1 The Foundations: Logic and Proofs

Introduction

This chapter describes how *Mathematica* 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 *Mathematica* can be used as part of proofs, specifically to find counterexamples, carry out proofs by exhaustion, and search for witnesses for existence proofs.

1.1 Propositional Logic

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

In the Wolfram Language, the truth values true and false are represented by the symbols `True` and `False`. Propositions can be represented by symbols (variables) such as `p`, `q`, or `prop1`. Note that if you have not yet made an assignment to a symbol, entering it will return the name.

```
In[1]:= prop1
Out[1]= prop1
```

Once you have assigned a value, *Mathematica* will evaluate the symbol to the assigned value whenever it appears.

```
In[2]:= prop1=True
Out[2]= True
In[3]:= prop1
Out[3]= True
```

You can cause *Mathematica* to "forget" the assigned value using either the function `Clear` or the `Unset` (`=.`) operator. Both of the expressions below have the effect of removing the assigned value from the symbol `prop1`. Neither expression returns an output.

```
In[4]:= Clear[prop1]
In[5]:= prop1=.
```

Logical Connectives

The Wolfram Language supports all of the basic logical operators discussed in the textbook. We illustrate negation (`Not`, `!`), conjunction (`And`, `&&`), disjunction (`Or`, `||`), exclusive or (`Xor`), implication (`Implies`), and the biconditional (`Equivalent`). Note that in the Wolfram Language these are referred to as Boolean operators, and expressions formed from them are Boolean expressions.

For all of the operators, you can enter expressions in standard form, that is, by putting the names of the operators at the head of an expression with truth values or other expressions as operands. For example, the computations $T \vee F$, $T \Rightarrow (F \wedge T)$, and $T \oplus T$ are shown below.

```
In[6]:= Or[True,False]
Out[6]= True
In[7]:= Implies[True,And[False,True]]
Out[7]= False
In[8]:= Xor[True,True]
Out[8]= False
```

For negation, conjunction, and disjunction, you can use the infix operators `!`, `&&`, and `||` instead. These are commonly used in place of $\neg$, $\wedge$, and $\vee$ that can be easily typed on a standard keyboard. The computations below show $\neg T$ and $(T \vee F) \wedge T$ using the operators `!`, `&&`, and `||`.

```
In[9]:= !True
Out[9]= False
In[10]:= (True||False)&&True
Out[10]= True
```

Mathematica also allows you to enter and compute with expressions using the traditional symbols. You enter the symbol by pressing the escape key, followed by a sequence identifying the symbol, and then the escape key once again. The Wolfram Language refers to this as an alias. For example, entering ESC and ESC produces the traditional symbol for conjunction.

```
In[11]:= True ∧ False
Out[11]= False
```

An alias is the only way to produce an infix implication operator, via ESC => ESC (escape, followed by equals, the greater than sign, and terminating with escape).

```
In[12]:= False ⇒ False
Out[12]= True
```

In this manual, we will typically not use aliases as part of commands, since it is less clear how to imitate such commands. However, for convenience, we include a table of the operators defined in the textbook along with their names in the Wolfram Language and their infix representations with and without aliases.

name	function	withoutalias	alias	symbol
negation	**Not**	!	ESC not ESC	¬
conjunction	**And**	&&	ESC and ESC	∧
exclusive or	**Xor**		ESC xor ESC	⊻
disjunction	**Or**	||	ESC or ESC	∨
biconditional	**Equivalent**		ESC equiv ESC	⇔
implication	**Implies**		ESC => ESC	⇒

Note that the symbol for exclusive or used by the Wolfram Language differs from that in the textbook. In addition, the order in which the operators appear in the table above is the order of precedence that the operators have in the Wolfram Language. Observe that the order of precedence of the biconditional and implication operators is the reverse of the order specified in the textbook. It is always a good idea to use parentheses liberally whenever precedence is in doubt.

Conditional Statements

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

From the perspective of formal 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 function, which adds 1 to the input value if the input is less than or equal to 5 and not otherwise.

```
In[13]:= ifExample[x_] := If[x≤5,x+1,x]
```

(To type the inequality into *Mathematica*, you type "x<=5". The graphical front end will automatically turn the key combination "<=" into ≤ unless you have set options to prevent it from doing so.) We now see that this function works as promised.

```
In[14]:= ifExample[3]
Out[14]= 4
In[15]:= ifExample[7]
Out[15]= 7
```

Because this is our first Wolfram Language function, let us spend a moment breaking down the general structure before detailing the workings of the conditional statement. First, we have the name of the function, `ifExample`. Note that symbols for built-in Wolfram Language functions typically begin with capital letters, so making a habit of naming functions you define with initial letters in lower case helps ensure that you do not accidentally try to assign to a built-in function.

Following the name of the function, we specify the arguments that will be accepted by the function enclosed in brackets. The underscore (_), referred to as `Blank`, indicates that this is a parameter and that the symbol preceding the underscore is the name that will be used to refer to the parameter.

Then comes the operator :=, the delayed assignment operator. The difference between using Set (=) and SetDelayed (:=) is that the delayed assignment ensures that *Mathematica* does not attempt to evaluate the function definition until the function is actually invoked. Generally speaking, SetDelayed (:=) should typically be used when you define a function, whereas Set (=) is appropriate for assigning values to variables.

On the right-hand side of the delayed assignment operator is the expression that determine what to do with the argument. In this case, the body of the function makes use of the If function to choose between two possible results. Note that we provided three arguments, separated by commas, to If. The first argument, x<=5, specifies the condition. *Mathematica* evaluates this expression to determine which of the branches, that is, which of the other two arguments, to execute. If the condition is true, then *Mathematica* evaluates the second argument, x+1, and this is the value of the function. This is traditionally called the "then" clause. If the condition specified in the first argument is false, then the third argument, called the "else" clause, is evaluated.

It is important to be aware of two additional variations on the If function. First, you are allowed to omit the "else" and provide only two arguments. As you can see in the example below, when the condition is false, *Mathematica* appears to return nothing. In fact, the expression returns the special symbol Null, which does not produce output.

```
In[16]:= If[3<1,5]
```

The second variation on If has four arguments. The Wolfram Language is very strict with regards to conditional statements. Specifically, it only evaluates the second argument if the result of evaluating the condition is the symbol True. Moreover, it only evaluates the third argument when the result of the condition is False. However, many expressions do not evaluate to either of these symbols. In these cases, *Mathematica* returns the If function unevaluated. For example, in the expression below, the symbol z has not been assigned a value and thus z > 5 cannot be resolved to a truth value.

```
In[17]:= If[z>5,4,11]
Out[17]= If[z>5,4,11]
```

By specifying a fourth argument, you can give explicit instructions on how to handle this situation.

```
In[18]:= If[z>5,4,11,0]
Out[18]= 0
```

This fourth argument is useful if there is some question of whether or not *Mathematica* will be able to resolve the condition into a truth value. We will typically not use the fourth argument, however, since in nearly all cases, a failure to properly evaluate the condition indicates an error in either our function definition or the input to it and providing the fourth argument will only hide such errors from us.

Evaluating Expressions

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

```
In[19]:= prop2 := !p
```

Note that we have defined the proposition `prop2` as an expression in terms of the symbol `p`, which has not been assigned. 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 *Mathematica* for the value of `prop2` as follows.

```
In[20]:= p = False
Out[20]= False
In[21]:= prop2
Out[21]= True
```

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 `Unset` (`=.`) it.

```
In[22]:= p=.
```

The better approach is to use the `ReplaceAll` operator (`/.`). This function has a variety of uses, one of which is to allow you to evaluate an expression for particular values of variables without the need to assign (and then `Clear`) values to the variables. We first demonstrate its use, and then we will explain the syntax.

```
In[23]:= prop2/.p→True
Out[23]= False
```

On the left-hand side of the `/.` operator is the expression to be evaluated. In this case, we have the symbol `prop2` on the left, which was assigned to be `!p`. On the right-hand side of the operator, we indicate the substitution to be made using the notation `a→b`, called a `Rule`, to indicate that `a` is replaced by `b`. Note that you obtain the arrow by typing a hyphen followed by the greater than symbol (`->`). The *Mathematica* front end will automatically turn that into the arrow character.

In order to substitute for more than one variable, list the substitutions as rules separated by commas and enclosed in braces. The following evaluates the proposition $p \wedge (\neg q)$ for p true and q false.

```
In[24]:= p&&(!q)/.{p→True,q→False}
Out[24]= True
```

Truth Tables and Loops

The Wolfram Language has a built-in function for producing a truth table, `BooleanTable`, which will be described in Section 1.2. While the built-in function is useful, it is worthwhile to consider how such tables can be created using more primitive programming tools. In this section, we will see how to create truth tables using only basic loop constructs.

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 the Wolfram Language). Specifically, we want to loop over the two possible truth values, true and false, so we will construct a loop over the list `{True, False}`.

In the Wolfram Language, the `Do` function is used to create a loop that executes commands for each member of a list. The `Do` function requires two arguments. The first argument is the expression that you want evaluated, typically involving one or more variables that change during the execution of the loop.

The second argument specifies the iterative behavior and can take several forms. The form we will be using here is $\{i, \{i_1, i_2, \ldots\}\}$. The character i represents the loop variable and the list $\{i_1, i_2, \ldots\}$ represents an explicit list of particular values that will be assigned to the loop variable.

The first example will be to produce a truth table for the proposition $\neg p$. Each iteration in the loop, therefore, should print out one line of the truth table. Since a `Do` loop does not produce any output unless explicitly told to do so (it normally returns `Null`), we will use the `Print` function to tell the loop what should be displayed. The `Print` function takes any number of arguments and displays them concatenated together. In this example, we want to display the value of the propositional variable p and the truth value of the proposition $\neg p$. We will explicitly insert some space between the two truth values by putting `" "` as an argument as well. Therefore, the first argument to `Do` will be `Print[p," ",!p]`.

