Maple for Math Majors

Roger Kraft
Department of Mathematics, Computer Science, and Statistics
Purdue University Calumet
roger@calumet.purdue.edu

14. Maple's Control Statements

14.1. Introduction

In order to write interesting examples of procedures, we need to define two new kinds of Maple commands, the **repetition statement** and the **conditional statement**. These are called **control statements** since they control the order in which Maple commands are executed. So far, whenever we have lumped Maple commands into a group, either in an execution group or in a procedure body, the commands in the group have been executed in a sequential, or linear, order, one after the other. Now we will see how to get Maple to execute commands in a "nonlinear" order.

```
> 
```

14.2. Repetition statements

Sometimes we want Maple to do more or less the same thing many times in a row. Another way to put this is that sometimes we might want Maple to repeat a command (or group of commands) many times. How can we get Maple to do something in a very repetitious way? By using a Maple command called a **for-loop**. Here is a basic example of a for-loop; it computes powers of 2.

```
> for i from 0 to 6 do 2^i; od;
```

This for-loop told Maple to evaluate the expression `2^i` seven times but to use a different value for `i` each time. The command produced seven lines of output, one line for each of the successive values that `i` takes from 0 to 6. What if you wanted a lot more, say 20 or 30 lines of output? Notice how easy it would be to go back and change the command to generate as many lines of output as you might want. (Try it.)

Here are some simple modifications of this last for-loop. First of all, we can have Maple do more then one command each time the for-loop loops (or iterates).

```
> for i from 0 to 6 do 2^i; 3^i; od;
```

For each time the loop looped, Maple executed two commands, one evaluating `2^i` and the other evaluating `3^i`, so there are 14 lines of results. But this is hard to read so instead of having Maple execute two commands per loop, let us have it compute two results per loop and put those results into one expression sequence per loop (i.e., change a semi colon into a comma).

```
> for i from 0 to 6 do 2^i, 3^i; od;
```

Now we are back to seven lines of output and this time each line has two numbers on it. Let us add a third number to each output line, the loop counter `i`, so that we can easily see which value of `i` was used to compute each line of output.

```
> for i from 0 to 6 do i, 2^i, 3^i; od;
```
Let us try to make those results a little more self explanatory.
```
> for i from 0 to 6 do 'i' = i, '2^i' = 2^i, '3^i' = 3^i; od;
```
Notice how the right quotes, for delayed evaluation, were useful. What output would the next command produce and why?
```
> for i from 0 to 6 do 'i' = i; '2^i' = 2^i, '3^i' = 3^i; od;
```
We can have our loop begin with a negative number.
```
> for i from -4 to 2 do i, 2^i, 3^i; od;
```
We can also make our loop count by twos. We just add an extra clause to the for-loop statement.
```
> for i from 0 to 10 by 2 do i, 2^i, 3^i; od;
```
We can even make our loop count backwards.
```
> for i from 6 to 0 by -1 do i, 2^i, 3^i; od;
>
```

Exercise: Write a loop that will count backwards by threes from 12 to -9. (Replace the question marks in the next command).
```
> for i from ? to ? by ? do i, 2^i, 3^i; od;
>
```

The syntax for a basic for-loop in Maple is

for *index_variable* **from** *initial_value* **to** *final_value* **by** *step_size* **do**
sequence_of_Maple_commands
od;

The Maple commands between the **do** and **od** are called the **body of the for-loop**. The for-loop executes (or "loops through", or "iterates") its body over and over again, as the index variable increments (or decrements) by the step size from the initial value to the final value. The **by** *step_size* part of the for-loop is optional. If *initial_value* is less than *final_value* and there is no *step_size*, then *step_size* defaults to one. The for-loop stops iterating when the *index_variable* is incremented, by the *step_size*, to a value that is greater than *final_value* (or, if *step_size* is negative, looping stops after the *index_variable* is decremented by *step_size* to a value less than *final_value*). Notice that this implies that the for-loop is not completed until the index variable is incremented *past* the final value. We will look at examples of this just below.

Exercise: What happens with a for-loop for which the *final_value* is less than the *initial_value* and there is no *step_size*? Does *step_size* default to –1?
```
>
```

The formatting of a for-loop as shown above, with the **od** on its own line and the *sequence_of_Maple_commands* indented a bit, is not part of the syntax of a for-loop. This way of formatting a for-loop is just a way to help a reader understand the structure of the loop. It is a very

good idea to write for-loops using a format like this. But be careful to have all of the lines that make up the for-loop in one execution group. (As far as Maple is concerned, the entire for-loop is just one single Maple command, so you can also have the entire for-loop on a single line if it is a short loop).

