,1,1], [0,1,0,1], [1,0,1,0], [1,1,0,0]\}, \{[0,1,1,1], [1,0,1,1]\}, \{\}] \quad \textbf{(12.85)}$$

Combining bit strings

Step 3 is to combine all of the bit strings that differ in exactly one location. We first write a procedure that takes as input two bit strings and either combines them if, in fact, they do differ in exactly one location, or returns false if they do not.

This procedure need to do two tasks. First, it has to check to see whether or not the two bit strings differ in more than one location. Second, it needs to combine them if they are allowed to be combined.

Combining two bit strings is easy, provided we know the one location in which they differ. For example,

```
> bit1 := [1,"-",0,1,1];
```

$$bit1 := [1, "-", 0, 1, 1] \quad \textbf{(12.86)}$$

```
> bit2 := [1,"-",0,0,1];
```

$$bit2 := [1, "-", 0, 0, 1] \quad \textbf{(12.87)}$$

You can see that these are identical except in position 4.

To merge them, we take either one and replace position 4 with "-".

```
> subsop(4="-",bit1);
```

$$[1, "-", 0, "-", 1] \quad \textbf{(12.88)}$$

We determine that they differ only in position 4 as follows. Begin by initializing a name **pos**, for position, to 0. This will hold the position at which the difference occurs. Setting it to 0 indicates that we have not found a difference.

Now use a for loop to compare each pair of entries in **bit1** and **bit2**. If we find a difference, check the value of **pos**. If **pos** is 0, then we know that this is the first time a difference was found and we set **pos** to the position of the difference. If **pos** is not 0, however, then we know that this is the second time a difference was found. In this case, the bit strings cannot be merged and we return false. If the loop completes without having returned false, then the two bit strings can be merged at position **pos**.

Here is the procedure.

```
> MergeBitstrings := proc(bit1::list,bit2::list)
    local i, pos;
    pos := 0;
    for i from 1 to nops(bit1) do
      if bit1[i] <> bit2[i] then
        if pos = 0 then
          pos := i;
        else
          return false;
        end if;
      end if;
    end do;
    return subsop(pos="-",bit1);
  end proc:
```

We see that it works correctly on our two example bit strings.

```
> MergeBitstrings(bit1,bit2);
```

$$[1, "-", 0, "-", 1] \quad \textbf{(12.89)}$$

Searching for combinations to make

The **MergeBitstrings** procedure will do the work of checking to see if bit strings can be merged and returning the result if they can. However, we need to give **MergeBitstrings** the bit strings to test.

Recall that, in our example, we have successfully grouped the minterms by the number of 1s they contain.

```
> groups;
```

$$[\{[0,0,0,0]\},\{[0,0,0,1],[0,1,0,0],[1,0,0,0]\},\{[0,0,1,1],[0,1,0,1],[1,0,1,0],[1,1,0,0]\},\{[0,1,1,1],[1,0,1,1]\},\{\}] \quad (12.90)$$

Here, we will produce a list containing all the bit strings formed by merging two members of **groups**. Since there may be multiple ways to obtain the same bit string, we store these as a set. We initialize to the empty set.

```
> Fbits1 := {};
```

$$Fbits1 := \{\} \quad (12.91)$$

Also recall that it is only possible to merge bit strings that are in successive locations in **groups**. In other words, we only need to check bit strings when one has n 1s and one has $n-1$ 1s. This suggests a for loop with **n** ranging from 1 to one less than the number of sets in **groups**. Within the body of the for loop, we will consider the sets with $n-1$ 1s (index **n**) and the set with n 1s (index **n+1**). (Remember that **groups[1]** is the set of bit strings with 0 1s.)

The loop is structured as follows.

```
> for n from 1 to nops(groups)-1 do
    A := groups[n];
    B := groups[n+1];
    #  look for bit strings from A and B to merge
  end do:
```

After A and B have been defined, we need to compare every possible pair. We use two more for loops, one for each member of A and one for each member of B. Within the inner for loop, we use **MergeBitstrings** and store the result. If it is not false, we add it to the new list of bit strings, **Fbits1**.

```
> for n from 1 to nops(groups)-1 do
    A := groups[n];
    B := groups[n+1];
    for a in A do
      for b in B do
        m := MergeBitstrings(a,b);
        if m <> false then
          Fbits1 := Fbits1 union {m};
        end if;
      end do;
    end do;
  end do:
> Fbits1 := [op(Fbits1)];
```

$$Fbits1 := [[0,0,0,"-"],[0,0,"-",1],[0,1,0,"-"],[0,1,"-",1],[0,"-",0,0],[0,"-",0,1],[0,"-",1,1],[1,0,1,"-"],[1,0,"-",0],[1,"-",0,0],["-",0,0,0],["-",0,1,1],["-",1,0,0]] \quad (12.92)$$

This is close to the procedure we want, but we need to think ahead a bit. Recall from the description of the Quine-McCluskey process in the text that, in order to proceed with the second half of the method, we need to know which of the bit strings are prime implicants. That is, which bit strings are never used in a simplification.

We will track which bit strings are used as follows. Before the first loop, we create a set consisting of all of the bit strings in groups. We can do this using the functional `` `union` `` applied to the operands of groups.

```
> `union`(op(groups));
```

$$\{[0,0,0,0],[0,0,0,1],[0,0,1,1],[0,1,0,0],[0,1,0,1],[0,1,1,1],[1,0,0,0],[1,0,1,0],[1,0,1,1],[1,1,0,0]\} \quad \textbf{(12.93)}$$

Then each time `MergeBitstrings` returns true, we remove the pair of bit strings from this set, using the `minus` set operator.

