1 The Foundations: Logic and Proofs

Introduction

This chapter describes how Maple can be used to further your understanding of logic and proofs. In particular, we describe how to construct truth tables, check the validity of logical arguments, and verify logical equivalence. In the final two sections, we provide examples of how Maple can be used as part of proofs, specifically to find counterexamples, carry out proofs by exhaustion, and to search for witnesses for existence proofs.

1.1 Propositional Logic

In this section we will discuss how to use Maple to explore propositional logic. Specifically, we will see how to use logical connectives in Maple, describe the connection between logical implication and conditional statements in a program, show how Maple can be used to create truth tables for compound propositions, and demonstrate how Maple can be used to carry out bit operations.

In Maple, the truth values true and false are represented by the names **true** and **false**. These are constants in Maple, just like a numeric constant such as π. Propositions can be represented by names (variables) such as **p**, **q** or **Prop1**. Note that if you have not yet made an assignment to a name, entering it will return the name.

```
> Prop1;
```

$$Prop1 \tag{1.1}$$

Once you've assigned a value, Maple will evaluate the name to the assigned value whenever it appears.

```
> Prop1 := true;
```

$$Prop1 := true \tag{1.2}$$

```
> Prop1;
```

$$true \tag{1.3}$$

You can cause Maple to "forget" the assigned value with the following syntax:

```
> Prop1 := 'Prop1';
```

$$Prop1 := Prop1 \tag{1.4}$$

```
> Prop1;
```

$$Prop1 \tag{1.5}$$

Right single quotes are used in Maple to delay evaluation of an expression. In the above, they are used to assign **Prop1** to the unevaluated **Prop1**, in other words, we set the value of the variable to the name of the variable which effectively unassigns the name. (Note that the right single quote is on the same key as the quotation mark on a standard keyboard. In Maple, the right single quote is different from the left single quote, which is on the same key as the tilde, ~.)

Logical Connectives

Maple supports all of the basic logical operators discussed in the textbook except for the biconditional. We illustrate the logical operators of negation (**not**), conjunction (**and**), disjunction (**or**), exclusive or (**xor**), and implication (**implies**). Note that expressions formed using the logical connectives are called boolean expressions, and variables that stand for true or false are called boolean variables.

```
> not true;
```

$$false \tag{1.6}$$

```
> true and false;
```

$$false \tag{1.7}$$

```
> true or false;
```

$$true \tag{1.8}$$

```
> true xor false;
```

$$true \tag{1.9}$$

```
> true implies false;
```

$$false \tag{1.10}$$

```
> (true and false) implies true;
```

$$true \tag{1.11}$$

Note that the precedence of the Maple logical operators agrees with Table 8 of Section 1.1, so the parentheses in the previous command is unnecessary, though they make for more readable propositions.

Maple does not support the biconditional directly, but it can be easily programmed to do so.

```
> IFF := proc(a,b)
    (a implies b) and (b implies a);
  end proc;
```

$$IFF := \mathbf{proc}(a, b)\ (a\ \mathbf{implies}\ b)\ \mathbf{and}\ (b\ \mathbf{implies}\ a)\ \mathbf{end\ proc} \tag{1.12}$$

Because this is our first Maple procedure, let's spend a moment breaking it down. First we have the name, **IFF**, followed by the assignment operator **:=**. Then we use the keyword **proc**, short for procedure. After the keyword **proc**, we list, inside parentheses, names for the inputs to the procedure.

After closing the parentheses to terminate the list of arguments, we begin the body of the procedure, called the statement sequence. The statement sequence can be as long as is required. In this case, only one statement was required. Note that the result of the procedure is the value of the last statement that is executed.

Finally, we end the procedure with the statement **end proc;**. As is usual for assignment statements, Maple's output is a repetition of the assignment. For this reason, we will generally use a colon rather than a semicolon to terminate the **end proc:** statement. This suppresses the output of the assignment command but doesn't alter the procedure.

To execute a procedure that we've written, give the name of the procedure followed by arguments, in parentheses.

```
> IFF(true,true);
```

$$true \tag{1.13}$$

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

$$false \tag{1.14}$$

Conditional Statements

We saw above that Maple includes the operator **implies** for evaluating logical implication or conditional statements. In mathematical logic, "if p, then q" has a very specific meaning, as described in detail in the text. In computer programming, and Maple in particular, conditional statements also appear very frequently, but have a slightly different meaning.

From the perspective of logic, a conditional statement is, like any other proposition, a sentence that is either true or false. In most computer programming languages, when we talk about a conditional statement, we are not referring to a kind of proposition. Rather, conditional statements are used to selectively execute portions of code. Consider the following example of a procedure, which adds 1 to the input value if the input is less than or equal to 5 and not otherwise.

```
> IfProc1 := proc(x)
    local y;
    y := x;
    if y <= 5 then
      y := y + 1;
    else
      y := y - 1;
    end if;
    return y;
  end proc:
```

Let's see that this procedure works as promised:

```
> IfProc1(3);
```

$$4 \qquad (1.15)$$

```
> IfProc1(7);
```

$$6 \qquad (1.16)$$

There's quite a lot in this procedure. First, we use the keyword **local** followed by the name **y**. This tells Maple that **y** is a name that is used only in the procedure, so that if the name **y** is being used someplace else in the worksheet, they won't interfere with each other. In particular, if you're using **y** as a name for some value and then you run this procedure, having declared **y** as local to the procedure guarantees that the **y** you assigned outside of the procedure isn't changed. You should always declare variables you use in a procedure as local and this declaration needs to be the first statement after the parameter list. If you have more than one variable to declare as local, you list them separated by commas and end the entire list with a semicolon. If you write a procedure and use variables that are not declared local, Maple will warn you about them. Also note that the names you give parameters to are implicitly local.

After we declare **y** to be local in our procedure, we assign **y** equal to the value of **x**. It may seem strange to "copy" the value of **x** to **y** and then work with **y** rather than just using **x**. The reason is that Maple will not let you assign to a parameter. If you try including **x := x + 1** in the procedure (go ahead and try), Maple will raise an error when you try to apply the procedure to an input value. There is a very good reason for this: it prevents you from accidentally modifying an input. For example, suppose you had a name **important** and stored some value in that variable, perhaps a value obtained after considerable work. If you then wrote a procedure that accidentally modifies its input parameter and you ran your procedure on **important**, then the value would be lost and you'd have to redo all your work. So Maple prevents you from assigning to parameters in the body of a procedure in order to help you avoid making that kind of mistake. (There is a way around this, but we won't discuss it now.)

Next are the five lines comprising the conditional statement: **if y <= 5 then y := y + 1; else y := y -1; end if;**. Again, this is not a proposition, it is a command. When Maple comes to this part of the procedure, it sees the keyword **if** and knows that what follows the **if**, and up to the **then** keyword, is a conditional expression (which is a proposition). Maple checks to see if the conditional expression is true or not. If the conditional expression is true, then Maple knows to execute the statement(s) after the **then** keyword. If the conditional expression is false,

then Maple executes the statement(s) after the **else** keyword.

Note also the way in which we change the value of y: **y := y + 1;**. This is another instance of Maple trying to make sure we "really mean it" when we want to change a value. The command **y + 1;** would not change the value of **y**. To change the value of y we have to add 1 to **y**, and then assign that number to the name **y**.

Finally, the last line of the procedure is **return y**. We mentioned in the discussion of the **IFF** procedure, that a Maple procedure's output is the result of the last statement that is executed. Imagine the **IfProc1** procedure without the **return y;** line. In that case, we wouldn't be able to tell in advance what the last command to be executed is. For small values, the last command would be the assignment **y := y + 1;**, but for inputs greater than 5, **y := x;** would be the last statement executed. It is better programming style to make it clear what the output of the procedure will be, since (especially in more complicated programs) it can get quite difficult to tell what the output will be. In this example, we made the output explicit with the **return y;** command. Since this follows the end of the conditional statement, we know it will always be the last statement executed.

The **return** keyword is used in this case to be very explicit that the output from the procedure is the value of **y**. The **return** command, however, is very useful as it can allow short-circuiting of procedures. That means that execution of the procedure immediately stops and the value following the **return** keyword is output by the procedure.

Truth Tables and Loops

In the textbook, you saw how to construct truth tables by hand. Here we'll see how to have Maple create them for us. We'll begin by considering the simplest case of a compound proposition: the negation of a single propositional variable.

```
> Prop2 := not p;
```

$$Prop2 := \textbf{not}\ p \qquad \textbf{(1.17)}$$

Note that we've defined the proposition **Prop2** as an expression in terms of the name **p**, which has not been assigned a variable. We can determine the truth value of **Prop2** in one of two ways. The obvious way is to assign a truth value to **p** and then ask Maple for the value of **Prop2** as follows.

```
> p := false;
```

$$p := false \qquad \textbf{(1.18)}$$

```
> Prop2;
```

$$true \qquad \textbf{(1.19)}$$

The drawback of this approach, however, is that our variable p is now identified with false and if we want to use it as a name again, we need to manually unassign it.

```
> p := 'p';
```

$$p := p \qquad \textbf{(1.20)}$$

The other approach is to use the **eval** command. **eval**, short for evaluate, has several different uses, but the most common is for evaluating an expression (including a boolean expression) at given values for the variables in the expression. For example.

```
> eval(Prop2,p=true);
```

$$false \qquad \textbf{(1.21)}$$

In this usage, the command takes two arguments. The first argument is the expression to be evaluated. The second argument specifies the values that are to be substituted for the variables in

the expression. If there is only one variable to be replaced, as in the **Prop2** example, the second argument can be given as the single equation **variable=value**. If there are multiple variables, you must provide a list of such equations as the second argument, as in the next example.

```
> eval(p and (not q),[p=true,q=false]);
```

$$true \tag{1.22}$$

To make a truth table for a proposition, we need to evaluate the proposition at all possible truth values of all of the different variables. To do this, we make use of loops (refer to the Introduction for a general discussion of loops in Maple). Specifically, we want to loop over the two possible truth values, true and false, so we will construct a for loop over the list **[true,false]**.

```
> for loopP in [true,false] do
    print(loopP,eval(Prop2,p=loopP));
  end do;
```

$$true, false$$
$$false, true \tag{1.23}$$

The **for** keyword identifies this as a for loop for Maple. The name following the **for** keyword, **loopP**, is used as the index variable in the loop. The **in [true,false]** clause tells Maple that we want the index variable to be sequentially assigned to the elements in the list **[true,false]**. And the **do** keyword indicates the beginning of the body of the loop.