Here are some examples of simple for-loops. Notice that a for-loop ignores whatever value the index variable might have before the for-loop is executed.

```
> i := -100;
> for i from 5 to 10 by 2 do 'i' = i od;
```

Notice that the last output from the for-loop was **i=10**. Let us check the current value of **i**

```
> i;
```

When a for-loop terminates, the value of *index_variable* is always greater than *final_value*. But exactly how much greater depends on all three of the initial value, the final value, and the step size. Consider the following examples.

```
> for i from 3 to 7 do 'i'=i od;
> i;

> for i from 3 to 7 by 2 do 'i'=i od;
> i;

> for i from 3 to 8 by 2 do 'i'=i od;
> i;

> for i from 2 to 8 by 2 do 'i'=i od;
> i;

> for i from 2 to 8 by 3 do 'i'=i od;
> i;
>
```

Exercise: For a for-loop, let *i* denote the *initial_value*, let *f* denote the *final_value*, and let *s* denote the *step_size*. Suppose all three of these are positive and $i < f$. Let *a* denote the value of the *index_variable* after the loop has completed. Find a formula for the value of *a* in terms of *i*, *f*, and *s*. Use Maple to verify your formula.

```
>
```

The **from** *initial_value* and the **to** *final_value* parts of a for-loop can be omitted. This is more likely to happen by accident than by design, but if you do leave them out, even accidentally, Maple will not warn you since it is not a syntax error. If you omit **from** *initial_value*, then the *initial_value* defaults to 1.

```
> for i to 5 do 'i'=i od;
>
```

However, if you should omit the **to** *final_value* part of a for-loop, then the for-loop never terminates! This can be inconvenient. If this should happen, sometimes you can get Maple to stop by

clicking on the "**Stop**" button near the top of the Maple window. But Maple does not always respond to the "**Stop**" button. If Maple refuses to respond to the stop button, then you have no choice but to wait for some kind of error to occur, which will stop Maple, or you need to use the operating system to halt the Maple program, which will cause you to loose all of your work since the last time you saved your worksheet. So look over your for-loops carefully before hitting the return key.

The for-loop syntax we just looked at is the syntax for the *basic* loop. Maple has many other forms of loops. We will look at some of these other forms in some of our examples later on.

For-loops are useful for generating "tables" of results. For example the next command lists the binary and hexadecimal number system versions of some integers.

```
> for i from 100 to 110 do
>   i, convert(i, binary), convert(i, hex);
> od;
```

Notice how the for-loop was written on three lines and the middle line was indented. This is to try and make the for-loop easier to read. It is important that all three of those lines be in the same execution group, otherwise Maple will generate error messages. But it is also important to realize that those three lines were not separate Maple commands. The three lines together make one for-loop and Maple treats that for-loop as a *single* Maple command.

```
>
```

The next command creates an interesting table of polynomials and their factorizations.

```
> for i from 1 to 10 do
>   exp1 := add( x^j, j=0..i );
>   exp2 := factor( exp1 );
>   print( exp1 = exp2 );
> od:
```

Let us examine this for-loop more carefully since it has a few new features. First of all, notice that the **od** at the end of the for-loop has a colon after it, not a semi colon. That actually means that the for-loop does *not* print out any result. So why do we get ten lines of output? The output lines are produced by the **print** command, which always prints out a result. For example, consider the next two print commands. One has a colon and the other has a semi colon after it but they both print out a result.

```
> print( "hello" ):
> print( "hello again" );
```

Now go back up to the last for-loop and change the colon after the **od** to a semi colon. If you re-execute the loop it will now produce 30 lines of output since there are three commands in the loop body and the body is executed ten times. Try changing the semi colons in the loop body to colons and re-executing the for-loop. If you do, you will still get 30 lines of output. In the body of a for-loop (as in the body of a procedure) there is no difference between a colon and a semi colon. So either all the commands in the body of a for-loop produce an output or they all do not produce an output, and it is the semi colon or colon after the **od** that controls this. The best way to control which results you want printed out and which results you want to suppress from the body of a loop is

to put a colon after the final **od** of the for-loop and then use the **print** command inside the body of the loop to selectively print out the results that you want to see. (There is an optional section later in this worksheet that has more to say about the use of the **print** command inside loops.)

```
> 
```

Notice that for-loops can quickly and easily produce a lot of results. For example, the next for-loop produces quite a few polynomials.

```
> for i from 1 to 5 do
>    add( x^j, j=0..i );
>    factor( % );
> od;
```

What if we wanted to give these polynomials names so that we can refer to them later on? We need a way for the for-loop to automatically generate names, similar to how it can automatically generate the polynomials. In fact, Maple provides two ways for a for-loop to create names as it loops. Here we will look briefly at these two ways, which are called dotted names and indexed names. (In an earlier worksheet there were optional sections on dotted and indexed names that had more information in it them then what we need now.)