The procedure will return the sequence consisting of the next level of bit strings and the prime implicants from this stage. Here is our second attempt at the procedure.

```
> NextBitList := proc(lastgroups)
    local nextL, primeImps, n, A, B, a, b, m;
    nextL := {};
    primeImps := `union`(op(lastgroups));
    for n from 1 to nops(lastgroups)-1 do
      A := lastgroups[n];
      B := lastgroups[n+1];
      for a in A do
        for b in B do
          m := MergeBitstrings(a,b);
          if m <> false then
            nextL := nextL union {m};
            primeImps := primeImps minus {a,b};
          end if;
        end do;
      end do;
    end do;
    nextL := [op(nextL)];
    return nextL,primeImps;
  end proc:
```

This still isn't sufficient, however, because we also need to update the coverage dictionary as we create new bit strings. Recall that "coverage dictionary" is the name we gave to the table that records, for each bit string, which of the original minterms are covered by that bit string. The coverage dictionary was initialized with the bit strings formed from the minterms.

```
> eval(coverageDict);
```

$$table([[0,1,0,0]=\{7\},[0,0,0,0]=\{10\},[1,0,1,1]=\{2\},[0,0,1,1]=\{8\},[0,1,1,1]=\{5\},[1,0,1,0]=\{3\},[0,0,0,1]=\{9\},[0,1,0,1]=\{6\},[1,1,0,0]=\{1\},[1,0,0,0]=\{4\}]) \quad \textbf{(12.94)}$$

Within the `NextBitList` procedure, we need to update the coverage dictionary. We will make the dictionary a parameter. Note that it will not be necessary to return the updated dictionary, nor is it necessary to make a copy of the parameter. This is because, as we mentioned in Section 2.6 of this manual, tables are "reference types," so unlike most arguments, they are altered by the procedure.

We update the dictionary within the `m <> false` if statement. When we form a new bit string `m`, we obtain the set of minterms it covers by taking the union of the sets of minterms covered by the two bit strings that were merged. That is,

```
coverDict[m] := coverDict[a] union coverDict[b];.
```

Note that bit strings formed beyond the first step are typically generated multiple times. However, each time they are generated they always cover the same set of original minterms.

Here is the final version of `NextBitList`.

```
> NextBitList := proc(lastgroups,coverDict)
    local nextL, primeImps, n, A, B, a, b, m;
    nextL := {};
    primeImps := `union`(op(lastgroups));
    for n from 1 to nops(lastgroups)-1 do
      A := lastgroups[n];
      B := lastgroups[n+1];
      for a in A do
        for b in B do
          m := MergeBitstrings(a,b);
          if m <> false then
            nextL := nextL union {m};
            primeImps := primeImps minus {a,b};
            coverDict[m]:=coverDict[a] union coverDict[b];
          end if;
        end do;
      end do;
    end do;
    nextL := [op(nextL)];
    return nextL,primeImps;
  end proc:
```

We apply it to `groups` to obtain `Fbits1` and `primes1`.

```
> Fbits1,primes1 := NextBitList(groups,coverageDict);
```

$$Fbits1, primes1 := [[0, 0, 0, "-"], [0, 0, "-", 1], [0, 1, 0, "-"], [0, 1, "-", 1], [0, "-", 0, 0], [0, "-", 0, 1], [0, "-", 1, 1], [1, 0, 1, "-"], [1, 0, "-", 0], [1, "-", 0, 0], ["-", 0, 0, 0], ["-", 0, 1, 1], ["-", 1, 0, 0]], \{\} \quad \textbf{(12.95)}$$

We see that there are

```
> nops(Fbits1);
```

$$13 \quad \textbf{(12.96)}$$

bit strings in the second level, but no prime implicants coming from the first pass.

```
> nops(primes1);
```

$$0 \quad \textbf{(12.97)}$$

Also, almost as a side effect, the procedure has updated `coverageDict`.

```
> eval(coverageDict);
```

$$table([[0, 1, 0, 0] = \{7\}, ["-", 0, 1, 1] = \{2, 8\}, [0, 0, 0, 0] = \{10\}, [1, 0, 1, 1] = \{2\}, [0, 0, 1, 1] = \{8\}, [1, 0, "-", 0] = \{3, 4\}, ["-", 0, 0, 0] = \{4, 10\}, [0, 1, 1, 1] = \{5\}, [0, 1, "-", 1] = \{5, 6\}, [1, 0, 1, 0] = \{3\}, [0, 0, 0, 1] = \{9\}, [0, "-", 0, 0] = \{7, 10\}, [1, "-", 0, 0] = \{1, 4\}, [0, 1, 0, 1] = \{6\}, [0, "-", 0, 1] = \{6, 9\}, [0, 1, 0, "-"] = \{6, 7\}, [1, 1, 0, 0] = \{1\}, [1, 0, 0, 0] = \{4\}, [0, 0, 0, "-"] = \{9, 10\}, [0, 0, "-", 1] = \{8, 9\}, [1, 0, 1, "-"] = \{2, 3\}, ["-", 1, 0, 0] = \{1, 7\}, [0, "-", 1, 1] = \{5, \quad \textbf{(12.98)}$$

```
    8}])
```

Repeating
Step 4 is to repeat steps 2 and 3.

The **Fbits1** list takes the place of **Fbits**. We apply **sortGroups** to produce **groups1**.

```
> groups1 := sortGroups(Fbits1);
```

$$groups1 := [\{[0, 0, 0, "-"], [0, "-", 0, 0], ["-", 0, 0, 0]\}, \{[0, 0, "-", 1], [0, 1, 0, "-"], [0, "-", 0, 1], [1, 0, "-", 0], [1, "-", 0, 0], ["-", 1, 0, 0]\}, \{[0, 1, "-", 1], [0, "-", 1, 1], [1, 0, 1, "-"], ["-", 0, 1, 1]\}, \{\}, \{\}] \quad (12.99)$$

Then applying **NextBitList** to **groups1** produces **Fbits2** and **primes2**.

```
> Fbits2,primes2 := NextBitList(groups1,coverageDict);
```

$$Fbits2, primes2 := [[0, "-", 0, "-"], [0, "-", "-", 1], ["-", "-", 0, 0]], \{[1, 0, 1, "-"], [1, 0, "-", 0], ["-", 0, 1, 1]\} \quad (12.100)$$