The body of the loop consists of a single statement involving two commands. The **print** command can take any number of arguments and causes Maple to display the arguments to the command. The first argument is the index variable **loopP**, meaning that the first thing displayed in each interation of the loop is the value of the boolean variable.

The second argument to **print** is a call to the **eval** command: **eval(Prop2,p=loopP)**. In evaluating this command, Maple first evaluates the names **Prop2** and **loopP**, replacing them with their values. So **Prop2** is replaced with **not p** and **loopP** is replaced with whichever truth value it is assigned to in the iteration of the loop. Then, the **eval** command causes Maple to replace the variable **p** in **Prop2** with the current value of **loopP**. Finally, Maple evaluates the boolean expression so that the **print** command causes the truth value to be displayed.

When there are multiple propositional variables, we use multiple for loops. This is called "nesting" the loops. For example,

```
> Prop3 := (p and q) implies p;
```

$$Prop3 := p \text{ and } q \Rightarrow p \tag{1.24}$$

```
> for loopP in [true,false] do
    for loopQ in [true,false] do
      print(loopP,loopQ,eval(Prop3,[p=loopP,q=loopQ]));
    end do;
  end do;
```

$$true, true, true$$
$$true, false, true$$
$$false, true, true$$
$$false, false, true \tag{1.25}$$

Note that the output indicates that the proposition, $(p \wedge q) \rightarrow p$, is a tautology. In fact, this is a rule of inference called simplification, discussed in Section 1.6 of the textbook.

Logic and Bit Operations

We can also use Maple to explore the bit operations of OR, AND, and XOR. Recall that bit operations correspond to logical operators by equating 1 with true and 0 with false. Maple has a package, called **Bits**, that provides a lot of support for working with bits and bit strings. For the purposes of this manual, however, we'll create some of this functionality from scratch. This gives us the opportunity to introduce some more general Maple commands that will be useful in a variety of contexts.

First of all, let's write the bitwise **AND** procedure. This procedure will take two arguments, which we'll call **a** and **b**, and should return 1 if both of the inputs are 1 and 0 if not. The body of the procedure will be an if statement that tests the proposition "$a = 1$ and $b = 1$". If this proposition is true, then the procedure will return the value 1. If not, in the else clause, it returns 0.

```
> AND := proc(a,b)
    if a=1 and b=1 then
      return 1;
    else
      return 0;
    end if;
  end proc:
```

```
> AND(1,0);
```

$$0 \qquad (1.26)$$

```
> AND(1,1);
```

$$1 \qquad (1.27)$$

The zip command

Now that we have a bitwise **AND** procedure, we can apply it to bit strings using Maple's **zip** command. The **zip** command requires three arguments. The first argument must be a binary procedure or function, *i.e.*, a procedure that takes two arguments. The second and third arguments must be two lists (or a similar data structure that holds multiple values, e.g., a matrix). The result of the command is the list whose entries are the result of applying the procedure to the corresponding values of the list.

```
> zip(AND,[1,1,0,0],[1,0,1,0]);
```

$$[1, 0, 0, 0] \qquad (1.28)$$

In the example above, the first element in the result is obtained by applying **AND** to the pair of first elements, namely (1,1). The second result value is obtained from the pair of values consisting of the second elements in the given list, namely (1,0), the third result from (0,1), and the last result from (0,0).

If the lists that are given to **zip** are of different lengths, the command will stop at the end of the the shorter list.

```
> zip(AND,[1,1],[1,0,1,0]);
```

$$[1, 0] \qquad (1.29)$$

If you want, however, you can provide an optional fourth argument to the command. This fourth argument will act as the default value for whichever list is shorter.

```
> zip(AND,[1,1],[1,0,1,0],0);
```

$$[1, 0, 0, 0] \qquad (1.30)$$

In this example, giving 0 as the optional argument means that when it gets to the third and fourth computations, it uses 0 as the value for the first (shorter) list. It is just as if we used [1,1,0,0] for the first list.

This last argument is helpful in many circumstances, but for operations on bit strings, it is not entirely satisfactory. The problem is that it would be more natural, in the context of **AND** on bit strings, to increase the length of the list by adding 0s on the left, rather than the right. That is, the result of **AND** on [1,1] and [1,0,1,0] should be the same as [0,0,1,1] and [1,0,1,0].

To overcome this, let's rewrite the **AND** procedure. Our new and improved **AND** will still take two inputs, but the inputs will be allowed to be either bits or bit strings. If the inputs are both bits (0 or 1), then we'll check to see if they are both 1 and return 1 if they are or 0 if not. If the inputs are lists, then we'll make sure the lists are the same length (possibly by adding 0s on the front of the list), and then use **zip** to **AND** the individual bits. Finally, if the input values are not bits and not lists, then we'll cause an **error** to be generated.

Implementing this will require some more Maple commands. The first thing that needs to be done is to determine the type of input. This will be done with an **if-elif-else** structure. First, the **if** condition is tested. In this case, the **if** condition will be that the input values are 1s or 0s. If the **if** condition passes, then the commands following the **then** keyword are executed. Otherwise, we jump ahead to the **elif** keyword which, like **if**, is followed by a boolean condition (that the input are lists) and the **then** keyword. If that condition is true, then the commands following the **elif**'s **then** are executed. Otherwise, we jump ahead to the **else** keyword and execute those commands.

We already know how to handle the first condition, that the inputs are 0s and 1s. This amounts to testing the proposition $(a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1)$. If that is the case, then we can use the same code from the original **AND** procedure above.

Using type to check for lists

For the second, **elif**, condition, we need to determine if the inputs are lists. We'll do this with the **type** command. Every Maple object, from the number **5** to the procedure **AND** is of one or more types. Just as in mathematics, we classify objects as integers or real numbers or matrices or functions, Maple classifies objects as **integer**, **realcons** (real constant), **Matrix**, or **procedure**. We test an object to see if it is of a particular type with the **type** command. For example, to see that 5 is an **integer** and a **realcons** but not a **Matrix**, but that **AND** is a **procedure** we issue the following commands.

```
> type(5,integer);
```

true **(1.31)**

```
> type(5,realcons);
```

true **(1.32)**

```
> type(5,Matrix);
```

false **(1.33)**

```
> type(AND,procedure);
```

true **(1.34)**

And to see whether an object is a list, we can test to see if it is of type **list**.

```
> type([1,2,3],list);
```

true **(1.35)**

```
> type(17,list);
```

false **(1.36)**

So our **elif** condition will use **type** to make sure both inputs are lists.

Padding lists with 0s

If the input are lists, we'll need to check their lengths and possibly add 0s to the front to make them the same length. We can get the length of a list using the **nops** command. **nops** is short for number of operands. It allows only one argument, which can be any expression. Its most common use is for determining the number of elements of a list or set.

```
> nops([1,0,1,1,0,1]);
```

$$6 \tag{1.37}$$

To compare the lengths of the two lists, we'll use the **max** command, which returns the largest of its arguments.

```
> max(3,5);
```

$$5 \tag{1.38}$$

Closely related to **nops** is the **op** command. This is another command with a variety of uses. In this case, we will use **op** to extract the elements of a list. To see why this is needed, consider the problem of adding a 0 to a list. The natural approach is to create a new list that has 0 as the first element, followed by the rest of the original list. You might try this:

```
> exList := [1,1,1,1];
```

$$exList := [1, 1, 1, 1] \tag{1.39}$$

```
> [0,exList];
```

$$[0, [1, 1, 1, 1]] \tag{1.40}$$

Instead of obtaining a list consisting of 0 followed by the **exList** elements, we get a list of two elements. The first element is 0 and the second element is the list **exList**. The elements of a list can be anything, including other lists. You can think of **exList** as a box in which we've stored five 1s. When we made the new list above, we made a new box and put a 0 and the **exList** in it producing a box within a box. In order to get the list we want, we need to extract the elements of the **exList**. We use **op** to take the data out of the **exList** box so that we can put them directly in this new box.

```
> [0,op(exList)];
```

$$[0, 1, 1, 1, 1] \tag{1.41}$$

In order to add more than one 0 at a time, we will use the **seq** command. This command creates a sequence of values. In its most basic form, seq requires two arguments. The first argument is an expression that determines the values of the sequence. The second argument has the form **i=m..n**, where **i** is an index variable that, like in a for loop, is sequentially assigned the integers between **m** and **n**, inclusive. Typically, the first argument will depend in some way on the value of **i**. For example, to produce the first few perfect squares, we can do the following.

```
> seq(i^2,i=1..10);
```

$$1, 4, 9, 16, 25, 36, 49, 64, 81, 100 \tag{1.42}$$

In our **AND** procedure, we will want all 0s, so the first argument will be 0 and the second argument will control how many are to be produced. For example, to make a sequence of 3 0s, we can issue the following command.

```
> seq(0,i=1..3);
```

$$0, 0, 0 \tag{1.43}$$

Note that we can put this in place of the single 0 from above to extend the **exList**.

```
> [seq(0,i=1..3),op(exList)];
```