A dotted name is a name followed by a dot followed by a number, such as **x.3**. The dot is really an operator and Maple evaluates the dot by taking the number on the right of the dot and concatenating it to the name on the left of the dot. So **x.3** becomes **x3**.

```
> x.3;
```

What makes this so useful is that the number on the right of the dot can come from evaluating a variable. So if the value of **i** is 3 them **x.i** evaluates to **x3**.

```
> i := 3;
> x.i;
```

In our loop that created lots of polynomials, here is how we can give the factored ones names.

```
> for i from 1 to 5 do
>    add( x^j, j=0..i );
>    p.i := factor( % );
> od;
```

And now **p3**, for example, is a name for one of these factored polynomials.

```
> p3;
> 
```

An indexed name is a name followed by a pair of brackets enclosing a number, such as **x[3]**. (We have seen indexed names used before as a way of accessing the data items in expression sequences, lists, sets, and strings.)

```
> x[3];
```

Notice how Maple typesets the indexed name as a subscripted variable. The number inside the brackets can come from evaluating a variable. So if 3 is the value of **i**, then **x[i]** evaluates to the indexed name **x[3]**.

```
> i := 3;
> x[i];
```
In our loop that created a lot of polynomials, here is how we can associate indexed names with the rest of the polynomials.
```
> for i from 1 to 5 do
>    p[i] = add( x^j, j=0..i );
>    p.i := factor( % );
> od;
```
Notice how in this example the indexed names were used in equations, not in assignment statements. Sometimes we only want the names for display purposes, not for assigning to. (If you want, you can try changing the equations into assignments in the for-loop.)
```
>
```

This next loop computes the first ten prime numbers. Each prime is displayed in an equation with an indexed name.
```
> for j from 1 to 10 do
>   prime[j] = ithprime(j);
> od;
```
We can assign dotted names to the primes like this.
```
> for j from 1 to 10 do
>   prime.j := ithprime(j);
> od;
```
Here is a fancy way to list a few of our primes using their dotted names.
```
> prime.(5..9);
>
```

Exercise: Compute the one thousandth through one thousand and tenth prime numbers.
```
>
```

For-loops can take on forms other than the ones used in the examples given above. One other form is the **for-in-loop**. Its syntax is

for *index_variable* **in** *data_structure* **do** *sequence_of_Maple_commands* **od**

Here are a few examples. Study them carefully. Try making some changes in them.
```
> for i in [10!, 11!, 12!, 13!, 14!, 15!, 16!] do length(i); od;

> counter := 0;
> for i in f(x, y, z, w) do counter := counter+1; od;

> for letter in "Hello world." do letter; od;

> for term in x^2+2*x+(1/x)-y+z do term, whattype(term); od;
```

What does the following command do?

```
> for n in seq(i!, i=10..16) do ifactor(n); od;
```

Here is a way to get one line from Pascal's triangle.

```
> for term in expand((a+b)^8) do coeffs(term); od;
>
```

Still another form for a repetition statement is the **while-loop**. It has the following syntax.

while *boolean_expression* **do** *sequence_of_Maple_commands* **od**

The while-loop iterates as long as the boolean expression is true (we will say more about boolean expressions later in this worksheet). Here is an example of how we can make a while-loop iterate exactly ten times. This loop determines which numbers between one and ten are or are not primes.

```
> i := 1;  # Give i an initial value.
> while i <= 10 do
>    i, "Is it a prime?", isprime(i);
>    i := i+1;
> od;
>
```

Let us go over this example in detail. Notice from the syntax for a while-loop that while-loops do not have index variables, so **i** is *not* an index variable. That is why we have to initialize **i** before executing the while-loop. The while-loop iterates as long as **i** is less than 10 (so **i** acts a lot like an index variable, but it is not one). Notice that the body of the loop has two statements, one of which increments **i** to the next value. In a for-loop, the index variable is incremented automatically from one iterate of the loop to the next, but **i** is not an index variable so we need to increment it ourselves. Notice what the value of **i** is when the loop terminates.

```
> i;
```

The value of **i** needed to reach 11 so that the boolean expression **i<=10** could become false and terminate the loop. If we were to forget the **i:=i+1** statement in the body of this while-loop, the loop would iterate for ever (why?), which is sometimes called an "infinite loop". (You can try this if you wish, but be careful. First be sure to save all of your work, in case you cannot get Maple to stop. Then delete the **i:=i+1** statement, execute the infinite loop, then use the "**Stop**" button at the top of the Maple window to bring the loop to a premature end.)

Exercise: Modify the while-loop so that it does not output all the results from the incrementing statement **i=i+1**.

```
>
```

Exercise: Modify the while-loop example so that it determines which integers between 1000 and 1010 are primes.

```
>
```

Exercise: Even numbers are never prime (except for 2) so there is no point in testing them. Modify the while-loop example so that it skips over the even numbers and only tests the odd numbers.

```
>
```

Exercise: Rewrite the while-loop example as a for-loop. Notice that the for-loop version is much more straight forward. In the next example we will look at a while-loop that cannot be rewritten as a for-loop.

```
>
```

The previous example is really not a good use for a while-loop. It is better written as a for-loop. This is because we know in advance exactly how many times we want the loop to iterate. The strength of a while-loop is that it can be used for loops in which we do not know in advance how many times the loop will need to iterate. For example, the following while-loop finds the first prime number greater than the integer **n**. Since we do not know which prime number this will be, we do not know in advance how many times the loop will iterate as it searches for the answer. Notice that the variable **n**, which is not an index variable, needs to be initialized before the loop starts and we need to increment **n** ourselves in the body of the loop.

```
> n := 1000:
> while not isprime(n) do n := n+1 od;
```