We see that we have found three prime implicants. The coverage dictionary was further expanded to include the new bit strings.

```
> eval(coverageDict);
```

$$table([[0, 1, 0, 0] = \{7\}, ["-", 0, 1, 1] = \{2, 8\}, [0, 0, 0, 0] = \{10\}, [1, 0, 1, 1] = \{2\}, [0, 0, 1, 1] = \{8\}, [1, 0, "-", 0] = \{3, 4\}, ["-", 0, 0, 0] = \{4, 10\}, [0, 1, 1, 1] = \{5\}, [0, 1, "-", 1] = \{5, 6\}, [1, 0, 1, 0] = \{3\}, ["-", "-", 0, 0] = \{1, 4, 7, 10\}, [0, 0, 0, 1] = \{9\}, [0, "-", 0, 0] = \{7, 10\}, [1, "-", 0, 0] = \{1, 4\}, [0, 1, 0, 1] = \{6\}, [0, "-", 0, 1] = \{6, 9\}, [0, 1, 0, "-"] = \{6, 7\}, [1, 1, 0, 0] = \{1\}, [0, "-", 0, "-"] = \{6, 7, 9, 10\}, [1, 0, 0, 0] = \{4\}, [0, 0, 0, "-"] = \{9, 10\}, [0, 0, "-", 1] = \{8, 9\}, [0, "-", "-", 1] = \{5, 6, 8, 9\}, [1, 0, 1, "-"] = \{2, 3\}, ["-", 1, 0, 0] = \{1, 7\}, [0, "-", 1, 1] = \{5, 8\}]) \quad (12.101)$$

Do the same thing again with **Fbits2**.

```
> groups2 := sortGroups(Fbits2);
```

$$groups2 := [\{[0, "-", 0, "-"], ["-", "-", 0, 0]\}, \{[0, "-", "-", 1]\}, \{\}, \{\}, \{\}] \quad (12.102)$$

```
> Fbits3,primes3 := NextBitList(groups2,coverageDict);
```

$$Fbits3, primes3 := [\,], \{[0, "-", 0, "-"], [0, "-", "-", 1], ["-", "-", 0, 0]\} \quad (12.103)$$

This time, **Fbits3** was empty, which indicates that no more merging is possible and all prime implicants have been found.

This part of the process concludes by forming the list of all the prime implicants.

```
> allprimeImps := [op(primes1 union primes2 union primes3)];
```

$$allprimeImps := [[0, "-", 0, "-"], [0, "-", "-", 1], [1, 0, 1, "-"], [1, 0, "-", 0], ["-", 0, 1, 1], ["-", "-", 0, 0]] \quad (12.104)$$

Forming the coverage table
Now that we have identified all of the prime implicants, we will use the coverage dictionary to create the coverage table.

Take a look at the final state of the coverage dictionary.

```
> eval(coverageDict);
```

(12.105)

```
table([[0, 1, 0, 0] = {7}, ["-", 0, 1, 1] = {2, 8}, [0, 0, 0, 0] = {10}, [1, 0, 1, 1]
    = {2}, [0, 0, 1, 1] = {8}, [1, 0, "-", 0] = {3, 4}, ["-", 0, 0, 0] = {4, 10}, [0, 1, 1,
    1] = {5}, [0, 1, "-", 1] = {5, 6}, [1, 0, 1, 0] = {3}, ["-", "-", 0, 0] = {1, 4, 7, 10},
    [0, 0, 0, 1] = {9}, [0, "-", 0, 0] = {7, 10}, [1, "-", 0, 0] = {1, 4}, [0, 1, 0, 1]
    = {6}, [0, "-", 0, 1] = {6, 9}, [0, 1, 0, "-"] = {6, 7}, [1, 1, 0, 0] = {1}, [0, "-", 0,
    "-"] = {6, 7, 9, 10}, [1, 0, 0, 0] = {4}, [0, 0, 0, "-"] = {9, 10}, [0, 0, "-", 1] = {8,
    9}, [0, "-", "-", 1] = {5, 6, 8, 9}, [1, 0, 1, "-"] = {2, 3}, ["-", 1, 0, 0] = {1, 7}, [0,
    "-", 1, 1] = {5, 8}])
```
(12.105)

Each bit string, and in particular each prime implicant, is an index in this table. The corresponding entry is the set of integers which are the indices to the original minterms in **Fbits**. Thus, to determine which of the original minterms are covered by each prime implicant, we just look it up in the table.

We will model the coverage table as a matrix. Each row corresponds to a prime implicant, so there will be

```
> nops(allprimeImps);
```
$$6 \qquad \textbf{(12.106)}$$

rows. And each column corresponds to a minterm, so there are

```
> nops(Fbits);
```
$$10 \qquad \textbf{(12.107)}$$

columns. The entries in the matrix will be 0s and 1s with 1 in position (i, j) indicating that the prime implicant at position i in **allprimeImps** covers the minterm at position j in **Fbits**.

Recall that if the **Matrix** command is given two integers as its only arguments, it will create the matrix whose size is specified by the integers and has all 0 entries.

```
> Matrix(nops(allprimeImps),nops(Fbits));
```
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \textbf{(12.108)}$$

To enter 1s in the appropriate positions, we loop over the rows, considering each prime implicant in turn. For each prime implicant, we look up its entry in the coverage dictionary to obtain the set of minterms it covers. For each of those minterms, we place a 1 in the matrix.

The following procedure initializes the coverage table.

```
> initCoverMatrix := proc(minterms,primeImps,coverDict)
    local M, i, C, j;
    M := Matrix(nops(primeImps),nops(minterms));
    for i from 1 to nops(primeImps) do
      C := coverDict[primeImps[i]];
      for j in C do
        M[i,j] := 1;
```

```
      end do;
    end do;
    return M;
  end proc:
```

Applied to our example, this produces the following coverage table.

```
> coverageTable := initCoverMatrix(Fbits,allprimeImps,
  coverageDict);
```

$$coverageTable := \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(12.109)}$$

Manipulating the matrix

Once the coverage table is set up, we move to steps 6 and 7, determining which prime implicants to include in the minimal expression. In step 6, we identify the essential prime implicants and in step 7 we identify which of the non-essential prime implicants we will include. We will see how to identify the prime implicants to use in a moment.

To aid in performing both steps 6 and 7, we will be manipulating the coverage table. Once we have decided to include a particular prime implicant in the minimal expression, we can take three actions.

First, record the decision by adding the prime implicant to a new list, say **minBits**, the list of bit strings to be included in the minimal expression.

Second, delete that prime implicant's row from the coverage table and delete the columns corresponding to the minterms it covered. We know the prime implicant will be in the expression and thus the minterms it covers are satisfied. Hence, there is no longer any need to keep track of that information.

Third, delete the prime implicant and the minterms it covers from the lists storing them (**Fbits** and **allprimeImps**). This is to ensure that the indices of **Fbits** and **allprimeImps** continue to match the row and column numbers of the matrix.