For the second argument, the specification of the iteration, we must provide the name of the loop variable, in this case `p`, and the list of values that we want assigned to that variable in each iteration, namely true and false. Therefore, the second argument will be `{p,{True,False}}`.

```
In[25]:= Do[Print[p," ",!p],{p,{True,False}}]

True False

False True
```

As a second example, we will construct the truth table for $(p \wedge q) \Rightarrow p$. Notice that there are two variables instead of one. This indicates that two loops should be used, one for each variable. In most programming languages, the technique of "nesting" loops is the approach that you would need to take. In effect, you use a `Do` function as the first argument to another `Do` function. This approach is illustrated below.

```
In[26]:= Do[
           Do[Print[p," ",q," ",Implies[p&&q,p]],{q,{True,False}}],
           {p,{True,False}}]

True True True

True False True

False True True

False False True
```

However, there is another way. The `Do` syntax allows you to provide more than one iteration specification. For this example, we want both variables `p` and `q` to take on both truth values, so we provide the iteration specifications for both of them. *Mathematica* ensures that it executes the expression in the first argument with every possible pair of values for `p` and `q`.

```
In[27]:= Do[Print[p," ",q," ",Implies[p&&q,p]],{p,{True,False}},
           {q,{True,False}}]

True True True

True False True

False True True

False False True
```

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

Logic and Bit Operations

We can also use *Mathematica* to explore the bit operations OR, AND, and XOR. Recall that bit operations correspond to logical operators by equating 1 with true and 0 with false. The Wolfram Language provides a lot of support for working with bits and bit strings. Here, we will briefly introduce the relevant functions. Our main goal of this section, however, will be to develop a function essentially from scratch for computing with bit strings, in order to further illustrate programming in the Wolfram Language.

The Built-In Functions

The Wolfram Language provides several functions corresponding to the basic logical operations for operation on bits: `BitAnd`, `BitOr`, `BitXor`, and `BitNot`. With the exception of `BitNot`, these operations operate as you would expect. For example, you can compute $1 \wedge 0$ as follows.

```
In[28]:= BitAnd[1,0]

Out[28]= 0
```

In addition, you are not limited to two arguments. For example, computing $0 \vee 0 \vee 1 \vee 0$ requires only one application of `BitOr`.

```
In[29]:= BitOr[0,0,1,0]

Out[29]= 1
```

Conveniently, the bitwise functions are `Listable`. This means that the function is automatically threaded over lists that are given as arguments. This can be made clearer by demonstrating with another listable function: addition.

```
In[30]:= {1,2,3}+{a,b,c}

Out[30]= {1+a,2+b,3+c}
```

Because addition is listable, when it is applied to two lists of equal length, it returns the list formed by acting on corresponding elements of the lists. In the current context, this means we can apply the bitwise operations to bit strings by representing the bit strings as lists. For example, $10010 \wedge 01011$ can be computed as shown below.

```
In[31]:= BitAnd[{1,0,0,1,0},{0,1,0,1,1}]

Out[31]= {0,0,0,1,0}
```

The bitwise functions actually operate on integers, not just the bits 0 and 1. For example, we can apply `BitOr` to 18 and 5.

```
In[32]:= BitOr[18,5]

Out[32]= 23
```

The reason for this result is that *Mathematica* applied the bitwise OR to the binary representations of the integers 18 and 5. You can use the function `IntegerDigits` with an integer as the first coordinate and 2 as the second coordinate to see the binary representation of an integer.

```
In[33]:= IntegerDigits[18,2]
Out[33]= {1,0,0,1,0}
In[34]:= IntegerDigits[5,2]
Out[34]= {1,0,1}
```

We need to pad the result for 5 with 0s in order to have lists of equal size and then we can apply `BitOr` on the lists of bits as we did above.

```
In[35]:= BitOr[{1,0,0,1,0},{0,0,1,0,1}]
Out[35]= {1,0,1,1,1}
```

The `FromDigits` function reverses `IntegerDigits`. Given a list of bits and second argument 2, it will return the integer with that binary representation.

```
In[36]:= FromDigits[{1,0,1,1,1},2]
Out[36]= 23
```

Understanding the operation of `BitNot` is a bit more complicated. As expected, it accepts only one argument, although again, it will automatically thread through a list. The results on 0 and 1, however, are not what you would expect.

```
In[37]:= BitNot[0]
Out[37]= -1
In[38]:= BitNot[1]
Out[38]= -2
```

The reason for these results is that *Mathematica* represents integers in two's complement form with an unlimited number of digits. Interested readers should refer to the information prior to Exercise 46 in Section 4.2 of the textbook for an explanation of two's complement. For this context, it is enough to know that `BitNot` applied to an integer n will always return $-1 - n$, but that it will behave exactly as expected relative to the other functions. For example, $1 \wedge (\neg 0)$ results in 1, as it should.

```
In[39]:= BitAnd[1,BitNot[0]]
Out[39]= 1
```

Creating a New Bitwise And

As mentioned above, we will use the bitwise operations as an opportunity to further explore the Wolfram Language's programming capabilities and some important functions. Specifically, we will build a bitwise conjunction function that behaves much like the built-in function for bits and lists of bits.

We begin by creating a function that applies only to a pair of bits. Later, we will extend it to bit strings. We name our function `and`. Since Wolfram Language symbols are case-sensitive, this is different from the built-in function `And`.

To implement `and`, we will make use of the `Switch` function. `Switch` is an important mechanism for controlling flow in a program. It is equivalent to a series of if statements, but its structure makes it more efficient and more easily understood. `Switch` is executed in the form

`Switch[`*expr*`,` *form*$_1$`,` *value*$_1$`,` *form*$_2$`,` *value*$_2$`, ...]`

The first argument is an expression that is evaluated. The rest of the arguments are in form/value pairs. *Mathematica* checks the result of evaluating the expression against the forms, one at a time and in order. If it finds a match, then it stops checking and returns the value associated with the matching form. If none of the forms match, then the result is the `Switch` function unevaluated.

Our `and` function will accept two arguments. The expression we give to `Switch` will be the list formed from the two arguments. The rest of the `Switch` will essentially be the truth table for conjunction. The forms will be all the possible pairs of 0s and 1s, and the values will be 0 or 1 as appropriate.

```
In[40]:= and[p_,q_]:=Switch[{p,q},{1,1},1,{1,0},0,{0,1},0,{0,0},0]
```

The `and` function we created now works as expected on bits and does nothing if it is given other input.

```
In[41]:= and[1,1]
Out[41]= 1

In[42]:= and[1,0]
Out[42]= 0

In[43]:= and[18,5]
Out[43]= Switch[{18,5},
           {1,1},1,
           {1,0},0,
           {0,1},0,
           {0,0},0]
```

We can handle nonbit input a bit more elegantly by adding one more form/value pair. Using a `Blank` (`_`) for the form will create a default value. By creating a message associated to the `and` function, we can display a useful error message, as illustrated below. The message is defined by setting the symbol *f*::tag equal to the message in quotation marks, where *f* is the name of the function and tag is the "name" of the message. When this symbol is given as the argument to the `Message` function, the message is shown.

```
In[44]:= and::arg="and called with non-bit arguments.";

In[45]:= and[p_,q_]:=Switch[{p,q},{1,1},1,{1,0},0,{0,1},0,
           {0,0},0,_,Message[and::arg]]
```

Now, applying and to 18 and 5 produces a more useful result.

```
In[46]:= and[18,5]
... and: and called with non-bit arguments.
```

Threading and Listable

We saw above that the Wolfram Language's built-in function extends to lists of integers without any additional effort on our part. Here, we will see that it is easy to make our function do that as well.

The Wolfram Language provides `Map` (`/@`) and `MapThread` to facilitate the creation of functions that act on elements of lists. We describe `Map` first.

Given a function of one argument, such as $f(x) = x^2$, `Map` causes *Mathematica* to apply the function to all the elements of a list. First, define the function.

```
In[47]:= f[x_] := x^2
```

Now, call `Map` with the name of the function as the first argument and the list of input values as the second.

```
In[48]:= Map[f,{1,2,3,4,5,6}]
Out[48]= {1,4,9,16,25,36}
```

The result, as you see above, is the list of the results of applying the function to each element of the list. The same result can be obtained with the `/@` operator, as shown below.

```
In[49]:= f/@{1,2,3,4,5,6}
Out[49]= {1,4,9,16,25,36}
```

When a function has more than one argument, as `and` does, `MapThread` can be used. Like `Map`, `MapThread` takes two arguments with the first being a function. The second argument is a list of lists. Provided that each of the inner lists is of the same length, the result of `MapThread` is the list formed by evaluating the function on tuples of arguments from corresponding positions in the lists. For example, we can apply $g(x, y) = x^2 + y^3$ to $\{1, 2, 3\}$ and $\{a, b, c\}$ in order to obtain $\{g(1, a), g(2, b), g(3, c)\}$.

```
In[50]:= g[x_,y_]:=x^2+y^3
In[51]:= MapThread[g,{{1,2,3},{a,b,c}}]
Out[51]= {1+a^3,4+b^3,9+c^3}
```

Using `MapThread`, we can compute $10010 \wedge 01011$ as illustrated below.

```
In[52]:= MapThread[and,{{1,0,0,1,0},{0,1,0,1,1}}]
Out[52]= {0,0,0,1,0}
```

This shows how to thread a function in a particular case. However, what we really want is for our `and` function to behave like this automatically. Fortunately, this is such a common requirement for functions, that the Wolfram Language provides a very easy way to do this. The attribute `Listable` indicates that the function should be automatically threaded over lists whenever the function is given a list as its argument. The `SetAttributes` function causes *Mathematica* to associate the attribute specified in the second argument with the symbol in the first argument.

```
In[53]:= SetAttributes[and,Listable]
```

Now, applying `and` to lists works just as the built-in `BitAnd` does.

```
In[54]:= and[{1,0,0,1,0},{0,1,0,1,1}]
Out[54]= {0,0,0,1,0}
```

1.2 Applications of Propositional Logic

In this section, we will describe how *Mathematica*'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 that 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, and $r =$ false. We can confirm this by evaluating the list of propositions with that assignment of truth values.