This next while-loop will find the first prime number less than the integer **n**. Notice how we have to decrement **n** ourselves in the body of the loop.

```
> n := 1000:
> while not isprime(n) do n := n-1 od;
>
```

Exercise: What is the output from either of the last two while-loops if the initial value of **n** is a prime number? Explain why.

```
>
```

Exercise: What happens in either of the last two while loops if **n** is a negative number. (Warning: Think about this before trying it out!)

```
>
```

A common way to use loops is to nest them inside of each other. The following nested for-loops count how many prime numbers there are less than 2^i for each i from 2 to 18. The outer for-loop increments i from 2 to 18 and it contains two commands in the body of the loop, another for-loop and a **print** command. The inner for-loop counts how many primes there are between **2^(i-1)** and **2^i** (notice how the index variable from the outer loop is used to set the initial and final values for the index variable in the inner loop). When the inner loop completes its iterates, the **print** command in the outer loop prints out the running total of primes found so far. Notice how the

for-loops are indented to make it easier to distinguish them.

```
> counter := 1:
> for i from 2 to 18 do
>   for j from 2^(i-1)+1 to (2^i)-1 by 2 do
>     if isprime(j) then counter := counter+1 fi
>   od;
>   print( '2^i'=2^i, `number of primes`=counter );
> od:
```

The following nested loops do the same calculation as the above nested loops, but the following version uses a while-loop nested inside of a for-loop.

```
> j := 1:
> for i from 2 to 18 do
>   while ithprime(j) < 2^i do j := j+1 od;
>   print( '2^i'=2^i, `number of primes`=j-1 );
> od:
```

It is interesting to note that when the above two examples have their outer loop index run from 2 to 18, then the examples run in about the same amount of time. But when the outer loop index is changed to go from 2 to 19, then the second example takes *much* longer to run then the first example.

```
>
```

The following while-loop finds the first integer whose factorial has 1000 digits. Notice again that the variable **i**, which is not an index variable, needs to be initialized before the loop starts and we need to increment **i** ourselves in the body of the loop.

```
> i := 1:
> while length(i!)<1000 do i := i+1 od:
> %;  # Display the result.
```

What would happen if you changed the colon at the end of the **od** to a semi colon?

```
>
```

The next loop does the same calculation as the last loop but uses a combination while and for-loop. Notice that there is nothing between the **do** and the **od**. This loop has no body! In the previous while-loop, the only statement in the body was the incrementing statement. But notice that in this for-while-loop, **i** is an index variable and it is initialized and incremented by the for-loop part of the for-while-loop (so there is no need for an incrementing statement in the body).

```
> for i from 1 while length(i!)<1000 do od;
> i;  # Display the result.
```

What would happen in this example if you changed the semi colon at the end of the **od** to a colon? Why?

```
>
```

Here is an example of a loop that combines while and for-in. This for-in-while-loop finds the first

number in a list.

```
> L := [a,b,c,d,e,10,9,8,7];
> for i in L while not type(i, numeric) do od;
> i;
```

As the last couple of examples showed, Maple has a number of variations on the basic idea of a loop. The three basic kinds of loops, for-loop, for-in-loop, and while-loop, can be combined to form other kinds of loops such as for-while-loop, and for-in-while-loop. There are still other forms that a loop can take, but we will not make use of them.

Exercise: Rewrite the two while-loops for finding primes as for-while-loops.

```
>
```

Exercise: Write a procedure called **`prime_bracket`** that takes as input any positive real number (not just an integer) and finds the smallest prime number larger than or equal to the input and the largest prime number smaller than or equal to the input. The procedure should return the two primes in a list. So the output of **`prime_bracket(11.5)`** will be the list **`[11,13]`**. Do *not* use the Maple functions **`nextprime`** and **`prevprime`** in your procedure.

```
>
```

Exercise: If you had to choose between writing all of your loops as for-loops or writing all of your loops as while-loops, which would you choose? Why?

```
>
```

Exercise: Does Maple have a while-in-loop?

```
>
```

Exercise: What doe the following loops tell you about how Maple executes a for-loop and a for-in-loop?

```
> L := [a, b, c]:
> for n from 1 to nops(L) do L := [n, op(L), n] od;
> L := [a, b, c]:
> for n in L do L := [n, op(L), n] od;
>
```

Exercise: From *Introduction to Maple*, 2nd Ed., by Andre Heck, page 221-222. Define the following procedure.

```
> f := proc( x, n )
>   local i, t;
>   t := 1;
>   for i from 1 to n do
>     t := x^t;
>   od;
```

```
>   t;
> end;
```

Now call this procedure a couple of times. Explain how the procedure computes these results.

```
> f( x, 22 );
> f( sin(x), 12 );
```

For fun, ask Maple to differentiate the above results.

```
> diff( f(x,7), x );
> diff( f(sin(x),12), x );
>
```

In the next section we will do some longer, more complicated examples that make use of for-loops.

```
>
```

```
>
```

14.3. More loop examples

In this section we work on more involved examples using loops.

```
>
```

14.3.1. Example 1: Riemann sums

A common use a for-loop is to compute a sum. In this example we show how to use a loop to compute Riemann sums from calculus. But first let us look at a couple of simple sums written as for-loops.