We will now write a procedure that implements these actions. Its input will be the index to the prime implicant that has been chosen. It will also accept the names of the coverage matrix, the list of minterms, and the list of prime implicants. All of these will be modified in the procedure (refer to *Modifying arguments* above). The procedure will return the bit string of the prime implicant that was chosen.

Our procedure will be called **UpdateCT**, for "update coverage table." The **minBits** list, the list of chosen prime implicants, will be updated via the return value. This accomplishes the first task for this procedure.

Second, we must delete the row corresponding to the chosen prime implicant and the columns corresponding to the minterms covered by that implicant. Suppose, in our example, that we have decided to include the fourth prime implicant in the final result. This is

```
> allprimeImps[4];
```

$$[1, 0, \text{"-"}, 0] \tag{12.110}$$

From **coverageTable**, we need to remove row 4 (since this corresponds to the prime implicant). We also need to remove the columns corresponding to the minterms covered by this prime implicant. To determine which columns are to be removed, we find the locations of the 1s in the row of the matrix.

To determine the locations of the 1s, we'll loop over the columns checking each position in row 4 to see if it is 1 or not. We use the **ColumnDimension** command from the **LinearAlgebra** package to determine the number of columns.

```
> with(LinearAlgebra):
> covered := {};
```

$$covered := \{\} \tag{12.111}$$

```
> for i from 1 to ColumnDimension(coverageTable) do
    if coverageTable[4,i] = 1 then
      covered := covered union {i};
    end if;
  end do:
  covered;
```

$$\{3, 4\} \tag{12.112}$$

We now know that we need to remove row 4 and columns 3 and 4. To do this, we'll use a complicated selection.

We have seen that ranges can be used to select from lists. The same is true for matrices. For example, we can obtain the first three rows of this matrix as follows.

```
> coverageTable[1..3,1..-1];
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{12.113}$$

The first range indicates rows 1 through 3, the second that we want all the columns, from the first to the last.

You can also give lists of the rows instead of ranges.

```
> coverageTable[[1,2,3],[1,2,3,4,5,6,7,8,9,10]];
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{12.114}$$

In our example, we want all of the columns except for the fourth, which we can obtain from the pair of ranges **1..3** and **5..6**. More generally, if **newPI** is the index of the new prime implicant to be included in the minimal expression, we would use **1..(newPI-1)** and **(newPI+1)..-1**.

```
> row4 := 4;
```

$$row4 := 4 \tag{12.115}$$

```
> coverageTable[[1..(row4-1),(row4+1)..-1],[1..-1]];
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(12.116)}$$

For the rows we will take a different approach. Begin with the set of all column indexes.

```
> colList := {$1..ColumnDimension(coverageTable)};
```

$$colList := \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \quad \textbf{(12.117)}$$

Now remove from this set the columns that are to be removed by subtracting the **covered** set.

```
> colList := colList minus covered;
```

$$colList := \{1, 2, 5, 6, 7, 8, 9, 10\} \quad \textbf{(12.118)}$$

Then turn it into a list.

```
> colList := [op(colList)];
```

$$colList := [1, 2, 5, 6, 7, 8, 9, 10] \quad \textbf{(12.119)}$$

Using this to select the columns, we obtain the desired matrix.

```
> coverageTable[[1..(row4-1),(row4+1)..-1],colList];
```

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(12.120)}$$

The reader is encouraged to compare this to the original matrix. Note that in this example, we have not actually modified the table.

The last tasks are to remove the selected prime implicant from the list of prime implicants, and remove the covered minterms from **Fbits**. We use **subsop** to remove elements from lists, as usual. For instance, the following removes the fourth prime implicant.

```
> subsop(4=NULL,allprimeImps);
```

$$[[0, "-", 0, "-"], [0, "-", "-", 1], [1, 0, 1, "-"], ["-", 0, 1, 1], ["-", "-", 0, 0]] \quad \textbf{(12.121)}$$

When removing the minterms, we remove them in reverse order. For instance, to remove the minterms in locations 3 and 4, we first remove the minterm in position 4, and then the minterm in position 3. Otherwise, if we remove the minterm in location 3 first, then all the other minterms shift location by 1. That is, the minterm previously in location 4 is now in location 3. By removing them in reverse order, this is not a concern.

Here is the procedure. Remember that since we're using the **evaln** parameter option in order to make parameters modifiable, we must use **eval** when we need the values of those objects.

```
> UpdateCT := proc(newPI,coverTable::evaln,minterms::evaln,

  primeImps::evaln)
    local newPIbits, numcols, covered, i, colList;
```

```
    newPIbits := eval(primeImps)[newPI];
    numcols:= LinearAlgebra[ColumnDimension](eval(coverTable));
    covered := {};
    for i from 1 to numcols do
      if eval(coverTable)[newPI,i] = 1 then
        covered := covered union {i};
      end if;
    end do;
    colList := [op({$1..numcols} minus covered)];
    coverTable :=
        eval(coverTable)[[1..(newPI-1),(newPI+1)..-1],colList];
    primeImps := subsop(newPI=NULL,eval(primeImps));
    for i from nops(covered) to 1 by -1 do
      minterms := subsop(covered[i]=NULL,eval(minterms));
    end do;
    return newPIbits;
  end proc:
```

Finding essential prime implicants

Next we write a procedure to identify the essential prime implicants. Recall that a prime implicant is essential when it is the only prime implicant to cover some minterm. In terms of the coverage table, this is equivalent to the existence of a column with only one 1.

We will locate the essential prime implicants as follows. First, we initialize the set of essential prime implicants to the empty list.

We proceed in a manner similar to the **MergeBitstrings** procedure. We use a for loop to step through the columns of the coverage table. Within this loop, we initialize a name, **rowhas1**, to 0.

We then enter a second for loop to step through the entries in the columns. When a 1 entry has been found, we check **rowhas1**. If that name is 0, then it is assigned to the current row number. If it is not 0, then we have found a second 1 in the column and we assign **rowhas1** to -1 and use **break** to terminate the inner loop. After the inner loop, we test **rowhas1**. If it is positive, then we know that only one 1 was located in that column, and hence the row the solitary 1 was found in corresponds to an essential prime implicant. In this case, we add the row number (**rowhas1**) to **essentials**.

The following procedure implements this algorithm and returns the list of essential prime implicants.