Above, we saw that you can evaluate an expression using the replacement operator `/.`. On the left side of the replacement operator, we put the expression we want evaluated, in this case a list of the three logical propositions. On the right side of the `/.`, we enter the assignments as a list of rules of the form `s -> v` for symbol `s` and value `v`.

```
In[55]:= {Implies[p,q&&r],p||q,p||(!r)}/.
           {p→False,q→True,r→False}

Out[55]= {True,True,True}
```

To determine if a collection of propositions is consistent, we can create a truth table. In the previous section, we created truth tables from scratch using the `Do` function to loop through all possible assignments of truth values to the variables. In this section, we will instead use the Wolfram Langauge built-in function `BooleanTable`.

The `BooleanTable` function produces the truth values obtained by replacing the variables by all possible combinations of true and false. Its first argument is the expression to be evaluated and the second argument is a list of the propositional variables.

```
In[56]:= BooleanTable[p&&(!q),{p,q}]

Out[56]= {False,True,False,False}
```

Note that, unlike a truth table you construct by hand, `BooleanTable` does not show the assignments to the propositional variables. We can see the values of the propositional variables by making the first argument a list that includes them.

```
In[57]:= BooleanTable[{p,q,p&&(!q)},{p,q}]

Out[57]= {{True,True,False},{True,False,True},{False,True,False},
           {False,False,False}}
```

The TableForm function will make the output easier to read. We will apply TableForm with the postfix operator (//). The postfix operator allows you to put the name of a function after an expression. It is commonly used for functions that affect the display of a result and has the benefit of making the main part of the command being evaluated easier to read.

```
In[58]:= BooleanTable[{p,q,p&&(!q)},{p,q}]//TableForm
```

```
Out[58]//TableForm=
True  True   False
True  False  True
False True   False
False False  False
```

Returning to the question of consistency, consider Example 4 from Section 1.2 of the text. We translate the three specifications into list of propositions that follows.

```
In[59]:= specEx4={p||q,!p,Implies[p,q]}

Out[59]= {p||q,!p,p⇒q}
```

Then we can construct the truth table using BooleanTable.

```
In[60]:= BooleanTable[{p,q,specEx4},{p,q}]//TableForm
```

```
Out[60]//TableForm=
                    True
True  True          False
                    True

                    True
True  False         False
                    False

                    True
False True          True
                    True

                    False
False False         True
                    True
```

Notice that, because specEx4 is itself a list, TableForm displays the results from the three component propositions as a column within the row corresponding to the values for p and q. 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: $(p \vee q) \wedge (\neg p) \wedge (p \rightarrow q)$.

```
In[61]:= specEx4b=And[(p||q),!p,Implies[p,q]]

Out[61]= (p||q)&&!p&&(p⇒q)

In[62]:= BooleanTable[{p,q,specEx4b},{p,q}]//TableForm

Out[62]//TableForm=
True  True   False
True  False  False
False True   True
False False  False
```

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

The Wolfram Language also has useful built-in functions for checking for consistency. The `SatisfiableQ` function accepts the same arguments as `BooleanTable` (a Boolean expression and the list of propositional variables). Note that you may not give a list of expressions as the first argument to `SatisfiableQ`.

```
In[63]:= SatisfiableQ[specEx4b,{p,q}]

Out[63]= True
```

The `SatisfiabilityInstances` command will generate an assignment of truth values to the variables that do in fact satisfy the proposition, assuming it is satisfiable.

```
In[64]:= SatisfiabilityInstances[specEx4b,{p,q}]

Out[64]= {{False,True}}
```

By providing a positive integer as an optional third argument, you can ask for more choices that make the proposition true. Below, we find all three ways that $p \rightarrow q$ can be satisfied.

```
In[65]:= SatisfiabilityInstances[Implies[p,q],{p,q},3]

Out[65]= {{True,True},{False,True},{False,False}}
```

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

```
In[66]:= specEx5=specEx4b && !q

Out[66]= (p||q)&&!p&&(p⇒q)&&!q
```

```
In[67]:= BooleanTable[{p,q,specEx5},{p,q}]//TableForm

Out[67]//TableForm=
True  True   False
True  False  False
False True   False
False False  False
```

In addition, note that `SatisfiableQ` returns false and `SatisfiabilityInstances` returns an empty list.

```
In[68]:= SatisfiableQ[specEx5,{p,q}]

Out[68]= False

In[69]:= SatisfiabilityInstances[specEx5,{p,q}]

Out[69]= {}
```

Logic Puzzles

Recall the knights and knaves puzzle presented in Example 8 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 which kind of inhabitants A and B are.

We can solve this problem using truth tables. First, we must write A's and B's statements as propositions. Let a represent the statement that A is a knight and b represent 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's 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 "b" is false. In other words, the truth value of the proposition a, that is, 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 function `Equivalent`, we can express the two statements in the puzzle.

```
In[70]:= ex7a=Equivalent[a,b]

Out[70]= a⇔b

In[71]:= ex7b=Equivalent[b,(a&&!b)||(!a&&b)]

Out[71]= b⇔(a&&!b)||(!a&&b)
```

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.

```
In[72]:= SatisfiabilityInstances[ex7a&&ex7b,{a,b},4]

Out[72]= {{False,False}}
```

We see that both statements are satisfied when both propositions a and b are false, that is, when A and B are both knaves. Note also that since we asked, in the final argument, for as many as four different instances but only one was returned, we know that this is the only solution to the puzzle.

1.3 Propositional Equivalence

In this section, we consider logical equivalence of propositions. We will first look at the built-in functions for testing equivalence, and then we will create a function from scratch to accomplish the same goal. Additionally, we will use *Mathematica* to solve the n-Queens problem as a satisfiability problem.

Built-In Functions

Two propositions p and q are logically equivalent if the proposition $p \leftrightarrow q$ is a tautology. The Wolfram Language includes a function for checking whether a proposition is a tautology, `TautologyQ`. This function uses the same arguments as `BooleanTable`, `SatisfiableQ`, and `SatisfiabilityInstances` do, as described above. Specifically, the first argument should be the proposition and the second argument should be a list of the propositional variables.

For example, we confirm that the DeMorgan's law $\neg(p \wedge q) \equiv \neg p \vee \neg q$ is a propositional equivalence below.

```
In[73]:= TautologyQ[Equivalent[!(p&&q),!p||!q],{p,q}]
Out[73]= True
```

Remember that the `Equivalent` function, used above, is the Wolfram Language function for forming the biconditional and should not be confused with the notion of equivalence as used in Section 1.3 of the textbook.

Note that the second argument to `TautologyQ` is not generally necessary. The `BooleanVariables` function, which determines the variables in a logical expression, will invisibly supply the missing argument. This is, in fact, true about most of the functions that require the variable list as the second argument. We demonstrate with the other DeMorgan's law.

```
In[74]:= TautologyQ[Equivalent[!(p||q),!p&&!q]]
Out[74]= True
```

You might find it convenient to have a single function that, given two propositions, will determine whether they are logically equivalent. In the Wolfram Language, this is easy to achieve. We just need to create a function that takes two propositions, uses the `Equivalent` function to create the biconditional, and then applies `TautologyQ`.

```
In[75]:= equivalentQ[p_,q_]:=TautologyQ[Equivalent[p,q]]
```

We apply this function to see if we can generalize DeMorgan's laws to three variables.

```
In[76]:= equivalentQ[!(p||q||r),!p&&!q&&!r]
Out[76]= True
In[77]:= equivalentQ[!(p&&q&&r),!p||!q||!r]
Out[77]= True
```

Built from Scratch Function

The Wolfram Language provides extensive built-in support for working with logical propositions and, in particular, checking propositional equivalence. Here, however, we are going to build a new function for checking whether or not two propositions are logically equivalent using a minimum of high-level functions. In fact, other than asking *Mathematica* to evaluate propositional expressions for particular truth values assigned to propositional variables, we will make use only of the Wolfram Language's essential programming functionality.

There are two goals here. First, to illustrate more of the Wolfram Language's programming abilities. Second, to reveal some of the more fundamental concepts and methods used in *Mathematica*.

We will create a function `myEquivalentQ` that has the same effect as the `equivalentQ` function that we built above using the Wolfram Language's built-in symbols. Specifically, our function should take two propositions and determine whether or not they are equivalent. This will require quite a bit of work. The main hurdles for such a function are (1) having *Mathematica* determine what propositional variables are used in the input propositions and (2) without *a priori* knowledge of the number of propositional variables, having *Mathematica* 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 symbols, perhaps p, q, r, and s. Then, we could implement the function using a static nested `Do` loop.

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 *Mathematica* to determine the variables used in a logical expression. Consider the following example.

```
In[78]:= variableEx=((p&&q)||(p&&!r))&&Implies[s,r]
Out[78]= ((p&&q)||(p&&!r))&&(s⇒r)
```

Our task is to write a function that will, given the above expression, tell us that the variables in use are p, q, r, and s.