A well known result about sums of integers is that the sum of the first n positive integers is $n\,(n+1)/2$. Let us verify this for $n = 1000$ using a for-loop. We want a for-loop that will compute the following sum, written in sigma notation.

$$\sum_{i=1}^{1000} i$$

The basic idea of using a for-loop to compute a sum is that we compute a running total. If you want to add 1+2+3+4+5+6+7, we start with 0, then we add 1 to 0, then we take the result and add 2 to it, then we take that result and add 3 to it, then we take that result and add 4 to it, etc. In the following for-loop, we will let the running total be called **s** (for **s**um) and we will initialize **s** with the value 0. Notice how the single command in the body of the for-loop adds another integer to the sum for each iterate of the loop. Compare this for-loop with the sigma notation above. In what ways are they alike and in what ways do they differ.

```
> s := 0:
> for i from 1 to 1000 do s := s + i od:
```

Notice that we have a colon at the end of the loop so that we do not see 1000 lines of output. Let us check the value of **s**, the sum.

```
> s;
```

Now verify this against the formula n($n + 1$)/2.

```
> 1000*(1000+1)/2;
```

By the way, Maple can verify the above (symbolic) formula for us by using the **sum** command.

```
> i:='i': n:='n':
> sum( i, i=1..n );
> factor( % );
>
```

Exercise: Write a procedure called **add_list** that takes one input, a list of numbers, and computes the sum of the numbers in the list. Do not use Maple's **add** or **sum** commands.

```
>
```

Here is another example of computing a sum. We know from calculus that the number **e** is the sum of the reciprocals of all the factorials. So we can compute an approximation of **e** by summing the reciprocals of the factorials from say 0! to 10!. So we want a for-loop that will compute the following sum, written in sigma notation.

$$\sum_{n=0}^{10} \frac{1}{n!}$$

Compare the next for-loop with this sigma notation.

```
> e := 0;
> for n from 0 to 10 do e := e + 1/(n!) od;
```

That is not what we really want. Let us modify the loop so that it computes with decimal numbers.

```
> e := 0;
> for i from 0 to 10 do e := e + 1.0/(i!) od;
```

In this example we put a semi colon at the end of the loop so that we could see the answers converge to the correct value of **e**, which is given by the next command (to ten decimal places).

```
> evalf( exp(1) );
>
```

Now let us turn to Riemann sums. The following execution group will compute a (left hand) Riemann sum for the function **f** over the interval from **a** to **b** using **n** rectangles. If you want to, you can change any of the values for **f**, **a**, **b** or **n**.

```
> f := x -> sin(x);          # The function.
> a := 0;                    # Left hand endpoint.
> b := 2*Pi;                 # Right hand endpoint.
> n := 100;                  # How many rectangles.
> L.n := 0:                  # Set the running total to 0.
> for i from 0 to n-1 do
>    x.i := a+i*(b-a)/n;         # Compute a partition point.
>    L.n := L.n+f(x.i)*(b-a)/n   # Add area of rectangle to sum.
> od:
> 'L[n]' = %;              # Display the result symbolically,
```

```
> 'L[n]' = evalf(%%);    # and numerically
```

You should think of the for-loop in this execution group as the Maple version of the following formula from calculus.

$$L_n = \sum_{i=0}^{n-1} \mathrm{f}(x_i)\,\Delta x$$

This sigma-notation represents the sum and the for-loop in the execution group computes the sum. An interesting feature of actually computing the sum, instead of just representing it abstractly, is that the computation must be very precise about what it is doing. For example, notice how x_i in the sigma-notation is a short hand for the Maple expression **a+i*(b-a)/n**. In calculus courses where this sigma-notation is used, students often do not realize just what the x_i represents. But if you have to write a Maple for-loop to implement the sum, then you are forced to explicitly state what the symbol x_i means. Maple will not be able to figure out for you what x_i represents if you use it in a for-loop without defining it.

```
>
```

Exercise: Modify the execution group for computing L_n to compute R_n, right hand sums, and then try T_n nd S_n, the trapezoid sums and Simpson's rule.

```
>
```

```
>
```

14.3.2. Example 2: Pascal's triangle

Our next example shows how we can develop a procedure for printing out Pascal's triangle. The next for-loop prints out a table of expansions of $(x+y)^n$.

```
> x:='x': y:='y':
> for i from 0 to 8 do
>   expand( (x+y)^i );
> od;
```

The coefficients in these polynomials have a name (the binomial coefficients) and they make up what is called Pascal's triangle. Pascal's triangle and the binomial coefficients have a lot of interesting properties, so let us see if we can extract Pascal's triangle out of the last example.