```
> FindEssentials := proc(coverTable)
   local essentials, i, j, rowhas1;
   essentials := {};
   for i from 1 to LinearAlgebra[ColumnDimension](coverTable)
  do
     rowhas1 := 0;
     for j from 1 to LinearAlgebra[RowDimension](coverTable) do
       if coverTable[j,i] = 1 then
         if rowhas1 = 0 then
           rowhas1 := j;
         else
           rowhas1 := -1;
           break;
         end if;
       end if;
     end do;
```

```
      if rowhas1 > 0 then
        essentials := essentials union {rowhas1};
      end if;
    end do;
    return essentials;
  end proc:
```

We use this to determine the essential prime implicants of our example.

```
> essentialPIs := FindEssentials(coverageTable);
```

$$essentialPIs := \{2, 6\} \tag{12.122}$$

Now that we have the essential prime implicants, we can initialize **minBits** and apply **UpdateCT** to the essential prime implicants. Once again, we loop through the list backwards.

```
> minBits := [];
```

$$minBits := [\,] \tag{12.123}$$

```
> for i from nops(essentialPIs) to 1 by -1 do
    minBits := [op(minBits),
    UpdateCT(essentialPIs[i],coverageTable,Fbits,allprimeImps)];
  end do:
> minBits;
```

$$[["-", "-", 0, 0], [0, "-", "-", 1]] \tag{12.124}$$

```
> coverageTable;
```

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{12.125}$$

Completing the coverage

Provided that the essential prime implicants did not completely cover the original minterms, we must complete the coverage with non-essential prime implicants. First, we ensure that the coverage is not complete by checking the column dimension.

```
> evalb(LinearAlgebra[ColumnDimension](coverageTable)>0);
```

$$true \tag{12.126}$$

As we mentioned earlier, we will use a heuristic approach to find a minimal set of prime implicants rather than using an exhaustive search to determine the minimum. The heuristic we use will be to choose the prime implicant with the most extensive coverage of the remaining minterms.

To find such a prime implicant, we will do the following. First, initialize **maxCoverage** and **bestImp** both to 0. Then loop over each row of the (modified) coverage table. For each row, we will compute the sum of the entries. If this sum is greater than **maxCoverage**, then set **maxCoverage** to the sum and set **bestImp** to the row number. Once the loop is complete, **bestImp** will be the index to a row with maximum coverage and will be the next prime implicant added to the **minBits** list.

Here is the procedure that implements this strategy.

```
> findBestImp := proc(coverTable)
    local maxCoverage, bestImp, i, j, sum;
```

```
    maxCoverage := 0;
    bestImp := 0;
    for i from 1 to LinearAlgebra[RowDimension](coverTable) do
     sum := 0;
     for j from 1 to LinearAlgebra[ColumnDimension](coverTable)
    do
     sum := sum + coverTable[i,j];
    end do;
    if sum > maxCoverage then
     maxCoverage := sum;
     bestImp := i;
    end if;
    end do;
    return bestImp;
  end proc:
```

As long as the coverage table is not empty, we apply this procedure to it to obtain the next implicant. We add the implicant to the list **minBits** representing the minimal expression and update the coverage table using **updateCT**.

```
> while LinearAlgebra[ColumnDimension](coverageTable) > 0 do
    nextPI := findBestImp(coverageTable);
    minBits := [op(minBits),
      UpdateCT(nextPI,coverageTable,Fbits,allprimeImps)];
  end do;
```

$$nextPI := 2$$

$$minBits := [["-", "-", 0, 0], [0, "-", "-", 1], [1, 0, 1, "-"]] \quad \textbf{(12.127)}$$

All that's left is to translate **minBits** back into a logical expression. This can be done using **BitStringtoMT** applied to each element of **minBits** with the map command and then combined into one expression with **OrList**.

```
> OrList(map(BitStringtoMT,minBits,[w,x,y,z]));
```

$$(\&not(y)\ \&and\ \&not(z)\ \&or\ \&not(w)\ \&and\ z)\ \&or\ (w\ \&and\ \&not(x))\ \&and\ y \quad \textbf{(12.128)}$$

Putting it all together

Finally, we assemble the pieces into a single procedure, which accepts a logical expression in disjunctive normal form and a list of its variables. It returns a minimal equivalent expression.

```
> QuineMcCluskey := proc(F,variables)
    local Fbits, FbitsL, coverageDict, groups, primes, i,
  allprimeImps, j, coverageTable, essentialPIs, minBits,
  nextPI;
    uses LinearAlgebra;
    Fbits := DNFtoBitList(F,variables);
    coverageDict := initCoverDict(Fbits);
    i := 0;
    FbitsL[0] := Fbits;
    while FbitsL[i] <> [] do
      i := i + 1;
      groups[i] := sortGroups(FbitsL[i-1]);
      FbitsL[i],primes[i]:=NextBitList(groups[i],coverageDict);
    end do;
    allprimeImps := {};
    for j from 1 to i do
      allprimeImps := allprimeImps union primes[j];
```

```
    end do;
    allprimeImps := [op(allprimeImps)];
    coverageTable :=
              initCoverMatrix(Fbits,allprimeImps,coverageDict);
    essentialPIs := FindEssentials(coverageTable);
    minBits := [];
    for i from nops(essentialPIs) to 1 by -1 do
      minBits := [op(minBits),
      UpdateCT(essentialPIs[i],coverageTable,Fbits,
  allprimeImps)];
    end do;
    while ColumnDimension(coverageTable) > 0 do
      nextPI := findBestImp(coverageTable);
      minBits := [op(minBits),
        UpdateCT(nextPI,coverageTable,Fbits,allprimeImps)];
    end do;
    return OrList(map(BitStringtoMT,minBits,variables));
  end proc:
```

Define **Ex10** to be the expression in Example 10 from Section 12.4 of the text.