$$[0, 0, 0, 1, 1, 1, 1] \tag{1.44}$$

Also, the bounds of the range can be expressions that depend on another variable. For example,

```
> upperBound := 10:
> seq(0,i=1..upperBound-2);
```

$$0, 0, 0, 0, 0, 0, 0, 0 \quad \textbf{(1.45)}$$

For example, if we want to extend `exList` to a list of length 10, we would need to add

```
> upperBound - nops(exList);
```

$$6 \quad \textbf{(1.46)}$$

0s to the front of `exList`. We can obtain these 6 0s by

```
> seq(0,i=1..upperBound-nops(exList));
```

$$0, 0, 0, 0, 0, 0 \quad \textbf{(1.47)}$$

The new list can thus be obtained as follows:

```
> [seq(0,i=1..upperBound-nops(exList)),op(exList)];
```

$$[0, 0, 0, 0, 0, 0, 1, 1, 1, 1] \quad \textbf{(1.48)}$$

```
> nops((1.48));
```

$$10 \quad \textbf{(1.49)}$$

Finally, if the range `i=m..n` is empty, *i.e.*, `n` is less than `m`, then `seq` outputs nothing. This means that we don't need to test the length of `exList` against the desired length because `seq` won't add anything if it doesn't need to.

```
> smallBound := 4:
> [seq(0,i=1..smallBound-nops(exList)),op(exList)];
```

$$[1, 1, 1, 1] \quad \textbf{(1.50)}$$

Now we have all the pieces in place to write the new and improved `AND` procedure.

```
> AND := proc(A,B)
    local i, maxlen, newA, newB;
    if (A=0 or A=1) and (B=0 or B=1) then
      if A=1 and B=1 then
        return 1;
      else
        return 0;
      end if;
    elif type(A,list) and type(B,list) then
      maxlen := max(nops(A),nops(B));
      newA := [seq(0,i=1..maxlen-nops(A)),op(A)];
      newB := [seq(0,i=1..maxlen-nops(B)),op(B)];
      return zip(AND,newA,newB);
    else
      error "Only bits and bit strings are allowed.";
    end if;
  end proc:
> AND([1,1,1],[1,0,1,1]);
```

$$[0, 0, 1, 1] \quad \textbf{(1.51)}$$

We leave the procedures for the other operations, NOT, OR, and XOR, to the reader.

1.2 Applications of Propositional Logic

In this section we will describe how Maple's computational abilities can be used to solve applied problems in propositional logic. In particular, we will consider consistency for system

specifications and Smullyan logic puzzles.

System Specifications

The textbook describes how system specifications can be translated into propositional logic and how it is important that the specifications be consistent. As suggested by the textbook, one way to determine whether a set of specifications is consistent is with truth tables.

Recall that a collection of propositions is consistent when there is an assignment of truth values to the propositional variables which makes all of the propositions in the collection true simultaneously. For example, consider the following collection of compound propositions: $p \rightarrow (q \wedge r)$, $p \vee q$, and $p \vee \neg r$. We can see that these propositions are consistent because we can satisfy all three with the assignment $p = false$, $q = true$, $r = false$. In Maple, we can confirm this by evaluating the list of propositions with that assignment of truth values.

```
> eval([p implies (q and r),p or q, p or (not r)],[p=false,q=
  true,r=false]);
```

$$[true, true, true] \quad \textbf{(1.52)}$$

To determine if a collection of propositions is consistent, we can create a truth table.

Consider Example 4 from Section 1.2 of the text. We translate the three specifications as the following list of propositions.

```
> specEx4 := [p or q, not p, p implies q];
```

$$specEx4 := [p \textbf{ or } q, \textbf{not } p, p \Rightarrow q] \quad \textbf{(1.53)}$$

Then we can construct the truth table exactly as we did in the previous section.

```
> for loopP in [true,false] do
    for loopQ in [true,false] do
      print(loopP,loopQ,eval(specEx4,[p=loopP,q=loopQ]));
    end do;
  end do;
```

$$true, true, [true, false, true]$$
$$true, false, [true, false, false]$$
$$false, true, [true, true, true]$$
$$false, false, [false, true, true] \quad \textbf{(1.54)}$$

We see that the only assignment of truth values that results in all three statements being satisfied is with $p = false$ and $q = true$.

We can make the output a bit easier to read if, instead of considering the truth table for the list of the propositions, we consider the proposition formed by the conjunction of the individual propositions: $(p \vee q) \wedge (\neg p) \wedge (p \rightarrow q)$.

```
> specEx4b := (p or q) and (not p) and (p implies q);
```

$$specEx4b := (p \textbf{ or } q) \textbf{ and not } p \textbf{ and } (p \Rightarrow q) \quad \textbf{(1.55)}$$

```
> for loopP in [true,false] do
    for loopQ in [true,false] do
      print(loopP,loopQ,eval(specEx4b,[p=loopP,q=loopQ]));
    end do;
  end do;
```

$$true, true, false$$
$$true, false, false$$

```
                    false, true, true
                    false, false, false                    (1.56)
```

In this case, the fact that the final truth value in the third row is true tells us that that assignment of truth values satisfies all of the propositions in the system specification.

If we add, as in Example 5, the proposition $\neg q$, we see that all of the assignments yield false for the conjunction of all four propositions.

```
> specEx5 := specEx4b and (not q);
```

$$specEx5 := (p \text{ or } q) \text{ and not } p \text{ and } (p \Rightarrow q) \text{ and not } q \qquad (1.57)$$

```
> for loopP in [true,false] do
    for loopQ in [true,false] do
      print(loopP,loopQ,eval(specEx5,[p=loopP,q=loopQ]));
    end do;
  end do;
```

```
                    true, true, false
                    true, false, false
                    false, true, false
                    false, false, false                    (1.58)
```

This tells us that this new system specification is inconsistent.

Logic Puzzles

Recall the knights and knaves puzzle presented in Example 7 of Section 1.2 of the text. In this puzzle, you are asked to imagine an island on which each inhabitant is either a knight and always tells the truth or is a knave and always lies. You meet two people named A and B. Person A says "B is a knight" and person B says "The two of us are opposite types." The puzzle is to determine who kind of inhabitants A and B are.

We can solve this problem with Maple using truth tables. First we must write A and B's statements as propositions. Let a represent the statement that A is a knight and b the statement that B is a knight. Then A's statement is "b", and B's statement is "$(a \wedge \neg b) \vee (\neg a \wedge b)$", as discussed in the text.

While these propositions precisely express the content of A and B's assertions, it does not capture the additional information that A and B are making the statements. We know, for instance, that A either always tells the truth (knight) or always lies (knave). If A is a knight, then we know the statement "b" is true. If A is not a knight, then we know the statement is false. In other words, the truth value of the proposition a, that A is a knight, is the same as the truth value of A's statement, and likewise for B. Therefore, we can capture the meaning of "A says proposition p" by the proposition $a \leftrightarrow p$. Using the procedure **IFF**, we can express the two statements in the puzzle in Maple as follows.

```
> Ex7A := IFF(a,b);
```

$$Ex7A := (a \Rightarrow b) \text{ and } (b \Rightarrow a) \qquad (1.59)$$

```
> Ex7B := IFF(b,(a and not b) or (not a and b));
```

$$Ex7B := (b \Rightarrow a \text{ and not } b \text{ or not } a \text{ and } b) \text{ and } (a \text{ and not } b \text{ or not } a \text{ and } b \Rightarrow b) \qquad (1.60)$$

Like the system specifications above, a solution to this puzzle will consist of an assignment of truth

values to the propositions a and b that make both people's statements true.

```
> for loopA in [true,false] do
    for loopB in [true,false] do
      print(loopA,loopB,eval(Ex7A and Ex7B,[a=loopA,b=loopB]));
    end do;
  end do;
```

true, true, false

true, false, false

false, true, false

false, false, true **(1.61)**

We see that the only way to satisfy both statements is when both a and b are false. That is, when A and B are both knaves.

1.3 Propositional Equivalences

In this section we consider logical equivalence of propositions and create a fairly sophisticated Maple procedure to check equivalence of propositions.

Logical Equivalence

Recall that propositions p and q are said to be logically equivalent if the biconditional $p \leftrightarrow q$ is a tautology. With Maple, we can test logical equivalence fairly easily by producing a truth table for the biconditional. For example, we can demonstrate De Morgan's Laws as follows.

```
> demorgan1 := IFF(not(p and q),not p or not q);
```

$$demorgan1 := \mathbf{not}\ (p\ \mathbf{and}\ q) \Rightarrow \mathbf{not}\ (p\ \mathbf{and}\ q) \quad \textbf{(1.62)}$$

```
> demorgan2 := IFF(not(p or q),not p and not q);
```

$$demorgan2 := \mathbf{not}\ (p\ \mathbf{or}\ q) \Rightarrow \mathbf{not}\ (p\ \mathbf{or}\ q) \quad \textbf{(1.63)}$$

```
> for loopP in [true,false] do
    for loopQ in [true,false] do
      print(loopP,loopQ,eval(demorgan1,[p=loopP,q=loopQ]));
    end do;
  end do;
```

true, true, true

true, false, true

false, true, true

false, false, true **(1.64)**

```
> for loopP in [true,false] do
    for loopQ in [true,false] do
      print(loopP,loopQ,eval(demorgan2,[p=loopP,q=loopQ]));
    end do;
  end do;
```

true, true, true

true, false, true

false, true, true

false, false, true **(1.65)**

We'd like to have a procedure, **AreEquivalent**, that would take two propositions and determine whether or not they are equivalent. Such a proposition will be our next goal, but it will require quite

a bit of work. The main hurdles for such a procedure are: (1) having Maple determine what propositional variables are used in the given compound propositions, and (2) without *a priori* knowledge of the number of propositional variables, having Maple test every possible assignment of truth values. Note that we could avoid both of these hurdles by insisting that the propositional variables be limited to a certain small set of names, perhaps p, q, r, and s. Then we could implement the procedure as four nested for loops. Many times not all four would be needed, which would add redundancy but would not impact functionality.