Replacing the Head

Fundamentally, everything in the Wolfram Language is an expression. Moreover, every expression is of the form $head[arg_1, arg_2, \ldots]$, that is, a head followed by arguments in brackets and separated by commas. You can see the structure at the heart of any expression by using the `FullForm` function. Below, we show the full form of three examples. Recall that the postfix operator (`//`) allows us to put the name of the function at the end of the input.

```
In[79]:= x+y//FullForm
Out[79]//FullForm=
         Plus[x,y]

In[80]:= variableEx//FullForm
Out[80]//FullForm=
         And[Or[And[p,q],And[p,Not[r]]],Implies[s,r]]
```

```
In[81]:= {p,q,r,s}//FullForm
Out[81]//FullForm=
List[p,q,r,s]
```

The Wolfram Language provides a function, `Head`, that takes an expression and returns the head of that expression.

```
In[82]:= Head[x+y]
Out[82]= Plus
In[83]:= Head[variableEx]
Out[83]= And
In[84]:= Head[{p,q,r,s}]
Out[84]= List
```

You can also access the head of an expression using the `Part` (`[[...]]`) operator with index 0.

```
In[85]:= (x+y)[[0]]
Out[85]= Plus
In[86]:= variableEx[[0]]
Out[86]= And
In[87]:= {p,q,r,s}[[0]]
Out[87]= List
```

Remember that our goal here is to transform a logical expression, such as $((p \wedge q) \vee (p \wedge \neg r)) \wedge (s \rightarrow r)$ into a list $\{p, q, r, s\}$. Since the main difference, in terms of the internal representation of the two objects, is their heads, it is natural to ask if we can change the head. In particular, in our example `variableEx`, the head is `And`. If we can replace the `And` head with a `List` head, we would have a list comprised of the two parts of the expression, as illustrated below.

```
In[88]:= List[Or[And[p,q],And[p,Not[r]]],Implies[s,r]]
Out[88]= {(p&&q)||(p&&!r),s⇒r}
```

Our strategy, broadly, will be to replace all of the heads in the logical expression with `List` heads. There are two approaches to replacing the head of an expression. One is to use the fact that the head lies at index 0 to replace the heads by assigning the 0 indexed element to `List` using the syntax `x[[0]]=List`, as illustrated below.

```
In[89]:= sumExample=x+y
Out[89]= x+y
In[90]:= sumExample[[0]]=List
Out[90]= List
```

```
In[91]:= sumExample

Out[91]= {x,y}
```

The second approach is to use the `Apply` (`@@`) operator. An expression formed from the desired head, followed by two at symbols and the original expression will output the expression with the new head. Unlike the previous approach, if the expression is stored as a symbol, the stored expression is not changed, unless you explicitly reassign the output to the symbol. We illustrate by transforming `sumExample` from a list into a product.

```
In[92]:= sumExample=Times@@sumExample

Out[92]= x y
```

Note that the `Head` command gives us a way to test what kind of expression we have. In particular, we can differentiate between variables, which have head `Symbol`, and other expressions. To compare heads, you must use the `SameQ` relation (===) rather than `Equal` (==), which only applies to raw data (such as numerical values and strings).

```
In[93]:= If[Head[x+y]===Head[x-y],
            Print["+ equals -"],
            Print["different"]]

         + equals -
```

The above shows that the heads of $x + y$ and $x - y$ are in fact the same. Both expressions have head `Plus`. (We could also do this with the `[[0]]` syntax, but the `Head` function is clearer.)

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 replacing the heads of the subexpressions until we are down to nothing but names. The strategy we will use is a fairly typical one. We illustrate the approach step by step with the `variableEx` example first, and then we will build a function.

First, we define a new symbol, `variableExList`, to be the result of applying (`Apply`, `@@`) the `List` head to the `variableEx` expression. Note that this does not change the expression stored in `variableEx`. We wish to preserve `variableEx`, which is why we take this approach. Moving forward, we will use the `Part` (`[[...]]`) approach.

```
In[94]:= variableExList=List@@variableEx

Out[94]= {(p&&q)||(p&&!r),s⇒r}
```

Observe that the topmost conjunction has been removed and we now have a list of the two subexpressions.

Now, we need to do the same thing to the elements of this list. Remember that the `Part` function (`[[...]]`) is used to obtain and to modify elements of a list, so we can obtain the first element in the list as follows.

```
In[95]:= variableExList[[1]]

Out[95]= (p&&q)||(p&&!r)
```

We can turn this into a list by assigning the 0-indexed element of `variableExList[[1]]` to `List`.

```
In[96]:= variableExList[[1]][[0]]=List
Out[96]= List
```

Inspecting `variableExList`, we see that this has replaced what was the first element with the new result.

```
In[97]:= variableExList
Out[97]= {{p&&q,p&&!r},s⇒r}
```

You can see that we have already made quite a bit of progress, but now we have lists nested together. We can eliminate this nesting with the `Flatten` function. We assign the result of applying the function to `variableExList` back to `variableExList`, so the result is kept.

```
In[98]:= variableExList=Flatten[variableExList]
Out[98]= {p&&q,p&&!r,s⇒r}
```

The first element of `variableExList` is still a logical expression, so we repeat. This time, we will use `[[1,0]]`, which is shorthand for `[[1]][[0]]`. We also combine the assignment and the inspection of `variableExList` into one input.

```
In[99]:= variableExList[[1,0]]=List;
         variableExList
Out[100]= {{p,q},p&&!r,s⇒r}
```

Again, we use `Flatten` since this has created a nested list structure.

```
In[101]:= variableExList=Flatten[variableExList]
Out[101]= {p,q,p&&!r,s⇒r}
```

The first two elements of `variableExList` are now symbols. Thus, we skip to the third element. Again, we change the head of the third element to the `List` head.

```
In[102]:= variableExList[[3,0]]=List;
          variableExList
Out[103]= {p,q,{p,!r},s⇒r}
```

And again, flatten the resulting list.

```
In[104]:= variableExList = Flatten[variableExList]
Out[104]= {p,q,p,!r,s⇒r}
```

Now that the third element is a symbol, we do the same thing with the fourth element of `variableExList`. We also include the `Flatten` step in the same input.

```
In[105]:= variableExList[[4,0]]=List;
          variableExList=Flatten[variableExList]

Out[106]= {p,q,p,r,s⇒r}
```

And we repeat the process once more.

```
In[107]:= variableExList[[5,0]]=List;
          variableExList=Flatten[variableExList]

Out[108]= {p,q,p,r,s,r}
```

Now that every element in the list is a variable, we remove the duplicate elements with `DeleteDuplicates`.

```
In[109]:= variableExList=DeleteDuplicates[variableExList]

Out[109]= {p,q,r,s}
```

The Function

The explicit example above gives us the outline of our procedure:

1. Initialize a list, `varList`, to the list with the given proposition as the sole element. We did not do this in the example, but doing so means that we will always be working with a list, rather than having the first step be different.
2. We also initialize an index variable, `i`, to 1. This will keep track of where we are in the list, taking the place of the explicit value 5, for example, in the third to last line of code above.
3. Use `Head` to test whether the element in position `i` in the list is a `Symbol`.
   - If it is a symbol, then it is the name of a variable, and we move on to the next position in the list by increasing `i` by 1.
   - If `varList[[i]]` is not a symbol, then it must be an expression. Replace its head with `List` and flatten `varList`, using the same syntax as above.
4. Repeat step 3 until the end of the list. This repetition is controlled by a `While` loop that continues as long as `i` is not greater than the number of elements in the list, determined by `Length`. Once the loop is complete, remove duplicate entries.

Here is the implementation.

```
In[110]:= getVars[p_]:=Module[{L={p},i=1},
            While[i<=Length[L],
              If[Head[L[[i]]]===Symbol,
                i++,
                L[[i,0]]=List;
                L=Flatten[L]
              ]
            ];
            DeleteDuplicates[L]
          ]
```

The use of `Module` requires explanation. The purpose of `Module` is to encapsulate the variables used within a function so that they do not change the values of variables used outside of the function. For

example, if you set L equal to some value before executing getVars, it will still have that value afterward. Likewise, Module prevents values set outside the function from impacting the behavior of the function. In the jargon of programming languages, Module ensures that the specified variables are treated as local to the module, or that they have a local scope, as distinguished from global.

The Module function takes two arguments. The first is the list of variables to be held local. Within the list of variables, either you can provide just the name of the variable or, if you wish, you can assign the initial value of the variable, as was done in getVars. The expression {L={p}, i=1} as the first argument to Module means that the symbols L and i are local and that they are initially assigned values {p} and 1, respectively.

The second argument to Module is the body of the function definition. Note that semicolons are used to separate functions when there is more than one within the body. For example, in the third to last line of getVars, the semicolon separates the conclusion of the While loop from the application of DeleteDuplicates.

Finally, observe that the function getVars works as expected.

```
In[111]:= getVars[variableEx]

Out[111]= {p,q,r,s}

In[112]:= getVars[Implies[!w,Equivalent[Q||q,P&&p]]]

Out[112]= {w,P,p,Q,q}
```

Truth Value Assignments

The second hurdle that we mentioned at the beginning of this section is that we do not know the number of propositional variables in advance. If we knew there would always be two variables, we would use two nested for loops. However, since we want our procedure to work with any number of variables, we need a different approach.

Since our getVars function 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,

```
In[113]:= variableExVars=getVars[variableEx]

Out[113]= {p,q,r,s}

In[114]:= truthValEx={True,True,False,True}

Out[114]= {True,True,False,True}
```

We consider the truthValEx (for truth values example) to indicate that we assign the first variable of variableExVars to the value true, the second variable to true, the third variable to false, and the fourth variable to true.