The Maple command **coeffs** returns the coefficients of a polynomial as an expression sequence. The coefficients of the above polynomials make up Pascal's triangle. So let us see if the next for-loop will give us Pascal's triangle.

```
> for i from 0 to 8 do
>   coeffs( expand( (x+y)^i ) );
> od;
```

That did not work. The output from **coeffs** gives the coefficients of each polynomial as an expression sequence, but the coefficients are not in the same order as in the polynomial (see the next two commands).

```
> expand( (x+y)^8 );
```

```
> coeffs( % );
```
So we do not yet have Pascal's triangle. Let us try another approach. Since polynomials are data structures, let us use our knowledge of Maple's data structure commands to get the coefficients of each polynomial in the order we want them. Since we want the coefficients to be in an expression sequence, we will use the **seq** command. We can use an **op** command to pick off the individual terms of the polynomial, and then we can use the **coeffs** command to return the coefficient of just one term at a time.
```
> expand( (x+y)^8 );
> seq( coeffs(op(j, %)), j=1..nops(%) );
```
That worked. Here is a for-loop built around the last two commands.
```
> for i from 0 to 8 do
>    expand( (x+y)^i );
>    seq( coeffs(op(j, %)), j=1..nops(%) );
> od;
```
Oops, too much output. Put a colon at the end of the final **od** and use a **print** command in the body of the loop.
```
> for i from 0 to 8 do
>    expand( (x+y)^i );
>    seq( coeffs(op(j, %)), j=1..nops(%) );
>    print( % );
> od:
```
There we go. Notice that every number in Pascal's triangle is the sum of the two numbers just above it.

Of course, if binomial coefficients are important in mathematics, then we should expect Maple to have a command for computing them directly instead of picking them out of the expansion of $(x+y)^n$. The command **binomial(n,j)** returns the coefficient of the j'th term of $(x+y)^n$.
```
> seq( binomial(8,j), j=0..8 );
```
So we can get Pascal's triangle with the following for-loop.
```
> for i from 0 to 8 do
>   seq( binomial(i,j), j=0..i );
> od;
```
Notice how this loop uses two index variables, one for the loop itself and one for the **seq** command. The **i** index variable is counting the rows of our output and the **j** index variable is counting the "columns". And the **i** index variable from the "outer loop" is the final value for the **j** index in the "inner loop".
```
>
```

Here is a procedure that allows us to conveniently print out as many lines of Pascal's triangle as we want. (Notice the multiple levels of indentation to make this easier to read.)
```
> pascal := proc(m, n)
>   local i, j, x, y;
```

```
>   for i from m to n do
>      expand( (x+y)^i );
>      seq( coeffs(op(j, %)), j=1..nops(%) );
>      print( % );
>   od;
> end;
```

Let us try it out. (On my computer screen I can fit up to 18 lines of Pascal's triangle.)

```
> pascal(0,8);
```

Notice how this procedure uses the **print** command to print out one line of Pascal's triangle for each loop through the for-loop. Normally a procedure only has one output, which is the last line executed by the procedure (the "return value" of the procedure). But this procedure has **n** lines of output. These extra lines of output have a name, they are called side effects. Anything else that a procedure does besides returning its "return value" is called a side effect. (You may wonder just what the return value of this procedure is. We will look at that question later in this worksheet.)

```
>
```

Exercise: Modify the procedure so that it uses the **binomial** command instead of the **expand** and **coeff** commands.

```
>
```

```
>
```

14.3.3. Example 3: Periodic extensions

Our third example will use a while-loop. We show how to take an arbitrary function g defined on an interval between two numbers a and b, $a < b$, and produce a function f that is periodic on the whole real line, with period $p = b - a$, and is equal to the original function g between a and b. This new function is called the **periodic extension** of the original function.

```
>
```

Before showing how to use Maple to define the periodic extension, let us try to describe it in words. We start with what we might think of as a segment of a function g, defined just between a and b. The periodic extension f will have copies of this segment repeated over and over again on the real line. There will be one copy between b and $b + p$ (where $p = b - a$ is the length of the interval that g is originally defined on) and another copy between $b + p$ and $b + 2\,p$, etc. How should we define the extension f on the "first" interval from b to $b + p$? Visually, we would just imagine sliding the graph of g from the interval $[\,a, b\,]$ over onto the interval $[\,b, b + p\,]$. We would express this mathematically by saying that, if x is in $[\,b, b + p\,]$, then $f(x) = g(x - p)$ (make sure you understand this step). For the interval from $b + p$ to $b + 2\,p$, we would again just imagine sliding the graph of g from $[\,a, b\,]$ over to $[\,b + p, b + 2\,p\,]$. And if x is in $[\,b + p, b + 2\,p\,]$, then we would let $f(x) = g(x - 2\,p)$. In general, for any positive integer k, if x is in $[\,b + k\,p, b + (k + 1)\,p\,]$, then $f(x) = g(x - (k + 1)\,p)$. In short, to define $f(x)$ for $b < x$ we

need to keep subtracting p from x until we get a number that is between a and b, and then use that number to evaluate g. This process of subtracting p's from x until we get a number between a and b is exactly what a while-loop can do for us. Since we do not know the value of x ahead of time, we do not know how many p's we need to subtract, but while-loops are exactly what we should use when we want to iterate something an unknown number of times.