However, the two hurdles mentioned are not insurmountable, will provide a much more elegant and flexible procedure, and will also give us the opportunity to see examples of some important programming constructs.

Extracting Variables

The first hurdle is to get Maple to determine the variables used in a logical expression. Consider the following example.

```
> variableEx := ((p and q) or (p and not r)) and (s implies r);
```

$$variableEx := (p \text{ and } q \text{ or } p \text{ and not } r) \text{ and } (s \Rightarrow r) \qquad \textbf{(1.66)}$$

Our goal is to write a procedure that will, given the above expression, tells us that the variables in use are p, q, r, and s.

The op command and the name type

We've already discussed the **op** command, which, applied to a list returns the sequence underlying the list. However, **op** can be applied to any expression. If you apply **op** to an algebraic expression like **a*b**, it will return the sequence consisting of the operands 2 and 3.

```
> op(a*b);
```

$$a, b \qquad \textbf{(1.67)}$$

Applied to a more complicated algebraic expression, say **a*b+c**, **op** effectively obeys order of operations. In this expression, the addition is the last operation to be performed. In other words, the expression **a*b+c** is the sum of **a*b** and **c**. So **op** returns **a*b** and **c** as the operands to the addition.

```
> op(a*b+c);
```

$$a\,b, c \qquad \textbf{(1.68)}$$

(Note the multiplication sign is not printed.)

If the expression involved multiple additions, Maple considers the expression to be an addition of three terms.

```
> op(a+b/c+d);
```

$$a, \frac{b}{c}, d \qquad \textbf{(1.69)}$$

Boolean expressions are essentially the same, except **not**, **and**, and **or** are the operators instead of the arithmetic operations.

```
> op(variableEx);
```

$$p \text{ and } q \text{ or } p \text{ and not } r, s \Rightarrow r \qquad \textbf{(1.70)}$$

In this case, **op** indicates that the boolean expression had two operands which were joined by the final **and** operator. Note that **op** has the effect of removing the operator.

If a call to a procedure is involved, the result is the sequence of arguments of the procedure. For instance,

```
> op(IFF(not(p and q),not p or not q));
```

$$\text{not } (p \text{ and } q), \text{not } (p \text{ and } q) \qquad \textbf{(1.71)}$$

Since we're trying to obtain the variables in use, this is ideal. Our strategy is to keep applying the **op** command until we get down to variables. Doing this is a bit more complicated than saying it, of course.

How do we know when we have a variable as opposed to a compound proposition? In Maple, whenever you need to distinguish between different kinds of things, it makes sense to consider types. In fact, one of Maple's fundamental types is the **name** type. We use the **type** command as follows to see that $s \to r$ is not a name, but that s is.

```
> type(s implies r,'name');
```

$$\mathit{false} \qquad \textbf{(1.72)}$$

```
> type(s,'name');
```

$$\mathit{true} \qquad \textbf{(1.73)}$$

Illustrating with an example

We can now remove operators to obtain simpler expressions, and we have a way to test whether an expression is a variable or not. The general idea is that we keep applying the **op** command to the operands until we're down to nothing but names. The strategy we will use is a fairly typical one.

It is natural to create a list that will store the results of intermediate steps, so we'll start by initializing it to the list consisting of just one element, the expression we're analyzing.

```
> varExList := [variableEx];
```

$$\mathit{varExList} := [\,(p \text{ and } q \text{ or } p \text{ and not } r) \text{ and } (s \Rightarrow r)\,] \qquad \textbf{(1.74)}$$

To apply the **op** command to our expression, which is now the first element of the list, we use the list selection operation.

```
> op(varExList[1]);
```

$$p \text{ and } q \text{ or } p \text{ and not } r, s \Rightarrow r \qquad \textbf{(1.75)}$$

Next we'll replace the original expression with the result. We can replace elements in a list with the **subsop** command. This command can be used in a variety of ways depending on the particular arguments given. The form we use here will include two arguments. The second argument will be the list **varExList**. The first argument will be an equation of the form **i=e**, where **i** is an integer that refers to an index in the list and **e** is the expression that will replace whatever is currently in that position of the list. For example, we can replace the 5 with a 6 in the list **[2,4,5,8,10]** with the following command.

```
> subsop(3=6,[2,4,5,8,10]);
```

$$[2, 4, 6, 8, 10] \qquad \textbf{(1.76)}$$

We emphasize that the left side of the **i=e** equation is an integer representing the index of the element of the list that we are replacing while the right side is the replacement. In the above, the element 5 is in position 3 and is being replaced by the element 6, hence the use of **3=6**. Also note that this command does not modify an existing list (it creates a new list object), so we will need to reassign the result to the name of our list.

Using **subsop** on **varExList**, we obtain the following.

```
> varExList := subsop(1=op(varExList[1]),varExList);
```
$$varExList := [\, p \text{ and } q \text{ or } p \text{ and not } r, s \Rightarrow r \,] \quad \textbf{(1.77)}$$

The above command is substituting, for the first element of **varExList**, the result of applying the **op** command to the first element of **varExList**. Now repeat.

```
> varExList := subsop(1=op(varExList[1]),varExList);
```
$$varExList := [\, p \text{ and } q, p \text{ and not } r, s \Rightarrow r \,] \quad \textbf{(1.78)}$$

```
> varExList := subsop(1=op(varExList[1]),varExList);
```
$$varExList := [\, p, q, p \text{ and not } r, s \Rightarrow r \,] \quad \textbf{(1.79)}$$

Observe that it required 3 applications of **op** to the first element of the list. Continued use of **op** on **varExList[1]** will have no effect since the first element is a name. Likewise the second element is a name, so we move to element 3.

```
> varExList := subsop(3=op(varExList[3]),varExList);
```
$$varExList := [\, p, q, p, \text{not } r, s \Rightarrow r \,] \quad \textbf{(1.80)}$$

The procedure

The explicit example gives us the outline of our procedure:

1. Initialize a list, **L**, to the list consisting of the input expression as the sole element. Initialize an index variable, **i**, to one.
2. Test, using **type**, to see if the element of **L** referred to by the index **i** is a **name** or an expression. If it is a **name**, move on to the next element of the list by incrementing **i**.
3. If **L[i]** is not a **name**, the substitute that element of **L** with the result of applying the command **op** to it.
4. Repeat steps 2 and 3 until reaching the end of the list. This repetition is controlled by a while loop which continues as long as **i** does not exceed the length of the list.

Here is the implementation.

```
> GetVars := proc(exp)
    local L, i, j;
    L := [exp];
    i := 1;
    while i <= nops(L) do
      if type(L[i],name) then
        i := i + 1;
      else
        L := subsop(i=op(L[i]),L);
      end if;
    end do;
    return [op({op(L)})];
  end proc:
```

Note that the last line of this procedure makes use of Maple's set object to remove repetitions from the list of variables. We turn the list **L** into a set by applying **op** and enclosing the result in braces and then back into a list by applying **op** and enclosing it in brackets.

```
> GetVars(((p and q) or (p and not r)) and (s implies r));
```
$$[\, p, q, r, s \,] \quad \textbf{(1.81)}$$

```
> GetVars(prop23 implies IFF(Q or q,P and p));
```
$$[\, P, Q, p, prop23, q \,] \quad \textbf{(1.82)}$$

Truth Value Assignments

The second hurdle that we mentioned at the beginning of this section is that we don't know the number of propositional variables in advance. If we knew there would always be two variables, we

would use two nested for loops. But since we want our procedure to work with any number of variables, we need a different approach.

First note that since our **GetVars** procedure produces a list of variables, it is natural to model an assignment of truth values to variables as a list of truth values. For example,

```
> variableExVars := GetVars(variableEx);
```

$$variableExVars := [p, q, r, s] \quad \textbf{(1.83)}$$

```
> TAex := [true,true,false,true];
```

$$TAex := [true, true, false, true] \quad \textbf{(1.84)}$$

We consider the **TAex** list (for Truth Assignment example) to indicate that we assign the first variable of **variableExVars** to true, the second variable to true, the third to false, and the fourth to true.

The eval command

Recall the use of the **eval** command introduced above. In particular, by giving a list of equations of the form **var=val** as the second argument, we can evaluate the truth value of the proposition specified in the first argument with the assignments of the **val**s to the **var**s.

```
> eval(variableEx,[p=true,q=true,r=false,s=true]);
```

$$false \quad \textbf{(1.85)}$$

We can produce a list of assignments as follows. We saw how the **zip** command applies a binary function to the elements of two lists. We'll use **zip** with the function that takes a variable and a truth value and creates the appropriate equation. A numeric example will be most illustrative, but the syntax is identical for logical expressions.

```
> zip((a,b) -> a=b,[x,y,z],[5,3,9]);
```

$$[x = 5, y = 3, z = 9] \quad \textbf{(1.86)}$$

```
> eval(x+2*y-z,zip((a,b) -> a=b,[x,y,z],[5,3,9]));
```

$$2 \quad \textbf{(1.87)}$$

Indeed, $5 + 2 \cdot 3 - 9 = 2$. Note that the first argument to **zip**, **(a,b) -> a=b**, is a functional operator. A functional operator is a particular kind of procedure designed to mimic function notation. The left hand side indicates the input variables and the right hand side is an expression in terms of the variables that yields the value of the function.