Evaluating an Expression

Recall the use of the ReplaceAll operator (/.) to evaluate an expression. In particular, this operator requires that the second operand is a list of rules of the form $s \to v$ with s a symbol and v a value. For example, the following evaluates variableEx at the values p = true, q = true, r = false, and s = true.

```
In[115]:= variableEx/.{p→True,q→True,r→False,s→True}

Out[115]= False
```

In order to perform that evaluation programmatically, using the result of `getVars` and a list representing an assignment of truth values, we need to turn the pair of lists into a list of rules. We will demonstrate how to do this with the `variableExVars` and `truthValEx` lists defined above.

We introduced the `MapThread` function at the end of Section 1.1 of this manual. Recall that the basic purpose of `MapThread` is to take a function of *n* variables together with a list of *n* lists (with the sublists having the same size) and apply the function to corresponding elements of the lists. For example, we can use `MapThread` to add corresponding elements of two lists using the `Plus` function. (Note that this is generally unnecessary since addition automatically threads, but it serves as an example.)

```
In[116]:= MapThread[Plus,{{1,2,3},{a,b,c}}]
Out[116]= {1+a,2+b,3+c}
```

In our context, the two lists are the lists of variables, `variableExVars`, and the truth value assignment, `truthValEx`. The function that forms a rule is `Rule`.

```
In[117]:= MapThread[Rule,{variableExVars,truthValEx}]
Out[117]= {p→True,q→True,r→False,s→True}
```

We can thus evaluate the expression with the following.

```
In[118]:= variableEx/.MapThread[Rule,{variableExVars,truthValEx}]
Out[118]= False
```

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 will use a strategy similar to binary counting. Start with the list all of whose elements are the symbol false. 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` function (for next truth assignment). The `nextTA` function 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 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 function 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 the list all of whose elements were the symbol true, which is the last possible truth assignment, and the function returns `Null` to indicate that there is no next truth assignment.

```
In[119]:= nextTA[A_]:=Module[{i,newTA=A},
            Catch[
              For[i=1,i<=Length[A],i++,
                If[newTA[[i]],
                  newTA[[i]]=False,
```

```
            newTA[[i]]=True;Throw[newTA]
          ]
        ];
        Throw[Null]
      ]
    ]
```

Once again, we use a `Module` structure. This ensures that `i`, the loop variable, and `newTA`, the truth assignment that is being constructed, are private to the function. Note that `newTA` is initialized to be a copy of `A`, the input list. We will describe `Catch` momentarily.

The `For` function is the Wolfram Language's implementation of a for loop. The first argument contains the initialization command, in this case setting the loop variable `i` equal to 1. The second argument defines the test that specifies the termination conditions of the loop. In `nextTA`, the loop is to run through all of the entries in the list representing the truth assignment, so the test is that the value of the index `i` has not exceeded the number of entries in the list, determined by the `Length` function. The third argument to `For` is the increment specification. In this case, we used the `Increment` (`++`) operator, which increases the value of `i` by 1. It has the same effect as `i=i+1`.

The final argument to `For` is the body of the loop. The basic strategy is to work our way through the "old" truth value assignment turning true into false until we hit a false. That first false is changed to true and we stop. The body of our for loop is dominated by an `If` statement. The first argument of the `If` statement accesses the value in the current position of `newTA`. In case that value is true, according to our strategy, we change it to false and move on to the next element in the list. If the current value is false, we change it to true.

Once a false has been changed to true, we want to stop the evaluation of the function and have the current value of `newTA` returned as the output of the function. This is the purpose of `Catch` and `Throw`. The `Throw` function is a way for you to tell *Mathematica*, "This (the argument) is the result of this section of code." `Catch` defines the scope of the `Throw`, that is, the argument of the `Catch` is the block of code to which `Throw` refers. In other words, when *Mathematica* encounters a `Throw`, it evaluates its argument and considers that result to be the result of the entire `Catch` block. In this case, when the loop encounters a false entry in `newTA`, it changes that entry to true and then executes the `Throw`, which has the effect of ending any further evaluation and declaring the result to be the current value of `newTA`. Should all of the entries be true initially, then the `Throw[newTA]` will never be encountered and the loop will be allowed to complete. Once the loop is complete, the `Throw[Null]` statement will be encountered, causing the `Catch`, and thus the module, to return `Null`.

You may be wondering why we did not use a `Return` statement in the above. While the Wolfram Language does have a `Return` function, the Wolfram Language has a functional style, as opposed to procedural. Because of this, the behavior of `Return` can be unexpected. In fact, it is impossible for `Return` to have the same behavior in a functional language such as the Wolfram Language as it would in a procedural language like C. More than this, `Return` in the Wolfram Language is a bit of a square peg in a round hole situation—it does not fit with the conceptual framework of a functional language.

We can confirm, in the case of three variables, that `nextTA` does in fact produce all of the possible truth value assignments using the following simple `While` loop. Note that the `While` function executes the second argument so long as the first argument is true. Also note the use of `=!=` in the test. This is the `UnsameQ` (`=!=`) relation, which is the negation of `SameQ` (`===`), which was discussed earlier.

```
In[120]:= nextTAdemo={False,False,False};
          While[nextTAdemo=!=Null,
            Print[nextTAdemo];
            nextTAdemo=nextTA[nextTAdemo]
          ]

          {False,False,False}

          {True,False,False}

          {False,True,False}

          {True,True,False}

          {False,False,True}

          {True,False,True}

          {False,True,True}

          {True,True,True}
```

Logical Equivalence Implementation

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

The function proceeds as follows:

1. First, we create the biconditional, which we name `bicond`, that asserts the equivalence of the two propositions. We use the `getVars` function to determine the list of variables used in the propositions and we initialize the truth assignment variable `TA` to the appropriately sized list of all false values using the `ConstantArray` function applied to the value `False` and the desired length of the list.
2. 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 throw `False`. Otherwise, we use `nextTA` to update `TA` to the next truth assignment.
3. 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.

```
In[122]:= myEquivalentQ[p_,q_]:=Module[{bicond,vars,numVars,TA,val},
            bicond=Equivalent[p,q];
            vars=getVars[bicond];
            numVars=Length[vars];
            TA=ConstantArray[False,numVars];
            Catch[
              While[TA=!=Null,
                val=bicond/.MapThread[Rule,{vars,TA}];
```

```
        If[!val,Throw[False]];
        TA=nextTA[TA]
      ];
      Throw[True]
    ]
  ]
```

We can use our function 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.

```
In[123]:= myEquivalentQ[!(p||(!p&&q)),!p&&!q]
Out[123]= True
```

The *n*-Queens Problem

The textbook describes the *n*-Queens problem and shows how propositions can be formed to solve it via propositional satisfiability. With the Wolfram Language's `SatisfiableQ` function, the main challenge is building the propositions. This will give us an opportunity to see how we can write functions that, instead of computing a value, actually create objects that can be used in other functions. In particular, we will define functions `nQueens1` through `nQueens5` whose outputs are the propositions Q_1 through Q_5, as defined in the main text.

First, we will need to form the basic propositions, those the main text refers to as $p(i,j)$. It might be surprising, but the Wolfram Language will allow us to use expressions such as `p[2,3]` as propositional variables. However, we need to be careful when using the built-in satisfiability functions to include the second argument, the list of propositional variables, so that *Mathematica* understands we mean them to be propositional variables.

We can use `Table` to create the list of all the needed propositions.

```
In[124]:= Table[p[i,j],{i,1,4},{j,1,4}]
Out[124]= {{p[1,1],p[1,2],p[1,3],p[1,4]},
           {p[2,1],p[2,2],p[2,3],p[2,4]},
           {p[3,1],p[3,2],p[3,3],p[3,4]},
           {p[4,1],p[4,2],p[4,3],p[4,4]}}
```

Since that produces a nested list, we will apply `Flatten` to obtain the list of propositional variables.

```
In[125]:= Flatten[Table[p[i,j],{i,1,4},{j,1,4}]]
Out[125]= {p[1,1],p[1,2],p[1,3],p[1,4],p[2,1],p[2,2],p[2,3],p[2,4],
           p[3,1],p[3,2],p[3,3],p[3,4],p[4,1],p[4,2],p[4,3],p[4,4]}
```

Now we will form the proposition $Q_1 = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} p(i,j)$. Initially, consider just the inner disjunction, $\bigvee_{j=1}^{n} p(i,j)$. We can produce this disjunction by using `Table` to create a list of the propositions and then transforming the list into a disjunction using the `Apply` (`@@`) operator. Recall that `Apply` replaces the head of an

expression and thus can change one kind of expression into another. For example, for $n = 7$, the inner disjunction is formed as shown below.

```
In[126]:= Or@@Table[p[i,j],{j,1,7}]

Out[126]= p[i,1]||p[i,2]||p[i,3]||p[i,4]||p[i,5]||p[i,6]||p[i,7]
```

To complete Q_1, we simply make that the first argument of `Table` and apply `And`.

```
In[127]:= And@@Table[
            Or@@Table[p[i,j],{j,1,7}],
            {i,1,7}]

Out[127]= (p[1,1]||p[1,2]||p[1,3]||p[1,4]||p[1,5]||p[1,6]||p[1,7])
          &&(p[2,1]||p[2,2]||p[2,3]||p[2,4]||p[2,5]||p[2,6]||p[2,7])
          &&(p[3,1]||p[3,2]||p[3,3]||p[3,4]||p[3,5]||p[3,6]||p[3,7])
          &&(p[4,1]||p[4,2]||p[4,3]||p[4,4]||p[4,5]||p[4,6]||p[4,7])
          &&(p[5,1]||p[5,2]||p[5,3]||p[5,4]||p[5,5]||p[5,6]||p[5,7])
          &&(p[6,1]||p[6,2]||p[6,3]||p[6,4]||p[6,5]||p[6,6]||p[6,7])
          &&(p[7,1]||p[7,2]||p[7,3]||p[7,4]||p[7,5]||p[7,6]||p[7,7])
```