```
> E10 := (w &and x &and y &and &not(z)) &or
         (w &and &not(x) &and y &and z) &or
         (w &and &not(x) &and y &and &not(z)) &or
         (&not(w) &and x &and y &and z) &or
         (&not(w) &and x &and &not(y) &and z) &or
         (&not(w) &and &not(x) &and y &and z) &or
         (&not(w) &and &not(x) &and &not(y) &and z);
```

$$E10 := (((((((w \,\&and\, x) \,\&and\, y) \,\&and\, \&not(z)) \,\&or\, ((w \,\&and\, \&not(x)) \,\&and\, y) \,\&and\, z) \,\&or\, ((w \,\&and\, \&not(x)) \,\&and\, y) \,\&and\, \&not(z)) \,\&or\, ((\&not(w) \,\&and\, x) \,\&and\, y) \,\&and\, z) \,\&or\, ((\&not(w) \,\&and\, x) \,\&and\, \&not(y)) \,\&and\, z) \,\&or\, ((\&not(w) \,\&and\, \&not(x)) \,\&and\, y) \,\&and\, z) \,\&or\, ((\&not(w) \,\&and\, \&not(x)) \,\&and\, \&not(y)) \,\&and\, z \quad \textbf{(12.129)}$$

```
> QuineMcCluskey(E10,[w,x,y,z]);
```

$$((w \,\&and\, y) \,\&and\, \&not(z) \,\&or\, \&not(w) \,\&and\, z) \,\&or\, (w \,\&and\, \&not(x)) \,\&and\, y \quad \textbf{(12.130)}$$

Note that this is the first of the two answers given in the solution to Example 10.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 2

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

Solution: The Boolean functions of degree three are in one-to-one correspondence with the subsets of $\{true, false\}^3$. This is because each subset S of $\{true, false\}^3$ can be identified with the unique Boolean function of degree three which returns true on the members of S and false on all other inputs.

Thus, we begin by constructing the set $\{true, false\}^3$ and its power set.

To construct $\{true, false\}^3$, we will use the **cartprod** command from the **combinat** package. (Refer to Section 2.1 of this manual for information on the cartprod command.) Like many of the commands for generating combinatorial objects, **cartprod** produces a list with indices **finished** and **nextvalue**. We use it to form the set **TF3**.

```
> TF3 := {};
```

$$TF3 := \{\} \tag{12.131}$$

```
> TF3iterator := combinat[cartprod]([[true,false]$3]):
> while not TF3iterator[finished] do
    TF3 := TF3 union {TF3iterator[nextvalue]()};
  end do:
> TF3;
```

$$\{[false, false, false], [false, false, true], [false, true, false], [false, true, true], [true, false, false], [true, false, true], [true, true, false], [true, true, true]\} \tag{12.132}$$

To produce the subset of $\{true, false\}^3$, we use the **subsets** command from **combinat**. This command also produces a table with indices **finished** and **nextvalue**.

```
> TF3subsets := combinat[subsets](TF3):
```

Now we will create a list of all of the Boolean functions. The subsets that are produced by **TF3subsets** are each valid inputs to the **BooleanFromTable** procedure we wrote in Section 12.2. We also apply **BooleanSimplify**, in order to have simpler representations, before adding the expression to the list.

```
> allFunctions := [];
```

$$allFunctions := [\,] \tag{12.133}$$

```
> while not TF3subsets[finished] do
    nextTF3subset := TF3subsets[nextvalue]();
    nextTF3function:=BooleanFromTable(nextTF3subset,[x,y,z]);
    nextTF3function := BooleanSimplify(nextTF3function);
    allFunctions := [op(allFunctions),nextTF3function];
  end do:
```

The **allFunctions** list is lengthy, so we display only a few members.

```
> nops(allFunctions);
```

$$256 \tag{12.134}$$

```
> allFunctions[1..10];
```

$$[false, (\&not(x)\ \&and\ \&not(y))\ \&and\ \&not(z), (z\ \&and\ \&not(x))\ \&and\ \&not(y), (y\ \&and\ \&not(x))\ \&and\ \&not(z), (y\ \&and\ z)\ \&and\ \&not(x), (x\ \&and\ \&not(y))\ \&and\ \&not(z), (x\ \&and\ z)\ \&and\ \&not(y), (x\ \&and\ y)\ \&and\ \&not(z), (x\ \&and\ y)\ \&and\ z, \&not(x)\ \&and\ \&not(y)] \tag{12.135}$$

To obtain the values of the functions, we will use the **TruthTable** command. Recall from the first section that **TruthTable** requires two arguments, a Boolean expression and a list of variables that appear in the expression. The result is a table with indices lists of truth values and the corresponding entries are the value of the expression at the index.

We form the list of the truth tables associated to each function in the **allFunctions** list with the **map** command. Note that we use the third argument **[x,y,z]** in **map** so that this list of

variables is passed to **TruthTable** each time it is applied to an expression from **allFunctions**.

```
> allTables := map(TruthTable,allFunctions,[x,y,z]):
```

Each entry in **allTables** is a truth table. To obtain the value of the 136th function on (*true*, *false*, *true*), you would enter

```
> allTables[136][true,false,true];
```

$$true \tag{12.136}$$

To display the entire truth table for a particular function, loop through the members of **TF3**. We must apply **op** to the member of **TF3**, since the index to a truth table is a sequence not a list.

```
> for i from 1 to nops(TF3) do
    print(TF3[i],allTables[136][op(TF3[i])]);
  end do;
```

$$[false, false, false], false$$
$$[false, false, true], true$$
$$[false, true, false], true$$
$$[false, true, true], false$$
$$[true, false, false], false$$
$$[true, false, true], true$$
$$[true, true, false], true$$
$$[true, true, true], false \tag{12.137}$$

Computations and Explorations 6

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

Solution: To solve this problem, we need to find a way to generate random boolean expressions, and then we must find a method of examining the minimization process so that we can count the number of steps.

In the **Logic** package, the command **Random** produces a random Boolean expression. The only required argument is a set or list specifying the symbols to be used. For example, to produce a random Boolean expression on the symbols *w*, *x*, *y*, and *z*, you enter the following.