The same approach will work with the assignment of truth values to propositional variables.

```
> eval(variableEx,zip((a,b) -> a=b,variableExVars,TAex));
```

$$false \quad \textbf{(1.88)}$$

Finding all possible truth assignments

Now that we know that we can effectively use lists of truth values to represent truth value assignments, we need a way to produce all such lists. We'll use a strategy similar to binary counting. Start with the list of all falses. Get the next list by changing the first element to true. For the next assignment, change the first element back to false and the second element to true. Then change the first element to true. Then change the first true to false, the second true to false, and the third element becomes true. Continue in this pattern: given a list of truth values, obtain the next list by changing the left-most false to true and changing all trues up until that first false into false. (It is left to the reader to verify that this produces all possible truth value assignments.)

We implement this idea in the **NextTA** procedure (for Next Truth Assignment). The **NextTA**

procedure will accept a list of truth values as input and return the "next" list. The main work of this procedure is done inside of a for loop. The for loop considers each position in the list of truth values in turn. If the value in the current position is true, then it is changed to false. On the other hand, if the value is false, then it is changed to true and the procedure is terminated by returning the list of truth values. If the for loop ends without having returned a new list, then the input to the procedure was all trues, which is the last possibility, and the procedure returns **NULL** to indicate that there is no next truth assignment.

```
> NextTA := proc(A)
    local i, new;
    new := A;
    for i from 1 to nops(A) do
      if new[i] then
        new[i] := false;
      else
        new[i] := true;
        return new;
      end if;
    end do;
    return NULL;
  end proc:
```

Note that the **new** variable is necessary because procedure arguments can not be assigned to. Also note that, in this procedure, we are able to modify list elements using the assignment operator with the **list[index] := value;** syntax. This was not possible in **GetVars**, because in that case we were potentially replacing a single element with a sequence of values.

Logical Equivalence Implementation

We now have the necessary pieces in place to write the promised **AreEquivalent** procedure. This procedure accepts two propositions as arguments and returns true if they are equivalent and false otherwise.

The procedure proceeds as follows:

1. First we assign the function **(a,b) -> a=b** used in the **zip** command to the name **eqZip**. (This is a bit more efficient than including the definition of the function in the **zip** command.)
2. Then we create the biconditional, which we name **Bicond**, that asserts the equivalence of the two propositions. We also use the **GetVars** procedure to determine the list of variables used in the propositions and initialize the truth assignment variable **TA** to the appropriately sized list of all false values.
3. Then we begin a while loop. As long as **TA** is not NULL, we evaluate the biconditional **Bicond** on the truth assignment. If this truth value, is false, we know that the biconditional is not a tautology and thus the propositions are not equivalent and we immediately return false. Otherwise, we use **NextTA** to update **TA** to the next truth assignment.
4. If the while loop terminates, that indicates that all possible truth assignments have been applied to the biconditional and that each one evaluated true, otherwise the procedure would have returned false and terminated. Thus the biconditional is a tautology and true is returned.

Here is the implementation.

```
> AreEquivalent := proc(P,Q)
    local eqZip, Bicond, Vars, numVars, i, TA, val;
    eqZip := (a,b) -> a=b;
    Bicond := IFF(P,Q);
    Vars := GetVars(Bicond);
    numVars := nops(Vars);
```

```
    TA := [seq(false,i=1..numVars)];
    while TA <> NULL do
      val := eval(Bicond,zip(eqZip,Vars,TA));
      if not val then
        return false;
      end if;
      TA := NextTA(TA);
    end do;
    return true;
  end proc:
```

We can use this to computationally verify that $\neg (p \vee (\neg p \wedge q)) \equiv \neg p \wedge \neg q$. This was shown in Example 7 of Section 1.3 of the text via equivalences.

```
> AreEquivalent(not(p or (not p and q)),not p and not q);
```

$$true \tag{1.89}$$

1.4 Predicates and Quantifiers

In this section we will see how Maple can be used to explore propositional functions and their quantification over a finite universe. We can think about a propositional function P as a function, or procedure, that takes as input a member of the domain and that outputs a truth value.

For example, let $P(x)$ denote the statement "$x > 0$". We will construct a procedure that takes x as input and returns true or false as appropriate. Note however that Maple does not automatically evaluate inequalities to their truth value.

```
> 3 > 0;
```

$$0 < 3 \tag{1.90}$$

To have Maple evaluate such expressions to true or false, use the command **evalb**, for evaluate boolean.

```
> evalb(3>0);
```

$$true \tag{1.91}$$

We'll construct our procedure using the functional operator syntax that was discussed in the previous section. In this case, we'll assign the function to the name **GT0** (for greater than 0) and we include **evalb** in the definition of the propositional function.

```
> GT0 := x -> evalb(x > 0);
```

$$GT0 := x \rightarrow evalb(0 < x) \tag{1.92}$$

```
> GT0(3);
```

$$true \tag{1.93}$$

```
> GT0(-2);
```

$$false \tag{1.94}$$

Quantifiers

For finite domains, we can use Maple to determine the truth value of universally and existentially quantified statements. We will create two procedures, **Universal** and **Existential**. Both of these procedures will accept two arguments: a propositional function of one variable and a list or set representing the domain.

First consider universal quantification. This procedure should return true only when all members of the domain satisfy the propositional function and false if even one element of the domain fails to satisfy the predicate. The procedure will loop through the elements of the domain. If the

propositional function ever returns false, then the procedure will immediately return false. On the other hand, if we get through the loop without encountering false, then we can conclude that the universally quantified statement is true.

```
> Universal := proc(P,D)
    local d;
    for d in D do
      if not P(d) then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

```
> Universal(GT0,[1,2,3,4,5]);
```

$$true \tag{1.95}$$

```
> Universal(GT0,[1,2,3,4,5,-7]);
```

$$false \tag{1.96}$$

For existential quantification, if we ever encounter a member of the domain for which the propositional function returns true, then the existential statement is true. If the for loop terminates without having found such a value, then we return false.

```
> Existential := proc(P,D)
    local d;
    for d in D do
      if P(d) then
        return true;
      end if;
    end do;
    return false;
  end proc:
```

```
> Existential(GT0,[-3,-4,-5,-6,-7]);
```

$$false \tag{1.97}$$

```
> Existential(GT0,[-3,-4,-5,11,-7]);
```

$$true \tag{1.98}$$

1.5 Nested Quantifiers

In this section we'll consider propositions with nested quantifiers such as $\exists\, y \forall\, x(x+y=0)$ and $\forall\, y \exists\, x(x+y=0)$. These propositions are the subject of Example 4 of Section 1.5 in the textbook. Recall that the first statement is false while the second is true. Nested quantification can be difficult to understand, but writing procedures to test propositions such as those two examples can help make their interpretation clearer.

We'll create a procedure, **ExistsForAll**, which will accept two arguments, a propositional function with two input values and a domain which will be considered common for both variables. Consider the example mentioned in the previous paragraph:

```
> sumto0 := (x,y) -> evalb(x+y=0);
```

$$sumto0 := (x, y) \rightarrow evalb(x + y = 0) \tag{1.99}$$

```
> sumto0Domain := [seq(i,i=-10..10)];
```

$$sumto0Domain := [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \tag{1.100}$$

Now we will see how to write **ExistsForAll**. Note that we will keep our variables consistent with those in the examples above. That is, the "outer," existentially quantified, variable will be y and the "inner," universally quantified, variable will be x. To see how to write this procedure, think about how we would go about testing $\exists\, y \forall\, x(x + y = 0)$. The claim is that there is some y so that all values of x satisfy the propositional function. So we'll test each value of y in turn. Suppose we start with -10. Then we're checking to see if $\forall\, x(x - 10 = 0)$. We consider, in sequence, each possible value for x. If any value of x fails to satisfy the proposition, then we know that the universal statement is false and we can stop checking values of x and move on to the next possible y. On the other hand, if all of the x values satisfy the proposition, then we've found a y that demonstrates that the proposition is true.

We'll use two for loops to check all possible values of y and x, respectively. Note that the inner, x loop, can be stopped as soon as we find a value that fails to satisfy the proposition. We can abort a loop with the **break** command, which causes the innermost loop to terminate. In this procedure, that means that we move on to the next possible y.

Also note that the outer loop can be stopped as soon as we find a y value that satisfies the predicate for all possible x. This is a bit trickier to program. What we'll do is to assume that a y value "works" (*i.e.*, satisfies the proposition for all x) until we discover otherwise. We'll set a variable, **yworks**, equal to true as the first statement of the for y loop. Immediately after each new y value is chosen, **yworks** is set to true indicating that we "believe" that the current y value satisfies the requirements. If an x is found so that the proposition fails, then before using **break** to end the x loop, we'll set **yworks** to false. Thus, at the conclusion of the inner x loop, **yworks** is true when the current value of y causes every possible value of x to satisfy the proposition. In that case, we can abort the procedure and return true. If the y loop terminates, then no such y was found and so the procedure returns false.

Here is the procedure.

```
> ExistsForAll := proc(P,D)
    local x, y, yworks;
    for y in D do
      yworks := true;
      for x in D do
        if not P(x,y) then
          yworks := false;
          break;
        end if;
      end do;
      if yworks then
        return true;
      end if;
    end do;
    return false;
  end proc:
```

As was mentioned, this should return false for **sumto0**.

```
> ExistsForAll(sumto0,sumto0Domain);
```

$$false \qquad \textbf{(1.101)}$$

On the other hand, consider the predicate that asserts that the product is 0.