We generalize this process into a function by replacing the specific value 7 with a variable.

```
In[128]:= nQueens1[n_]:=And@@Table[
            Or@@Table[p[i,j],{j,1,n}],
            {i,1,n}]
```

Other than needing the `Min` function for Q_4 and Q_5, the other propositions can be formed similarly. The definitions of the propositions are displayed below, followed by the definitions of the functions.

$$Q_2 = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n-1} \bigwedge_{k=j+1}^{n} (\neg p(i,j) \vee \neg p(i,k))$$

$$Q_3 = \bigwedge_{j=1}^{n} \bigwedge_{i=1}^{n-1} \bigwedge_{k=i+1}^{n} (\neg p(i,j) \vee \neg p(k,j))$$

$$Q_4 = \bigwedge_{i=2}^{n} \bigwedge_{j=1}^{n-1} \bigwedge_{k=1}^{\min(i-1,n-j)} (\neg p(i,j) \vee \neg p(i-k,k+j))$$

$$Q_5 = \bigwedge_{i=1}^{n-1} \bigwedge_{j=1}^{n-1} \bigwedge_{k=1}^{\min(n-i,n-j)} (\neg p(i,j) \vee \neg p(i+k,j+k))$$

```
In[129]:= nQueens2[n_]:=And@@Table[
            And@@Table[
              And@@Table[!p[i,j]||!p[i,k],{k,j+1,n}],
              {j,1,n-1}],
            {i,1,n}];
          nQueens3[n_]:=And@@Table[
            And@@Table[
```

```
    And@@Table[!p[i,j]||!p[k,j],{k,i+1,n}],
    {i,1,n-1}],
  {j,1,n}];
nQueens4[n_]:=And@@Table[
  And@@Table[
    And@@Table[!p[i,j]||!p[i-k,k+j],{k,1,Min[i-1,n-j]}],
    {j,1,n-1}],
  {i,2,n}];
nQueens5[n_]:= And@@Table[
  And@@Table[
    And@@Table[!p[i,j]||!p[i+k,j+k],{k,1,Min[n-i,n-j]}],
    {j,1,n-1}],
  {i,1,n-1}]
```

Finally, we define a function that combines the propositions and produces a solution. Since `SatisfiabilityInstances` produces solutions as flat lists, we will use `Partition` to transform the output into a list of lists, with the inner lists representing the rows of the board. In its simplest form, `Partition` takes two arguments: a list and the desired length of the sublist. For example, the list of integers from 1 through 12 is broken into four sublists of length 3 below.

In[133]:=
```
Partition[Range[12],3]
```

Out[133]=
```
{{1,2,3},{4,5,6},{7,8,9},{10,11,12}}
```

Additionally, the `Grid` function can be used to display a list of lists as an array.

In[134]:=
```
Partition[Range[12],3]//Grid
```

Out[134]=
```
1     2     3
4     5     6
7     8     9
10    11    12
```

Here, then, is the function `nQueens` which produces a solution to the n-Queens problem. Recall that the output of the function `SatisfiabilityInstances` is a list of solutions. Thus, we need to apply `Part` (`[[...]]`) to access the first solution.

In[135]:=
```
nQueens[n_]:=
  Grid[
    Partition[
      SatisfiabilityInstances[
        nQueens1[n]&&nQueens2[n]&&nQueens3[n]&&
          nQueens4[n]&&nQueens5[n],
        Flatten@Table[p[i,j],{i,1,n},{j,1,n}]][[1]],n]]
```

Applying this function to $n = 8$ produces a solution to the 8-Queens problem different from the one shown in the main text.

```
In[136]:= nQueens[8]

Out[136]= False False  True   False  False  False  False  False
          False False  False  False  True   False  False  False
          False True   False  False  False  False  False  False
          False False  False  False  False  False  False  True
          False False  False  False  False  True   False  False
          False False  False  True   False  False  False  False
          False False  False  False  False  False  True   False
          True  False  False  False  False  False  False  False
```

1.4 Predicates and Quantifiers

In this section, we will see how *Mathematica* can be used to explore propositional functions and quantification. We can think about a propositional function P as a function 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 can create a function in the Wolfram Language, say `gt0` (for greater than 0), that takes x as input and returns true or false as appropriate. All we have to do is to assign the inequality as the body of the function.

```
In[137]:= gt0[x_]:=x>0
```

Evaluating the propositional function at different values demonstrates that the result is a truth value.

```
In[138]:= gt0[5]

Out[138]= True

In[139]:= gt0[-3]

Out[139]= False
```

Representation of Quantifiers

The Wolfram Language represents quantification using the functions `ForAll` and `Exists`. These functions have the same syntax. In their most basic form, they take two arguments. The first argument is the variable being bound by the quantifier, and the second is the expression being quantified. For example, to represent the statement $\forall_x P(x)$, you would enter the following.

```
In[140]:= ForAll[x,P[x]]

Out[140]= ∀x P[x]
```

Likewise, we can represent the assertion that there exists an x for which the opposite is negative as follows.

```
In[141]:= Exists[x,-x<0]

Out[141]= ∃x -x<0
```

The ForAll and Exists commands also allow you to express conditions on the bound variable by use of an optional second argument. For example, to symbolically express the assertion "For all $x > 0$, $-x < 0$," you include the condition $x > 0$ as the second argument and the expression $-x < 0$ as the third argument.

```
In[142]:= ForAll[x,x>0,-x<0]
Out[142]= ∀x,x>0 -x<0
```

You can, in particular, use the condition to specify the domain, or universe of discourse, by asserting that the variable belongs to one of the Wolfram Language's recognized domains using the Element function. To assert, for example, that x is a real number, use the Element function with first argument x and second argument Reals, the Wolfram Language symbol for the domain of real numbers.

```
In[143]:= Exists[x,Element[x,Reals],x^2<0]
Out[143]= ∃x,x∈R x^2<0
```

The Wolfram Language has seven defined domains that you can use: Reals, Integers, Complexes, Algebraics, Primes, Rationals, and Booleans.

Truth Values of Quantified Statements

In addition to symbolically representing quantified statements, *Mathematica* can determine whether they are true or false. The Resolve function, applied to an expression involving quantifiers, will eliminate the quantifiers. For expressions like the ones given above, this result will be the truth value of the statement.

```
In[144]:= Resolve[Exists[x,-x<0]]
Out[144]= True

In[145]:= Resolve[ForAll[x,x>0,-x<0]]
Out[145]= True

In[146]:= Resolve[Exists[x,Element[x,Reals],x^2<0]]
Out[146]= False
```

The syntax of the last example can be simplified by using a second argument to Resolve. Rather than using the Element function within the existential statement, we can obtain the same effect by putting the domain Reals as a second argument to Resolve.

```
In[147]:= Resolve[Exists[x,x^2<0],Reals]
Out[147]= False
```

Note that we obtain a different result by changing the domain.

```
In[148]:= Resolve[Exists[x,x^2<0],Complexes]
Out[148]= True
```

For existential quantification, *Mathematica* can go beyond just finding the truth value and actually give you witnesses for the existence of objects with the desired property. This is done using the FindInstance

function. For example, the statement $\exists_x x^3 = 8$ is true. (Note that to enter an equation, we must use the `Equal` (==) relation so as to avoid confusion with assignment.)

```
In[149]:= Resolve[Exists[x,x^3==8]]

Out[149]= True
```

We can find a witness for this by applying `FindInstance` with the expression $x^3 = 8$ as the first argument and the variable as the second variable.

```
In[150]:= FindInstance[x^3==8,x]

Out[150]= {{x→2}}
```

`FindInstance` accepts two optional arguments. You can ask for more than one witness just by giving the number of instances you would like to find as an argument.

```
In[151]:= FindInstance[x^3==8,x,3]

Out[151]= {{x→2},{x→-1-i Sqrt[3]},{x→-1+i Sqrt[3]}}
```

You can also restrict the results to a particular domain by giving the domain as an argument. Note that when giving both a domain and a specific number of results, the domain should be the third argument and the number the fourth argument. Below, we have asked for more instances than exist, so *Mathematica* just returns the one witness.

```
In[152]:= FindInstance[x^3==8,x,Integers,3]

Out[152]= {{x→2}}
```

If a statement has one or more free variables, *Mathematica* can be used to find conditions on those variables in order to make a statement true. For example, consider the statement $\forall_x x \cdot y = 0$. In this statement, x is bound and y is free. The `Reduce` function can be used to solve for free variables. Apply it with the statement as the first argument and the free variable (or list of variables) as the second argument.

```
In[153]:= Reduce[ForAll[x,x*y==0],y]

Out[153]= y==0
```

The result, $y = 0$, means that if the free variable y is replaced by the value 0, then the statement will be true.

`Reduce` accepts a domain as an optional third argument.

```
In[154]:= Reduce[Exists[x,x^2==y],y,Reals]

Out[154]= y≥0
```

This result means that, restricting all variables to the real numbers, if y is replaced by any nonnegative real number, the existential statement $\exists_x x^2 = y$ will be true. Note that if the domain restriction is removed, *Mathematica* defaults to complex numbers and so there would be no restriction on y.

```
In[155]:= Reduce[Exists[x,x^2==y],y]

Out[155]= True
```

1.5 Nested Quantifiers

The Wolfram Language can be used to represent statements with nested quantifiers with the same functions described in the previous section.