Exercise: Figure out how we would define f(x) for $x < a$.

```
>
```

Let **g** be a Maple function and let **a** and **b** be two numbers with **a<b**. Here is how we define the periodic extension **f** of **g**.

```
> f := proc(x)
>   local y;
>   y := x;
>   while y >= b do y := y-(b-a) od;
>   while y < a  do y := y+(b-a) od;
>   g(y);
> end;
```

This definition of **f** works for any **g**, **a** and **b**. Let us define specific **g**, **a**, and **b**.

```
> g := x -> x^2;
> a := -2; b:= 3;
```

Now graph the periodic extension of **g**.

```
> plot( f, -7..18, scaling=constrained, discont=true, color=red
  );
```

We should note here that if we try to graph **f** as an expression, then we get an error message.

```
> plot( f(x), x=-7..18, scaling=constrained, discont=true,
  color=red );
```

Similarly, if we try to look at the formula for **f** as an expression, we get the same error message.

```
> f(x);
```

We will find out how to fix this later in this worksheet. For now, here is a way to graph **f** as an expression if you really want to.

```
> plot('f(x)', x=-7..18, scaling=constrained, discont=true,
  color=red);
>
```

Exercise: With the same **g** as in the last example, try changing the values of **a** and **b**. How does this change the periodic extension?

Exercise: Try defining periodic extensions for some other functions **g**.

```
>
```

Exercise: How are we defining **f(a+k*p)** for any integer **k**? What other reasonable choices

are there for the value of **f(a+k*p)**?

```
> 
```

Recall that a function f is even if for all x it is true that $f(-x) = f(x)$. A function is even if its graph is symmetric with respect to the y-axis. Recall that a function is odd if for all x it is true that $f(-x) = -f(x)$. A function is odd if its graph is symmetric with respect to the origin, meaning that if we rotate the graph of f by 180 degrees, then the graph remains unchanged.

If b is a positive number and we define the periodic extension of a function g on the interval $[0, b]$, then the periodic extension will be an odd function for some g, an even function for some other g, and neither even nor odd for most g. For example, the periodic extension of $\sin(x)$ on the interval $[0, \pi]$ is an even function.

```
> g := x -> sin(x);
> a:=0; b:=Pi;
> plot( f, -3*Pi..3*Pi );
```

And the periodic extension of $\cos(x)$ on the interval $[0, \pi]$ is an odd function.

```
> g := x -> cos(x);
> a:=0; b:=Pi;
> plot( f, -2*Pi..2*Pi );
```

There is a way to define a periodic extension so that it is always even or always odd, no matter what the function g on the interval $[0, b]$ looks like. The following procedure defines the even periodic extension of a function g on the interval from 0 to b. The even extension defined by this procedure has period 2 b. (Notice the use of an anonymous function in the last line of the procedure.)

```
> f_even := proc(x)
>   local y;
>   y := x;
>   while y >= b do y := y-2*b od;
>   while y < -b do y := y+2*b od;
>   ( z -> piecewise(z<0, g(-z), g(z)) )(y);
> end;
```

The next procedure defines the odd, 2 b periodic extension of a function g on the interval from 0 to b.

```
> f_odd := proc(x)
>   local y;
>   y := x;
>   while y >= b do y := y-2*b od;
>   while y < -b do y := y+2*b od;
>   ( z -> piecewise(z<0, -g(-z), g(z)) )(y);
> end;
```

Let us try some examples using these two procedures. Here are the even and odd 2 π periodic extensions of $\sin(x)$ on the interval $[0, \pi]$.

```
> g := sin;
> b := Pi;
> plot( f_even, -3*Pi..3*Pi );
> plot( f_odd,  -3*Pi..3*Pi );
```
Here are the even and odd 2 π periodic extensions of cos(x) on the interval [0, π].
```
> g := cos;
> b := Pi;
> plot( f_even, -3*Pi..3*Pi );
> plot( f_odd,  -3*Pi..3*Pi );
```
Let us try the identity function, g(x) = x on the interval [0, 1].
```
> g := x -> x;
> b := 1;
> plot( f_even, -3..3 );
> plot( f_odd,  -3..3 );
> plot( f, -3..3 );
```
Let us try a non symmetric piece of a parabola on the interval [0, 1].
```
> g := x -> (x-1/4)^2;
> b := 1;
> plot( f_even, -3..3 );
> plot( f_odd,  -3..3, discont=true, color=red );
> plot( f, -3..3, discont=true, color=red );
>
```