```
> multo0 := (x,y) -> evalb(x*y = 0);
```

$$multo0 := (x, y) \rightarrow evalb(x\,y = 0) \qquad \textbf{(1.102)}$$

```
> ExistsForAll(multto0,sumto0Domain);
```

true **(1.103)**

As it can be a useful way to help better understand nested quantifiers, we will leave it to the reader to write the procedure for $\forall y \exists x P(x, y)$.

1.6 Rules of Inference

In this section we'll see how Maple can be used to verify the validity of arguments in propositional logic. In particular, we'll write a procedure, that, given a list of premises and a possible conclusion, will determine whether or not the conclusion necessarily follows from the premises. Recall from Definition 1 in the text that an argument is defined to be a sequence of propositions, the last of which is called the conclusion and all others are premises. Also recall that an argument $p_1, p_2, \ldots, p_n, q$ is said to be valid when $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \rightarrow q$ is a tautology.

We can use the **AreEquivalent** procedure from Section 1.4 of this manual to test whether a proposition is a tautology. Since a tautology is merely a proposition that is always true, it is logically equivalent to the proposition "true". So we can see if a proposition **P** is a tautology by executing the command **AreEquivalent(P,true)**. For example, we can confirm *modus tollens*. (See Table 1 in Section 1.6 of the text for the tautologies associated to the rules of inference.)

```
> AreEquivalent((not q and (p implies q)) implies not p,true);
```

true **(1.104)**

To determine if an argument is valid, we need to: (1) form the conjunction of the premises, (2) form the conditional statement that the premises imply the conclusion, and (3) test the resulting proposition with **AreEquivalent**. This **IsValid** procedure below will accept as input an argument, *i.e.*, a list of propositions, and return true if the argument is valid.

```
> IsValid := proc(L::list)
    local premises, i;
    premises := L[1];
    for i from 2 to nops(L)-1 do
      premises := premises and L[i];
    end do;
    AreEquivalent(premises implies L[-1],true);
  end proc:
```

Note that we use **L[-1]** to obtain the last element of the list. It is a useful property of selection that negative integers count from the end of the list.

We can use this procedure to verify that the argument described in Exercise 12 of Section 1.6 of the text is in fact valid.

```
> IsValid([(p and t) implies (r or s),q implies (u and t),u
  implies p, not s,q implies r]);
```

true **(1.105)**

Note that Exercise 12, which this example was based on, asks you to verify the validity of the argument using rules of inference. It is important to note that our procedure did not do that. **IsValid** used truth tables to establish validity. It would be considerably more difficult to program Maple to check validity with rules of inference than it was to do so with truth tables. On the other hand, for a human it is typically much easier to use rules of inference than a truth table. Especially with practice, you will develop an intuition about logical arguments that cannot be easily created in a

computer.

Finding Conclusions (optional)

In the remainder of this section we'll consider a slightly different question: given a list of premises, what conclusions can you draw using valid arguments? We'll approach this problem in Maple in a straightforward (and naïve) way: generate possible conclusions and use `IsValid` to determine which are valid conclusions.

Making compound propositions

To generate possible conclusions, we'll use the following procedure, `AllCompound`. This procedure takes a list of propositions and produces all possible propositions formed from one logical connective (from and, or, and implies) and two of the given propositions, along with the negations of the propositions. To avoid including some trivialities, we'll exclude those propositions that are tautologies or contradictions.

The procedure is provided below. We provide no additional explanation other than to mention the use, once again, of the `[op({op(L)})]` structure to remove duplicates. Also note that the original propositions are included in the result since these are equivalent to their conjunction with themselves which the procedure includes in the output.

```
> AllCompound := proc(L)
    local p, q, tempL, PropList;
    PropList := [];
    tempL := L;
    for p in L do
      tempL := [op(tempL),not p];
    end do;
    for p in tempL do
      for q in tempL do
        if not AreEquivalent(p and q,true) and
            not AreEquivalent(p and q,false) then
          PropList := [op(PropList),p and q];
        end if;
        if not AreEquivalent(p or q,true) and
            not AreEquivalent(p or q,false) then
          PropList := [op(PropList),p or q];
        end if;
        if not AreEquivalent(p implies q,true) and
            not AreEquivalent(p implies q,false) then
          PropList := [op(PropList),p implies q];
        end if;
      end do;
    end do;
    return [op({op(PropList)})];
  end proc:
```

Finding valid conclusions

Now we write a procedure to explore possible conclusions given a set of premises. This procedure will take two arguments. The first will be a list of premises. The second a positive integer indicating the number of times that `AllCompound` should, recursively, be used to generate possibilities. You will generally not want to use any number other than 1 for this second value as the time requirement for this procedure can be quite substantial.

The operation of this procedure is very straightforward. First, it determines the variables used in the

premises. These become the basis for the potential conclusions. Second, the procedure applies the **AllCompound** procedure to generate the propositions formed using one logical connective and two of the variables and their negations. (Note that the resulting list will include the variables with no connective as well.) Third, the procedure joins the premises into one proposition by conjunction. And finally, each possible conclusion is evaluated with the **IsValid** procedure.

```
> FindConsequences := proc(Premises,level)
    local Vars, possibleC, C, c, P, i;
    Vars := GetVars(Premises);
    possibleC := Vars;
    for i from 1 to level do
      possibleC := AllCompound(possibleC);
    end do;
    C := [];
    P := Premises[1];
    for i from 2 to nops(Premises) do
      P := P and Premises[i];
    end do;
    for c in possibleC do
      if AreEquivalent(P implies c,true) then
        C := [op(C),c];
      end if;
    end do;
    return C;
  end proc:
```

Here is the result of applying **FindConsequences** to the premises of Exercise 12 with only one iteration of **AllCompound**. (With two iterations of **AllCompound**, the procedure takes quite some time to complete and produces thousands of valid conclusions.)

```
> FindConsequences([(p and t) implies (r or s),q implies (u and
  t),u implies p, not s],1);
```

$$[\text{not } s, \text{not } (p \text{ and } s), \text{not } (q \text{ and } s), \text{not } (r \text{ and } s), \text{not } (s \text{ and } p), \text{not } (s \text{ and } q), \text{not } (s \text{ and } r), \text{not } (s \text{ and } t), \text{not } (s \text{ and } u), \text{not } (t \text{ and } s), \text{not } (u \text{ and } s), p \text{ or not } q, p \text{ or not } s, p \text{ or not } u, q \text{ or not } s, r \text{ or not } q, r \text{ or not } s, t \text{ or not } q, t \text{ or not } s, u \text{ or not } q, u \text{ or not } s, \text{not } q \text{ or } p, \text{not } q \text{ or } r, \text{not } q \text{ or } t, \text{not } q \text{ or } u, \text{not } s \text{ or } p, \text{not } s \text{ or } q, \text{not } s \text{ or } r, \text{not } s \text{ or } t, \text{not } s \text{ or } u, \text{not } u \text{ or } p, p \Rightarrow \text{not } s, q \Rightarrow p, q \Rightarrow r, q \Rightarrow t, q \Rightarrow u, q \Rightarrow \text{not } s, r \Rightarrow \text{not } s, s \Rightarrow p, s \Rightarrow q, s \Rightarrow r, s \Rightarrow t, s \Rightarrow u, s \Rightarrow \text{not } p, s \Rightarrow \text{not } q, s \Rightarrow \text{not } r, s \Rightarrow \text{not } s, s \Rightarrow \text{not } t, s \Rightarrow \text{not } u, t \Rightarrow \text{not } s, u \Rightarrow p, u \Rightarrow \text{not } s, \text{not } p \Rightarrow \text{not } q, \text{not } p \Rightarrow \text{not } s, \text{not } p \Rightarrow \text{not } u, \text{not } q \Rightarrow \text{not } s, \text{not } r \Rightarrow \text{not } q, \text{not } r \Rightarrow \text{not } s, \text{not } t \Rightarrow \text{not } q, \text{not } t \Rightarrow \text{not } s, \text{not } u \Rightarrow \text{not } q, \text{not } u \Rightarrow \text{not } s] \quad \textbf{(1.106)}$$

```
> nops((1.106));
```


(1.107)

Observe that some of the conclusions are just merely restating premises. But even after eliminating those, there are still 60 valid conclusions involving at most two of the propositional variables. Most of those conclusions are going to be fairly uninteresting in any particular context. This illustrates a fundamental difficulty with computer assisted proof. Neither checking the validity of conclusions nor generating valid conclusions from a list of premises are particularly difficult. The difficulty is in

creating heuristics and other mechanisms to help direct the computer to useful results.

1.7 Introduction to Proofs

In this section we will see how Maple can be used to find counterexamples. This is, perhaps, the proof technique most suitable to Maple's computational abilities.

Example 14 of Section 1.7 of the textbook considers the statement "Every positive integer is the sum of the squares of two integers." This is demonstrated to be false with 3 as a counterexample. Here, we'll consider the related statement that "every positive integer is the sum of the squares of *three* integers." This statement is also false.

Finding a counterexample

To find a counterexample, we'll create a procedure that, given an integer, looks for three integers such that the sum of their squares are equal to the given integer. If the procedure finds three such integers, it will return a list containing them. On the other hand, if it cannot find three such integers, it will return false. Here is the procedure:

```
> Find3Squares := proc(n)
    local a, b, c, max;
    max := floor(sqrt(n));
    for a from 0 to max do
      for b from 0 to max do
        for c from 0 to max do
          if n = a^2 + b^2 + c^2 then
            return [a,b,c];
          end if;
        end do;
      end do;
    end do;
    return false;
  end proc:
```

The **Find3Squares** procedure is fairly straightforward. We use three for loops to check all possible values of a, b, and c. Each for loop can range from 0 to the floor of $\sqrt{n}$ (the floor of a number is the largest integer that is less than or equal to the number). Note that these bounds are sufficient to guarantee that if n can be written as the sum of the squares of three integers, then this procedure will find them. We observe that 3, the counterexample from Example 14, can be written as the sum of three squares.