For statements in which all the quantifiers are of the same kind, you only need to use a single `Exists` or `ForAll` function with the list of variables surrounded by braces as the first argument. For example, to represent the statement $\forall_x \forall_y (x \cdot y = 0) \rightarrow (x = 0 \vee y = 0)$, we only need one `ForAll` function with `{x,y}` as the first argument.

```
In[156]:= ForAll[{x,y},Implies[x*y==0,x==0||y==0]]
Out[156]= ∀{x,y}(x y==0⇒x==0||y==0)
```

Using the `Resolve` command, we see that *Mathematica* recognizes this as true.

```
In[157]:= Resolve[ForAll[{x,y},Implies[x*y==0,x==0||y==0]]]
Out[157]= True
```

Note that since we did not specify a domain, *Mathematica* uses the default domain based on the context. In this case, it uses the complex numbers as its domain, since the content of the statement is about equations. In general, the default domain is the largest domain that makes sense in the context.

For statements that involve more than one type of quantifier, we must nest the `Exists` and `ForAll` functions. For example, to represent $\forall_{x \neq 0} \exists_y x \cdot y = 1$, we enter the following.

```
In[158]:= ForAll[x,x≠0,Exists[y,x*y==1]]
Out[158]= ∀x,x≠0∃y x y==1
```

Again, `Resolve` recognizes the truth of this statement.

```
In[159]:= Resolve[ForAll[x,x≠0,Exists[y,x*y==1]]]
Out[159]= True
```

However, limiting the domain to the integers makes the statement false.

```
In[160]:= Resolve[ForAll[x,x≠0,Exists[y,x*y==1]],Integers]
Out[160]= False
```

In addition, observe that reversing the order of the quantifiers changes the meaning of the statement.

```
In[161]:= Exists[y,ForAll[x,x≠0,x*y==1]]
Out[161]= ∃y∀x,x≠0 x==1
In[162]:= Resolve[Exists[y,ForAll[x,x≠0,x*y==1]]]
Out[162]= False
```

Finally, *Mathematica* will automatically apply DeMorgan's laws for quantifiers to a statement that you enter.

```
In[163]:= !ForAll[x,Exists[y,ForAll[z,P[x,y,z]]]]
Out[163]= ∃x∀y∃z!P[x,y,z]
```

1.6 Rules of Inference

In this section, we will see how *Mathematica* can be used to verify the validity of arguments in propositional logic. In particular, we will write a function that, given a list of premises and a possible conclusion, will determine whether or not the conclusion necessarily follows from the premises. Recall from 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 `TautologyQ` function described in Section 1.3 of this manual to test whether a proposition is a tautology. 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.)

```
In[164]:= TautologyQ[Implies[(!q&&Implies[p,q]),!p]]
Out[164]= True
```

To determine whether 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 `TautologyQ`. The `validQ` function below will accept as input an argument, that is, a list of propositions, and return true if the argument is valid.

```
In[165]:= validQ[A_]:=Module[{premiseList,premises},
            premiseList=A[[1;;-2]];
            premises=Apply[And,premiseList];
            TautologyQ[Implies[premises,A[[-1]]]]
          ]
```

Two comments on the code above are needed. First, the double semicolons used in the definition of `premiseList` is the `Span` (`;;`) operator. When used to refer to a `Part` (`[[...]]`) of a list, *i*`;;`*j* indicates the range from index *i* to index *j*. In this case, `-2` indicates the next to last entry of the list. Therefore, `A[[1;;-2]]` refers to all of `A` except the last entry and is thus the premises of the argument `A`. Second, the `Apply` operator is used to apply the function `And` to the arguments contained in the list `premiseList`. This is necessary because, while `And` can accept any number of arguments and form the logical conjunction, it will not do anything with a single list like `premiseList`. When `Apply` is given a function and a list, the result is the same as if the elements of the list were given as the arguments to the function. Fundamentally, `Apply` is replacing the head of the list, `List`, by the name of the function.

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

```
In[166]:= validQ[{Implies[p&&t,r||s],Implies[q,u&&t],Implies[u,p],
            !s,Implies[q,r]}]
Out[166]= True
```

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 function did not do that. It essentially used truth tables to check validity. It would be considerably more difficult to program *Mathematica* 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 consider a slightly different question: given a list of premises, what conclusions can you draw using valid arguments? We will approach this problem in a straightforward (and naive) way: generate possible conclusions and use `validQ` to determine which are valid conclusions.

Making Compound Propositions

To generate possible conclusions, we will use the following function, `allCompound`. This function takes a list of propositions and produces all possible propositions formed from one logical connective (selected from `And`, `Or`, and `Implies`) and two of the given propositions along with the negations of the propositions. To avoid including some trivialities, we will exclude those propositions that are tautologies or contradictions.

The function is provided below. Note the use of `AppendTo`, which accepts a list and an element to be added to the list as arguments. It has the result of adding the given element to the list and updating the list without the need of an explicit assignment. In addition, note that at the end of the function, we apply `DeleteDuplicates` so as to remove repeated elements from the list. Pay attention to the uses of `Do`, which allow us to loop over all the elements (or combinations of elements) of lists.

The bulk of the function is taken up by adding the conjunction, disjunction, and implication of the chosen pair to the result list, provided that they do not form tautologies or contradictions.

```
In[167]:= allCompound[vars_]:=Module[{p,q,tempList=vars,propList},
            Do[AppendTo[tempList,!p],{p,vars}];
            propList=tempList;
            Do[If[!TautologyQ[p&&q]&&!TautologyQ[!(p&&q)],
               AppendTo[propList,p&&q]];
              If[!TautologyQ[p||q]&&!TautologyQ[!(p||q)],
                AppendTo[propList,p||q]];
              If[!TautologyQ[Implies[p,q]]&&!TautologyQ[!Implies[p,q]],
                AppendTo[propList,Implies[p,q]]]
              ,{p,tempList},{q,tempList}];
            DeleteDuplicates[propList]]
```

Finding Valid Conclusions

Now we write a function to explore possible conclusions given a set of premises. This function will take two arguments. The first will be a list of premises. The second will be 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 can be quite substantial.

The operation of this function is fairly straightforward. First, it determines the variables used in the provided premises by applying the `getVars` function we wrote above. Second, beginning with the list of variables, it recursively applies `allCompound` a number of times equal to the level, which is the second argument to the function. Finally, for each of the possible conclusions generated by the `allCompound` function, it uses `validQ` to see if it is a valid conclusion from the premises, and adds those that are to the output.

```
In[168]:= findConsequences[premises_,level_]:=
            Module[{vars,P,possibleC,conclusions={},c,i},
              vars=getVars[premises];
              possibleC=vars;
              For[i=1,i≤level,i++,possibleC=allCompound[possibleC]];
              Do[
                If[validQ[Append[premises,c]],AppendTo[conclusions,c]],
                {c,possibleC}
              ];
              conclusions
            ]
```

Here is the result of applying `findConsequences` to the premises of Exercise 12 with only one iteration of `allCompound`. (With two iterations, the function takes quite some time to complete and produces thousands of valid conclusions.)

```
In[169]:= findConsequences[{Implies[p&&t,r||s],Implies[q,u&&t],
            Implies[u,p],!s},1]

Out[169]= {!s,p||!s,p⇒!s,p||!q,p||!u,t||!s,t⇒!s,t||!q,
            r||!s,r⇒!s,r||!q,s⇒p,s⇒t,s⇒r,s⇒q,s⇒u,
            s⇒!p,s⇒!t,s⇒!r,s⇒!s,s⇒!q,s⇒!u,q⇒p,q⇒t,
            q⇒r,q⇒u,q||!s,q⇒!s,u⇒p,u||!s,u⇒!s,u||!q,
            !p||!s,!p⇒!s,!p⇒!q,!p⇒!u,!t||!s,!t⇒!s,
            !t⇒!q,!r||!s,!r⇒!s,!r⇒!q,!s||p,!s||t,!s||r,
            !s||q,!s||u,!s||!p,!s||!t,!s||!r,!s&&!s,
            !s||!s,!s||!q,!s||!u,!q||p,!q||t,!q||r,!q||u,
            !q||!s,!q⇒!s,!u||p,!u||!s,!u⇒!s,!u⇒!q}

In[170]:= Length[%]

Out[170]= 64
```

Observe that some of the conclusions are merely restating premises. However, 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 *Mathematica* can be used to find counterexamples. This is the proof technique most suitable to *Mathematica*'s computational abilities.

Example 15 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 will 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 will create a function that, given an integer, looks for three integers with the property that the sum of their squares is equal to the given integer. If the function 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 function:

In[171]:=
```
find3squares[n_]:=Module[{a,b,c,max=Floor[Sqrt[n]]},
  Catch[
    For[a=0,a≤max,a++,
      For[b=0,b≤max,b++,
        For[c=0,c≤max,c++,
          If[n==a^2+b^2+c^2,Throw[{a,b,c}]]
        ]
      ]
    ];
    Throw[False]
  ]
]
```

The `find3squares` function is straightforward. We use three `For` loops to check all possible values of `a`, `b`, and `c`. Each 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 15, can be written as the sum of three squares.

In[172]:=
```
find3squares[3]
```

Out[172]=
```
{1,1,1}
```

To find a counterexample to the claim that "Every positive integer is the sum of the squares of three integers," we write a function that, starting with 1, tests numbers using `find3squares` until a value is found that causes it to return false.

```
In[173]:= find3counter:=Module[{n=1},
            While[find3squares[n]=!=False,n++];
            n
          ]
```

First, note that this "function" does not take an argument, so we will not use brackets when we execute it.