```
> Random([w,x,y,z]);
```

$$\begin{aligned}&((((((w \,\&and\, x) \,\&and\, y) \,\&and\, \&not(z)) \,\&or\, ((w \,\&and\, x) \,\&and\, z) \\ &\quad \&and\, \&not(y)) \,\&or\, ((w \,\&and\, x) \,\&and\, \&not(y)) \,\&and\, \&not(z)) \\ &\quad \&or\, ((w \,\&and\, y) \,\&and\, \&not(x)) \,\&and\, \&not(z)) \\ &\quad \&or\, ((w \,\&and\, z) \,\&and\, \&not(x)) \,\&and\, \&not(y)) \\ &\quad \&or\, ((x \,\&and\, y) \,\&and\, z) \,\&and\, \&not(w)) \\ &\quad \&or\, ((x \,\&and\, y) \,\&and\, \&not(w)) \,\&and\, \&not(z)) \\ &\quad \&or\, ((x \,\&and\, z) \,\&and\, \&not(w)) \,\&and\, \&not(y)\end{aligned} \tag{12.138}$$

The **Random** command also accepts a second optional argument: **form=CNF**, **form=DNF**, or **form=MOD2**, specifying the form of the expression produced. The default form is disjunctive

normal form.

Having determined how to generate random expressions, we need to find a way to count the number of steps taken during the minimization process. There are (at least) three approaches we could take to this part of the problem.

The first is to measure the time taken to execute the procedure. We have done this many times before.

```
> QMtime := []:
  for i from 1 to 10 do
    randExp := Random([w,x,y,z]);
    st := time();
    QuineMcCluskey(randExp,[w,x,y,z]);
    et := time() - st;
    QMtime := [op(QMtime),et];
  end do:
  Statistics[Mean](QMtime);
```

$$0.001200000000 \qquad \textbf{(12.139)}$$

The second approach is to modify the procedure to count the number of times certain operations are called. For example, we may be interested in the number of times that the **UpdateCT** procedure is executed. In this case, we can alter **UpdateCT** to include a global variable that is incremented at the start of every execution.

```
> UpdateCT := proc(newPI,coverTable::evaln,minterms::evaln,

  primeImps::evaln)
    local newPIbits, numcols, covered, i, colList;
    global countUpdateCT;
    countUpdateCT := countUpdateCT + 1;
    newPIbits := eval(primeImps)[newPI];
    numcols := LinearAlgebra[ColumnDimension](eval
  (coverTable));
    covered := {};
    for i from 1 to numcols do
      if eval(coverTable)[newPI,i] = 1 then
        covered := covered union {i};
      end if;
    end do;
    colList := [op({$1..numcols} minus covered)];
    coverTable :=
        eval(coverTable)[[1..(newPI-1),(newPI+1)..-1],
  colList];
    primeImps := subsop(newPI=NULL,eval(primeImps));
    for i from nops(covered) to 1 by -1 do
      minterms := subsop(covered[i]=NULL,eval(minterms));
    end do;
    return newPIbits;
  end proc:
```

We must initialize the variable to 0.

```
> countUpdateCT := 0;
```

$$countUpdateCT := 0 \qquad \textbf{(12.140)}$$

Now execute **QuineMcCluskey** on 10 random expressions.

```
> for i from 1 to 10 do
    randExp := Random([w,x,y,z]);
    QuineMcCluskey(randExp,[w,x,y,z]);
  end do:
```

The **countUpdateCT** variable will now store the number of times **UpdateCT** was called. Dividing by 10 gives us the average.

```
> countUpdateCT/10.;
```

$$4.500000000 \qquad \textbf{(12.141)}$$

The third approach is to make use of Maple's debugging facilities. We used **trace** in earlier chapters to get information about the workings of a procedure. Here, we will use the **showstat** command.

If you apply **showstat** to the name of a procedure, it will display the definition of the procedure with line numbers added on the left hand side. An integer or a range of integers can be given as a second argument to narrow the display to the desired lines.

```
> showstat(QuineMcCluskey,5..8);

QuineMcCluskey := proc(F, variables)
local Fbits, FbitsL, coverageDict, groups, primes, i,
allprimeImps, j, coverageTable, essentialPIs, minBits,
nextPI;
      ...
   5  while FbitsL[i] <> [] do
   6    i := i+1;
   7    groups[i] := sortGroups(FbitsL[i-1]);
   8    FbitsL[i], primes[i] := NextBitList(groups[i],
coverageDict)
      end do;
      ...
end proc
```

You can get more information by telling Maple to track the procedure. You do this by calling the **debugopts** command with argument **traceproc=** and then the name of the procedure.

```
> debugopts(traceproc=QuineMcCluskey);
```

Now we apply **QuineMcCluskey** to 10 random expressions. We suppress the output as it is not needed here.

```
> for i from 1 to 10 do
    randExp := Random([w,x,y,z]);
    QuineMcCluskey(randExp,[w,x,y,z]);
  end do:
```

If we call the **showstat** command again, the output gives us more information than it did before.

```
> showstat(QuineMcCluskey);

QuineMcCluskey := proc(F, variables)
local Fbits, FbitsL, coverageDict, groups, primes, i,
allprimeImps, j, coverageTable, essentialPIs, minBits,
nextPI;
     |Calls Seconds  Words|
PROC |   10   0.012  94598|
```

```
   1 |   10   0.000  18749| Fbits := DNFtoBitList(F,
variables);
   2 |   10   0.000   4018| coverageDict := initCoverDict
(Fbits);
   3 |   10   0.000      0| i := 0;
   4 |   10   0.000   2730| FbitsL[0] := Fbits;
   5 |   10   0.000    215| while FbitsL[i] <> [] do
   6 |   25   0.000      0|   i := i+1;
   7 |   25   0.003  13148|   groups[i] := sortGroups
(FbitsL[i-1]);
   8 |   25   0.003  22728|   FbitsL[i], primes[i] :=
NextBitList(groups[i],coverageDict)
                           end do;
   9 |   10   0.000      0| allprimeImps := {};
  10 |   10   0.000      0| for j to i do
  11 |   25   0.000    400|   allprimeImps := `union`
(allprimeImps,primes[j])
                           end do;
  12 |   10   0.000     40| allprimeImps := [op
(allprimeImps)];
  13 |   10   0.001   5476| coverageTable :=
initCoverMatrix(Fbits,allprimeImps,coverageDict);
  14 |   10   0.001   4120| essentialPIs := FindEssentials
(coverageTable);
  15 |   10   0.000      0| minBits := [];
  16 |   10   0.000      0| for i from nops(essentialPIs)
by -1 to 1 do
  17 |   29   0.003   8472|   minBits := [op(minBits),
UpdateCT(essentialPIs[i],coverageTable,Fbits,allprimeImps)]
                           end do;
  18 |   10   0.000    436| while 0 < LinearAlgebra:-
ColumnDimension(coverageTable) do
  19 |   12   0.000   1792|   nextPI := findBestImp
(coverageTable);
  20 |   12   0.000   2969|   minBits := [op(minBits),
UpdateCT(nextPI,coverageTable,Fbits,allprimeImps)]
                           end do;
  21 |   10   0.001   9305| return OrList(map
(BitStringtoMT,minBits,variables))
end proc
```

In addition to the line numbers, you see three columns labeled Calls, Seconds, and Words. Also note that above line 1 is line PROC. This refers to information for the procedure as a whole.

The Calls column reports the number of times the line was executed. That the Calls column contains 10 in the PROC row indicates that the procedure was called 10 times. Note that lines 17 and 20 are the lines containing the calls to **UpdateCT**. The sum of the values in the Count column is the number of times **UpdateCT** was called.

The Seconds column reports the amount of CPU time that was spent executing the line.

The Words column indicates the amount of memory that was allocated as a result of the statement.

Together, the three columns give you a considerable amount of information about the

computational complexity, performance, and memory requirements of a procedure. In **QuineMcCluskey**, we see that the most time and memory are used in line 8, the call to **NextBitList**. However, the **UpdateCT** subprocedure was executed more often.

You can also have Maple store this information in a table for you by calling debugopts with argument **traceproctable=** and the name of the procedure.