```
> Find3Squares(3);
```

$$[1, 1, 1] \qquad \textbf{(1.108)}$$

To find a counterexample, we write a procedure that, starting with 1, tests numbers with **Find3Squares**, until it finds a value that causes **Find3Squares** to return false.

```
> Find3Counter := proc()
    local n;
    n := 1;
    while Find3Squares(n) <> false do
      n := n + 1;
    end do;
    return n;
  end proc:
```

First observe that this procedure does not take an argument, but the parentheses are still required.

Also note that the while loop is controlled by the return value of the **Find3Squares** procedure. This is a fairly common approach when you are looking for an input value that will cause another procedure to return a desired result.

To find the counterexample, all we need to do is run the procedure.

```
> Find3Counter();
```

$$7 \qquad \textbf{(1.109)}$$

This indicates that 7 is an integer that is not the sum of the squares of three integers.

Let's take a step back and review what we did. Our goal was to disprove the statement $\forall\, n\, P(n)$ where $P(n)$ is the statement that "n can be written as the sum of the squares of three integers." We first wrote **Find3Squares**, which is a procedure whose goal is to find three integers whose squares add to n. Observe that if **Find3Squares** returns three values for a given n, then we know $P(n)$ is true for that n. Only after we wrote the **Find3Squares** procedure did we write **Find3Counter**, whose task was to find a counterexample to the universal statement. This is a common strategy when using a computer to find a counterexample — write a program that seeks to verify the $P(n)$ statement for n and then look to find a value of n that causes the program to fail.

Proof

We have not yet actually disproved the statement that "every positive integer is the sum of the squares of three integers." The procedures we wrote found a candidate for a counterexample, but we don't yet know for sure that it is in fact a counterexample (after all, our program could be flawed). To prove the statement is false, we must prove that 7 is in fact a counterexample. We can approach this in one of two ways. The first approach is to follow the Solution to Example 17 in Section 1.8 of the text.

The alternative is to prove the correctness of our algorithm. Specifically, we need to prove the statement: "The positive integer n can be written as the sum of the squares of three integers if and only if **Find3Squares(n)** returns a list of three integers." Let's prove this biconditional.

First we'll prove that: if the positive integer n can be written as the sum of the squares of three integers, then **Find3Squares(n)** returns a list of three integers. We'll use a direct proof. We assume that n can be written as the sum of three squares. Say $n = a^2 + b^2 + c^2$ for integers a, b, c. Note that we may take a, b, and c to be nonnegative integers, since an integer and its negative have the same square. Also, $n = a^2 + b^2 + c^2 \geq a^2$. So $n \geq a^2$ and $a \geq 0$, which means that $a \leq \sqrt{n}$. Since a is an integer and is less than or equal to the square root of n, a must be less than or equal to the floor of $\sqrt{n}$ since the floor of a real number is the greatest integer less than or equal to the real number. The same argument applies to b and c. We started with $n = a^2 + b^2 + c^2$ and have now shown that a, b, and c can be assumed to be nonnegative integers and must be less than or equal to the floor of the square root of n. The nested for loops in **Find3Squares** set **a**, **b**, and **c** equal to every possible combination of integers between 0 and **max**, which is the floor of the square root of **n**. Hence, **a**, **b**, and **c** must, at some point during the execution of **Find3Squares**, be set to a, b, and c, and thus the condition that **n=a^2+b^2+c^2** will be satisfied and **[a,b,c]** will be returned by the procedure. We've assumed that n can be written as the sum of three squares and concluded that **Find3Squares(n)** must return a list of the integers.

The converse is: if **Find3Squares** returns a list of three integers, then n can be written as the sum of the squares of three integers. This is nearly obvious, since if **Find3Squares(n)** returns **[a, b,c]**, it must have been because **n=a^2+b^2+c^2** was found to be true.

Therefore, the **Find3Squares** procedure is correct and since **Find3Squares(7)** returns false, we can conclude that 7 is, in fact, a counterexample to the assertion that every positive integer is the sum of the squares of three integers.

We will typically not be proving the correctness of procedures in this manual — that is a topic for another course. The above merely serves to illustrate how you can approach such a proof and to reinforce the principle that just because a program produces output does not guarantee that the program or the output is correct.

1.8 Proof Methods and Strategy

In this section, we will consider two additional proof techniques that Maple can assist with: exhaustive proofs and existence proofs.

Exhaustive Proof

In an exhaustive proof we must check all possibilities. For an exhaustive proof to be feasible by hand, there must be a fairly small number of possibilities. With computer software such as Maple, though, the number of possibilities can be greatly expanded. Consider Example 2 from Section 1.8 of the text. There it was determined by hand that the only consecutive positive integers not exceeding 100 that are perfect powers are 8 and 9.

We will consider a variation of this problem: prove that the only consecutive positive integers not exceeding 100,000,000 that are perfect powers are 8 and 9.

Our approach will be the same as was used in the text. We will generate all the perfect powers not exceeding the maximum value and then we will check to see which of the perfect powers occur as a consecutive pair. We will implement this strategy with two procedures. The first procedure, **FindPowers**, will accept as an argument the maximum value to consider (*e.g.*, 100) and will return all of the perfect powers no greater than that maximum. The second procedure, **FindConsecutivePowers**, will also accept the maximum value as its input. It will use **FindPowers** to generate the powers and then check them for consecutive pairs.

For the first procedure, **FindPowers**, we need to generate all perfect powers up to the given limit. To do this we'll use a nested pair of loops for the exponent (**p**) and the base (**b**). Both of the loops will be while loops controlled by a boolean variable, **continuep** and **continueb**. In the inner loop, we check to see if **b^p** is greater than the limit. If it is, then we set **continueb** to false, which terminates the loop, and if not, we add **b^p** to the list of perfect powers and increment **b**. Once the **b** loop has terminated, we increment the power **p**. If **2^p** exceeds the limit, then we know that no more exponents need to be checked and we terminate the outer loop by setting **continuep** to false.

```
> FindPowers := proc(n::posint)
    local L, b, p, continuep, continueb;
    L := [];
    p := 2;
    continuep := true;
    while continuep do
      b := 1;
      continueb := true;
      while continueb do
        if b^p > n then
          continueb := false;
        else
```

```
            L := [op(L),b^p];
            b := b + 1;
          end if;
        end do;
        p := p + 1;
        if 2^p > n then
          continuep := false;
        end if;
      end do;
      return {op(L)};
    end proc:
```

(Note that we return the set formed from the elements of the list of perfect powers so as to remove duplicates.) We can confirm that the list of powers produced by this algorithm is the same as the powers considered in Example 2 from the text.

```
> FindPowers(100);
```

$$\{1, 4, 8, 9, 16, 25, 27, 32, 36, 49, 64, 81, 100\} \qquad \textbf{(1.110)}$$

The second procedure, **FindConsecutivePowers**, begins by calling **FindPowers** and storing the set of perfect powers as **powers**. Then we begin a for loop, using the **for x in S** structure with **x** a variable and **S** a set or list. This sets the variable **x** equal to each element of the set or list **S** in turn. In our procedure, we are considering each perfect power **x** in the set **powers**. In the body of the loop, we check to see if the next consecutive integer, **x+1**, is also a perfect power by using the **in** operator. The syntax **element in obj** tests to see if the **element** is a member of the set or list **obj**. When we find consecutive perfect powers, we **print** them. At the end of the procedure, the value **NULL** is returned, which means that the procedure will not display anything other than what was printed.

```
> FindConsecutivePowers := proc(n)
    local powers, x;
    powers := FindPowers(n);
    for x in powers do
      if x + 1 in powers then
        print(x,x+1);
      end if;
    end do;
    return NULL;
  end proc:
```

Subject to the correctness of our procedures, we can demonstrate that the only consecutive perfect powers less than 100,000,000 are 8 and 9 by running the procedure.

```
> FindConsecutivePowers(100000000);
```

$$8, 9 \qquad \textbf{(1.111)}$$

It is worth pointing out that in fact, 8 and 9 are the only consecutive perfect powers. That assertion was conjectured by Eugéne Charles Catalan in 1844 and was finally proven in 2002 by Preda Mihăilescu.

Existence Proofs

Proofs of existence can also benefit from Maple. Consider Example 10 in Section 1.8 of the text. This example asks, "Show that there is a positive integer than can be written as the sum of cubes of positive integers in two different ways." The solution reports that 1729 is such an integer and indicates that a computer search was used to find that value. Let's see how this can be done.

The basic idea will be to generate numbers that can be written as the sum of cubes. If we generate a number twice, that will tell us that the number can be written as the sum of cubes in two different ways. We'll create a list **L** and every time we generate a new sum of two cubes, we'll check to see if that number is already in **L** using the **in** operator. If the new value is already in **L**, then that's the number we're looking for. Otherwise, we add the new number to **L** and generate a new sum of two cubes.

We'll generate the sums of cubes with two nested loops that control integers **a** and **b**. The inner loop will be a for loop that causes **b** to range from 1 to the value of **a**. Using **a** as the maximum value means that **b** will always be less than or equal to **a** and so the procedure will not falsely report results coming from commutativity of addition (e.g., $9 = 2^3 + 1^3 = 1^3 + 2^3$). The outer loop will be a **while true do** loop. The value of **a** will be initialized to 1 and incremented by 1 after the inner **b** loop completes. The **while true do** loop is called an infinite loop because it will never stop on its own. When the procedure finds an integer which can be written as the sum of cubes in two different ways, the procedure will return that value which ends the procedure. The infinite loop means that the value of **a** will continue getting larger and larger with no upper bound. This is useful because we don't know how large the numbers will need to be in order to find the example. However, infinite loops should be used with caution, especially if you're not certain that the procedure will terminate in a reasonable amount of time.

Here is the procedure and its result.