Moreover, note that the `While` loop is controlled by the return value of `find3squares`. This is a fairly common approach when you are looking for an input value that will cause another function to return a desired result. As before, when comparing nonnumerical objects, we use `UnsameQ` (=!=).

To find the counterexample, all we need to do is to call the function.

```
In[174]:= find3counter

Out[174]= 7
```

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

Let us 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 function whose goal is to find three integers whose squares sum to its argument. 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` function did we write `find3counter`, whose task was to find a counterexample to the universal statement in question. 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 input 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 functions we wrote found a candidate for a counterexample, but we do not 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 approach 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." We now prove this biconditional.

We first prove the statement: 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 use a direct proof. Assume that n can be written as the sum of three squares. Say $n = a^2 + b^2 + c^2$ for integers a, b, and c. Note that we may take a, b, and c to be nonnegative integers, since an integer and its negative have the same square. In addition, $n = a^2 + b^2 + c^2 \geq a^2$. Thus, $n \geq a^2$ 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 function. We have assumed that *n* can be written as the sum of three squares and concluded that `find3squares[n]` must return the list of the integers.

The converse is that if `find3squares[n]` 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[n]` 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. 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. For more about this topic, refer to Section 5.5 of the main text.

1.8 Proof Methods and Strategy

In this section, we will consider two additional proof techniques that *Mathematica* 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 *Mathematica*, 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 function, `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 function, `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 function, `findPowers`, we need to generate all perfect powers up to the given limit. To do this, we will use a nested pair of loops for the exponent (`p`) and the base (`b`). Each of the loops will be a `While` loop controlled by a Boolean variable, `continuep` and `continueb`. In the inner loop, we check to see if `b^p` is greater than the limit, `n`, given as the input to the function. If it is, then we set `continueb` to false, which terminates the inner loop, and if not, we add `b^p` to the list of perfect powers, `L`, and increment `b`. Once the inner `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.

```
In[175]:= findPowers[n_]:=
            Module[{L={},b,p=2,continuep=True,continueb},
              While[continuep,
                b=1;
                continueb=True;
                While[continueb,
                  If[b^p>n,continueb=False,AppendTo[L,b^p];b++]
                ];
                p++;
                If[2^p>n,continuep=False]
              ];
              Union[L]
            ]
```

Note that the `Union` function, applied to a single list, returns the list sorted and with duplicates removed. We confirm that the list of powers produced by this algorithm is the same as the powers considered in Example 2 from the text.

```
In[176]:= findPowers[100]

Out[176]= {1,4,8,9,16,25,27,32,36,49,64,81,100}
```

The second function, `findConsecutivePowers`, begins by calling `findPowers` and storing the list of perfect powers as `powers`. Then, we use a `Do` loop with second argument `{x,powers}`. This sets the variable `x` equal to each element of the list `powers`. In our procedure, this means that `x` is set to each of the perfect powers in turn. In the body of the loop, we check to see if the next consecutive integer, `x+1`, is also a perfect power using the `MemberQ` function. The `MemberQ` function requires two arguments. The first argument is a list to search and the second argument specifies what is being sought. When we find consecutive perfect powers, we `Print` them.

```
In[177]:= findConsecutivePowers[n_]:=Module[{powers,x},
              powers=findPowers[n];
              Do[If[MemberQ[powers,x+1],Print[x," ",x+1]],{x,powers}]
            ]
```

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 function.

```
In[178]:= findConsecutivePowers[100 000 000]

          8 9
```

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 *Mathematica*. Consider Example 10 in Section 1.8 of the text. This example asks, "Show that there is a positive integer that 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 us 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 will create a list `L` and every time we generate a new sum of two cubes, we check to see if that number is already in `L` using the `MemberQ` function. If the new value is already in `L`, then that is the number we are looking for. Otherwise, we add the new number to `L` and generate a new sum of two cubes.

We 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 arising from commutativity of addition (e.g., $9 = 2^3 + 1^3 = 1^3 + 2^3$). The outer loop will be a `While` loop with condition (first argument) `True`. The value of `a` will be initialized to 1 and incremented by 1 after the inner `b` loop completes. The `While` loop in this case is called an infinite loop because it will never stop on its own. When the function finds an integer which can be written as the sum of cubes in two different ways, the function will `Throw` that value. That ends the loop and is sent to the `Catch`, which encompasses the entire body. The infinite loop means that the value of `a` will continue getting larger and larger with no upper bound. This is useful because we do not 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 are not certain that the procedure will terminate in a reasonable amount of time.

Here is the function and its result.

```
In[179]:= twoCubes:=Module[{L={},a=1,b,n},
            Catch[
              While[True,
                For[b=1,b≤a,b++,
                  n=a^3+b^3;
                  If[MemberQ[L,n],Throw[n],AppendTo[L,n]]
                ];
                a++
              ]]]

In[180]:= twoCubes

Out[180]= 1729
```

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 `myEquivalentQ` function in Section 1.3. Note, of course, that the Wolfram Language provides a built-in function, `SatisfiableQ`, that performs this function. The goal of this exercise is to see how such a function might be implemented.

We create a function, `mySatisfiableQ`, that checks all possible assignments of truth values to the propositional variables. The `mySatisfiableQ` function 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 function returns the `result` variable.

Since this function is otherwise very similar to `myEquivalentQ`, we offer no further explanation.

In[181]:=
```
mySatisfiableQ[p_]:=
  Module[{result=False,vars,numVars,TA,val},
    vars=getVars[p];
    numVars=Length[vars];
    TA=ConstantArray[False,numVars];
    While[TA=!=Null,
      val=p/.MapThread[Rule,{vars,TA}];
      If[val,result=True;Print[TA]];
      TA=nextTA[TA];
    ];
    result
  ]
```

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

In[182]:=
```
mySatisfiableQ[(p||!q)&&(q||!r)&&(r||!p)]
```

```
{False,False,False}

{True,True,True}
```

Out[182]=
```
True
```

In[183]:=
```
mySatisfiableQ[(p||q||r)&&(!p||!q||!r)]
```

```
{True,False,False}

{False,True,False}

{True,True,False}

{False,False,True}

{True,False,True}

{False,True,True}
```

Out[183]=
```
True
```

```
In[184]:= mySatisfiableQ[(p||!q)&&(q||!r)&&(r||!p)&&(p||q||r)&&
            (!p||!q||!r)]
Out[184]= False
```

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. We will set a limit n, that is, the maximum integer that we are 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 $a_i^3 \leq n$.

We will also want to make our approach as efficient as possible in order to find as many such integers as we can. 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 expressed 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 do not need to write a function to find all possible sums of eight cubes. Instead, we will write a function that, given a list of numbers, will find all possible sums of two numbers that are both in that list. If we apply this function to the cubes of the numbers from 0 through $\sqrt[3]{n}$, that will produce all numbers that are the sum of two cubes. Applying the function 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 have 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 than $\sqrt[3]{n}$; however, these do not provide us with any information about numbers that cannot be generated as the sum of eight cubes. Moreover, 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 function that finds all possible sums of pairs of integers from the given list `L` up to the specified maximum value `max`. Note that we use the `Union` function to remove redundancies and also put the list in increasing order.

```
In[185]:= allPairSums[L_,max_]:=
            Module[{a=1,b,s,sumList={},num=Length[L]},
              While[a≤num,
                b=a;
                While[b≤num,
                  s=L[[a]]+L[[b]];
                  If[s≤max,AppendTo[sumList,s],b=num];
                  b++;
                ];
                a++;
              ];
              Union[sumList]
            ]
```

With this function in place, we need to apply it (three times) to a list of cubes. We consider cubes up to 7^3, including 0. The `Table` function used below forms the list of all values obtained by evaluating the first argument after replacing the variable `i` by every integer between (inclusive) the two given in the second argument.

```
In[186]:= someCubes=Table[i^3,{i,0,7}]

Out[186]= {0,1,8,27,64,125,216,343}
```

Applying the `allPairSums` function once gives us all the sums of pairs of cubes (up to $7^3 = 343$).

```
In[187]:= sumtwoCubes=allPairSums[someCubes,343]

Out[187]= {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}
```

Applying it to that result gives all possible sums of four cubes.

```
In[188]:= sumfourCubes=allPairSums[sumtwoCubes,343]

Out[188]= {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}
```

Once again, we obtain all integers up to 343 that can be obtained as the sum of eight cubes.

```
In[189]:= sumeightCubes=allPairSums[sumfourCubes,343]

Out[189]= {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,
  198,199,200,201,202,203,204,205,206,207,208,209,210,211,
  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}
```

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

```
In[190]:= For[i=1,i≤343,i++,If[!MemberQ[sumeightCubes,i],Print[i]]]

23

239
```

Exercises

1. Write functions `or`, `xor`, and `not` to implement those bit string operators.
2. Solve Exercises 23 through 27 in Section 1.2 of the main textbook, using the knights and knaves puzzle that was solved earlier in this chapter as a guide.

3. Implement the solution of Sudoku puzzles as a satisfiability problem described in the main text.
4. Write a function in the Wolfram Language 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 heads in logical expressions.)
5. Write a function `uniqueness`, based on the built-in `Exists` and `ForAll` functions, to implement the uniqueness quantifier, described in Section 1.4 of the text.
6. Write a function in the Wolfram Language that plays the obligato game in the role of the student, as described in the Supplementary Exercises of Chapter 1 in the main textbook. Specifically, the function 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 *Mathematica*'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 function/student accepts the first statement and denies the second statement, then you would obtain the response to the third statement by executing

```
obligato[r,{Implies[p,q|r],!(!p||q)}]
```

The function must accept the statement **r** and thus returns the list with that response included, as shown below:

```
{p⇒q||r,!(!p||q),r}
```