```
> traceTable := debugopts(traceproctable=QuineMcCluskey);
```

$$traceTable := \begin{bmatrix} 1..22 \times 1..3\ Array \\ Data\ Type:\ integer_4 \\ Storage:\ rectangular \\ Order:\ C\_order \end{bmatrix} \tag{12.142}$$

The entries in this table correspond to the information displayed. The first row stores information about the procedure as a whole.

```
> traceTable[1,1..3];
```

$$\begin{bmatrix} 10 & 12 & 94598 \end{bmatrix} \tag{12.143}$$

Information on specific lines is stored in the row one greater than the line number. For instance, the data related to the calls of **UpdateCT**, in line 17 and 20, are contained in the table at 18 and 21.

```
> traceTable[[18,21],1..3];
```

$$\begin{bmatrix} 29 & 3 & 8472 \\ 12 & 0 & 2969 \end{bmatrix} \tag{12.144}$$

Note that the second column, containing the time measurement, has been multiplied by 1000.

Executing

```
> debugopts(traceproc=QuineMcCluskey);
```

a secont time clears the information that was stored and toggles the option to have Maple record the Calls, Seconds, and Words information back off.

Exercises

Exercise 1. Use Maple to verify De Morgan's Laws and the commutative and associative laws. (See Table 5 of Section 12.1.)

Exercise 2. Construct truth tables for each of the following pairs of boolean expressions and decide whether they are logically equivalent.

a) $a \to \overline{b}$ and $b \to a$

b) $a \to b$ and $b \to \overline{a}$

c) $a + bc$ and $(a + b + d)(a + c + d)$.

Exercise 3. Write a Maple procedure that constructs a table of values of a boolean expressions in n variables that may include the following operators: **&and**, **&or**, **&xor**, **&nand**, **&nor**.

Exercise 4. Write a Maple procedure that, given a boolean function, represents this function using only the **&nand** operator.

Exercise 5. Use the procedure in the previous exercise to represent the following boolean functions

using only the `&nand` operator.
a) $F(x, y, z) = xy + \overline{yz}$
b) $G(x, y, z) = x + \overline{xy} + \overline{yz}$
c) $H(x, y, z) = xyz + \overline{xyz}$

Exercise 6. Write a Maple procedure that, given a boolean function, represents this function using the `&nor` operator.

Exercise 7. Use the procedure in the previous exercise to represent the boolean functions in Exercise 5 using only the `&nor` operator.

Exercise 8. Write a Maple procedure for determining the output of a threshold gate, given the values of n boolean variables as input, and given the threshold value and a set of weights for the threshold gate. (See the Supplementary Exercises of Chapter 12 for information on threshold gates.)

Exercise 9. Develop a Maple procedure that, given a boolean function in four variables, determines whether it is a threshold function, and if so, finds the appropriate threshold gate representing this function. (See the Supplementary Exercises of Chapter 12.)

Exercise 10. A boolean expression e is called *self dual* if it is logically equivalent to its dual e^d. Write a Maple procedure to test whether a given expression is self dual.

Exercise 11. Determine, for each integer $n \in \{1, 2, 3, 4, 5, 6\}$, the total number of boolean functions of n variables and the number of those functions that are self dual.

Exercise 12. Write a Maple procedure that, given a positive integer n, constructs a list of all boolean functions of degree n. Use your procedure to find all boolean functions of degree 4.

Exercise 13. At the end of Section 12.3 of this manual, it was suggested that the procedure for producing tree representations of logical circuits could be improved by combining successive **and** or **or** gates into gates accepting multiple inputs. Implement this.

Exercise 14. Use `DontCare` to compute a minimal sum of products expansion for the boolean functions with don't care conditions specified by the Karnaugh maps shown in Exercises 30 through 32 of Section 12.4.

Exercise 15. How can you change exactly one character in the definition of the procedure `DontCare` so that it returns the last expression of minimum length that it encounters that is equivalent to the input function? (As written now, it returns the first.)

Exercise 16. Use the procedure you wrote in Exercise 10 to write a Maple procedure to generate random boolean expressions in 4 variables and stop when it is has found one that is self dual. Run the program several times and time it. Find the average time. Repeat for boolean expressions in 5 and 6 variables. Can you make any conjectures from this information?

Exercise 17. Revise the procedure `DontCare` to return all minimal expressions that it finds, rather than just the first.

Exercise 18. Revise the procedure `DontCare` to use different measures of complexity of boolean expressions, such as the number of boolean operations, etc.

Exercise 19. Modify `QuineMcCluskey` to allow for don't care conditions. See the discussion at the end of the Don't Care Conditions subsection in Section 12.4 of this manual.

Exercise 20. Modify `QuineMcCluskey` to use backtracking instead of the heuristic approach in order to determine the expression with the minimum number of terms. Use a large number of randomly generated expressions to compare the old procedure with the new and determine how often the heuristic produces non-optimal output.