```
> TwoCubes := proc()
    local L, a, b, n;
    L := [];
    a := 1;
    while true do
      for b from 1 to a do
        n := a^3 + b^3;
        if n in L then
          return n;
        else
          L := [op(L),n];
        end if;
      end do;
      a := a + 1;
    end do;
  end proc:
> TwoCubes();
```

1729 **(1.112)**

Solutions to Computer Projects and Computations and Explorations

Computer Projects 3

Given a compound proposition, determine whether it is satisfiable by checking its truth value for all positive assignments of truth values to its propositional variables.

Solution: Recall that a proposition is satisfiable if there is at least one assignment of truth values to variables that results in a true proposition. Our approach will be similar to the way we checked for logical equivalence in the **AreEquivalent** procedure in Section 1.3.

We create a procedure, **IsSatisfiable**, that checks all possible assignments of truth values to the propositional variables. The **IsSatisfiable** procedure accepts one argument, a logical expression. It will print out all, if any, truth value assignments that satisfy the proposition. We will initialize a **result** variable to false. When an assignment that satisfies the proposition is found, this variable is set to true and the assignment is printed. After all possible assignments are considered, the procedure returns the **result** variable.

Since this procedure is otherwise very similar to **AreEquivalent**, we offer no further explanation.

```
> IsSatisfiable := proc(P)
    local eqZip, Vars, numVars, i, TA, val, TAeqns, result;
    result := false;
    eqZip := (a,b) -> a=b;
    Vars := GetVars(P);
    numVars := nops(Vars);
    TA := [seq(false,i=1..numVars)];
    while TA <> NULL do
      TAeqns := zip(eqZip,Vars,TA);
      val := eval(P,TAeqns);
      if val then
        result := true;
        print(TAeqns);
      end if;
      TA := NextTA(TA);
    end do;
    return result;
  end proc:
```

We apply this procedure to the propositions in Example 9 of Section 1.3 of the text.

```
> IsSatisfiable((p or not q) and (q or not r) and (r or not
  p));
```

$$[p = false, q = false, r = false]$$

$$[p = true, q = true, r = true]$$

$$true \qquad \textbf{(1.113)}$$

```
> IsSatisfiable((p or q or r) and (not p or not q or not r));
```

$$[p = true, q = false, r = false]$$

$$[p = false, q = true, r = false]$$

$$[p = true, q = true, r = false]$$

$$[p = false, q = false, r = true]$$

$$[p = true, q = false, r = true]$$

$$[p = false, q = true, r = true]$$

$$true \qquad \textbf{(1.114)}$$

```
> IsSatisfiable((p or not q) and (q or not r) and (r or not
  p) and (p or q or r) and (not p or not q or not r));
```

$$false \qquad \textbf{(1.115)}$$

Computations and Explorations 1

Look for positive integers that are not the sum of the cubes of eight positive integers.

Solution: We will find integers n such that $n \neq a_1^3 + a_2^3 + \cdots + a_8^3$ for any integers $a_1, a_2, \ldots, a_8$. We can restate the problem as finding a counterexample to the assertion that every integer can be written as the sum of eight cubes.

Our approach will be to generate all of the integers that are equal to the sum of eight cubes and then check to see what integers are missing. For this, we need to set a limit n, *i.e.*, the maximum integer that we're considering as a possible answer to the question. For instance, we might restrict our search to integers less than 100. Then we know that each a_i is at most the cube root of this limit, since $a_i^3 < n$.

We'll also want to make our approach as efficient as possible in order to find as many such integers as we can. So we make the following observations.

Every number that can be expressed as the sum of eight cubes can be expressed as the sum of two integers each of which is the sum of four cubes. Those, in turn, can be expresses as the sum of two integers which are the sum of two cubes each. That is,

$$n = \left[\left(a_1^3 + a_2^3\right) + \left(a_3^3 + a_4^3\right)\right] + \left[\left(a_5^3 + a_6^3\right) + \left(a_7^3 + a_8^3\right)\right].$$

This means that we don't need to write a procedure to find all possible sums of eight cubes. Instead, we'll write a procedure that, given a list of numbers, will find all possible sums of two numbers that are both in that list. If we apply this procedure to the cubes of the numbers from 0 through $\sqrt[3]{n}$, that will produce all numbers that are the sums of two cubes. Applying the procedure again to that result will give all numbers that are the sum of four cubes. And applying it once again to that result will produce the numbers (up to n) that are the sum of eight cubes.

Additionally, when we find all the possible sums of two integers, we will exclude any sum that exceeds our maximum. Recall that we've determined that if an integer less than or equal to n can be written as the sum of cubes, then it can be written as the sum of cubes with each a_i between 0 and $\sqrt[3]{n}$. There will be numbers greater than n that are generated as the sum of cubes of integers less that $\sqrt[3]{n}$, however, these do not provide us with any information about numbers that cannot be generated as the sum of eight cubes. And excluding them at each step of the process decreases the number of sums that need to be computed.

Finally, we may assume that the second number is at least as large as the first. Since if we add $2^3 + 5^3$ to our list of sums, there is no need to also include $5^3 + 2^3$.

Here is the procedure that finds all possible sums of pairs of integers from the given list **L** up to the specified maximum value **max**. Note that we again use the **[op({op(sums)})]** structure to turn the list into a set and then back into a list. This removes redundancies and also puts the list in increasing order.

```
> AllPairSums := proc(L, max)
    local a, b, s, sums, num;
    sums := [];
    num := nops(L);
    a := 1;
    while a <= num do
      b := a;
      while b <= num do
        s := L[a] + L[b];
```

```
        if s <= max then
          sums := [op(sums),s];
        else
          b := num;
        end if;
        b := b + 1;
      end do;
      a := a + 1;
    end do;
    return [op({op(sums)})];
  end proc:
```

With this procedure in place, we need to apply it (three times) to a list of cubes. We'll consider cubes up to 7^3, and including 0.

```
> somecubes := [seq(i^3,i=0..7)];
```

$$somecubes := [0, 1, 8, 27, 64, 125, 216, 343] \quad \textbf{(1.116)}$$

Applying the **AllPairSums** procedure once gives us all pairs of cubes.

```
> TwoCubes := AllPairSums(somecubes,343);
```

$$TwoCubes := [0, 1, 2, 8, 9, 16, 27, 28, 35, 54, 64, 65, 72, 91, 125, 126, 128, 133, 152, 189, 216, 217, 224, 243, 250, 280, 341, 343] \quad \textbf{(1.117)}$$

Applying it to that result gives all possible sums of four cubes (up to 343).

```
> FourCubes := AllPairSums(TwoCubes,343);
```

$$FourCubes := [0, 1, 2, 3, 4, 8, 9, 10, 11, 16, 17, 18, 24, 25, 27, 28, 29, 30, 32, 35, 36, 37, 43, 44, 51, 54, 55, 56, 62, 63, 64, 65, 66, 67, 70, 72, 73, 74, 80, 81, 82, 88, 89, 91, 92, 93, 99, 100, 107, 108, 118, 119, 125, 126, 127, 128, 129, 130, 133, 134, 135, 136, 137, 141, 142, 144, 145, 149, 152, 153, 154, 155, 156, 160, 161, 163, 168, 179, 180, 182, 187, 189, 190, 191, 192, 193, 197, 198, 200, 205, 206, 216, 217, 218, 219, 224, 225, 226, 232, 233, 240, 243, 244, 245, 250, 251, 252, 253, 254, 256, 258, 259, 261, 266, 270, 271, 277, 278, 280, 281, 282, 285, 288, 289, 296, 297, 304, 307, 308, 314, 315, 317, 322, 334, 341, 342, 343] \quad \textbf{(1.118)}$$

And once again we obtain all integers up to 343 which can be obtained as the sum of eight cubes.

```
> EightCubes := AllPairSums(FourCubes,343);
```

$$EightCubes := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200,$$ \quad \textbf{(1.119)}

201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343]

And finally, we print out the integers that are missing from the list.

```
> for i from 1 to 343 do
    if not (i in EightCubes) then
      print(i);
    end if;
  end do;
```

$$23$$

$$239 \qquad \textbf{(1.120)}$$

Exercises

Exercise 1. Write procedures **NOT**, **OR**, and **XOR** to implement those bit string operators.

Exercise 2. Use Maple to solve exercises 19 through 23 in Section 1.2 using the knights and knaves puzzle that was solved earlier in this chapter as a guide.

Exercise 3. Write a Maple procedure to find the dual of a proposition. Dual is defined in the Exercises of Section 1.3. (Hint: you may find it useful to know that **`and`** and **`or`** are considered types in Maple and thus can be used as the second argument to the **type** command.)

Exercise 4. Write a procedure **UniqueExists**, similar to the **Universal** and **Existential** procedures in Section 1.4 of this manual. This procedure should accept as its arguments a propositional function and a finite domain and return true if there is a unique element of the domain that satisfies the proposition and false otherwise.

Exercise 5. Write a procedure **ForAllExists** that is analogous to the **ExistsForAll** procedure given in Section 1.5 of this manual.

Exercise 6. Write a Maple procedure that plays the obligato game in the role of the student, as described in the Supplementary Exercises of Chapter 1. Specifically, the procedure should accept two arguments. The first argument is the new statement that you, as the teacher, provide. The second argument should be the list of Maple's responses to all the previous statements. For example, suppose the teacher's first statement is $p \rightarrow (q \vee r)$, the second statement is $\neg p \vee q$, and the third statement is r. If the procedure/student accepts the first statement and denies the second statement, then you would obtain the response to the third statement by executing

```
Obligato(r, [p implies (q or r), not(not p or q)]);
```

The procedure must accept the statement r and thus returns the list with this response included:

```
[p implies (q or r), not(not p or q),r]
```