Exercise: Try defining even and odd periodic extensions for some other functions **g**.
```
>
```

Exercise: Part (a) Under what conditions on the function **g** will the function **f_odd** be continuous at 0 and **b**? Under what conditions on the function **g** will the function **f_even** be continuous at 0 and **b**?
```
>
```
Part (b) Under what conditions on the function **g** will the functions **f_odd** and **f** be the same function (where **f** is the **b** periodic extension of **g** on the interval from 0 to **b**). Under what conditions on the function **g** will the functions **f_even** and **f** be the same function.
```
>
```

Here is an interesting example that uses a parabola.
```
> g := x -> 4*x*(1-x);
> b := 1;
> plot( f_odd, -4..4 );
```
Let us compare this periodic function to a sine curve.
```
> plot( [ f_odd, x->sin(Pi*x) ], -2..2 );
```
Notice how amazingly similar the two functions are. The following parametric graph uses the

odd periodic extension of **g**. Which is the real circle?

```
> plot( [ [f_odd, t->f_odd(t-1/2), 0..2],
>         [cos, sin, 0..2*Pi] ],
>       scaling=constrained );
>
```

Exercise: Given a function g defined on an interval [0, *b*], write a procedure **f_oddeven** that defines a 4 *b* periodic extension of g that is odd and such that the horizontal shift of the extension by the amount *b* is even. Similarly, write a procedure **f_evenodd** that defines a 4 *b* periodic extension of g that is even and such that the horizontal shift of the extension by the amount *b* is odd. (Hint: Think of how **sin** is built up from just one quarter of **sin**'s graph, the part from **0** to **Pi/2**, and then think of how **cos** is built up from just one quarter of **cos**'s graph.)

```
>
```

Exercise: Make up your own periodic function by extending some function **g**, and then use your periodic function in place of the trig functions sin and cos in several parametric curves and surfaces like cardiods, spirals, lemniscates, roses, sphere, torus, etc. Try to come up with a really unusual curve or surface.

```
>
```

```
>
```

14.3.4. Example 4: Drawing graphs

Our fourth example uses a for-loop and a list data structure to draw a graph. Each iterate of the for-loop will compute the coordinates of a point and put the point in a list of points that are to be plotted. Then the **plot** command will be used to draw a graph of all the points in the list. This is a fairly common way to create a graph in Maple and, as we showed in an earlier worksheet, this is fundamentally how all graphs are drawn in Maple.

The following for-loop computes 9 equally spaced sample points in the interval from 0 to 2 π. Each iterate of the loop computes one sample point, evaluates the **sin** function at the sample point, and puts the ordered pair of the sample point and the value of **sin** into a list called **data**. Then the **plot** command is used to graph the list **data**, and we get a graph of the **sin** function. Notice that **data** starts off as the empty list, **[]**, and as each new point **[x,sin(x)]** is computed, it is put into the list using a command of the form **data:=[op(data),[x,sin(x)]]**. Notice how this is very much like computing a running sum **s**, where we start off with **s** equal to the "empty" sum, 0, and then as each new term in the sum is computed it is added to the sum using a command like **s:=s+something**.

```
> N := 8;
> b := 2*Pi;
> data := [];
```

```
> for n from 0 to N do
>   data := [ op(data), [n*b/N, sin(n*b/N)] ];
> od;
```

Notice that the output from the for-loop is in symbolic form. Since we are going to just plot the contents of the list, we do not need exact symbolic results. So let us modify the for-loop so that it computes with decimal numbers. When working with loops that might create thousands of points, this could help speed things up. (If we want, we could even use the **evalhf** command and hardware floating points for even more speed.) First, we need to reinitialize **list** to be the empty list again.

```
> data := [];
> for n from 0 to N do
>   data := [ op(data), [evalf(n*b/N), sin(evalf(n*b/N))] ];
> od;
```

Now use **plot** to graph the list **data**.

```
> plot( data );
>
```

Exercise: Modify the above example so that it can be used to graph the **sin** function over any interval from **a** to **b**. (You should suppress the output from the for-loop in your example, so that you do not fill up the worksheet with an unwieldy amount of output.)

```
>
```

Exercise: Modify the above example so that it is easy to change the function that is being graphed. Try graphing the function $f(x) = 3x^2 - 2x - 1$ over the interval $[-2, 3]$.

```
>
```

Exercise: Consider the following three execution groups. Each one graphs the **sin** function and each one uses a slight variation on the last for-loop. Suppose **N** is a very large number. How do you think these three execution groups would compare, in terms of speed, with the last for-loop and with each other?

```
> N := 8;
> b := 2*Pi;
> data := []:
> for n from 0 to N do
>   data := [ op(data), [evalf(n*b/N), evalf(sin(n*b/N))] ];
> od:
> plot( data );
>
```

```
> N := 8;
> b := 2*Pi;
> data := []:
> n := 'n':
```

```
> x := evalf( n*b/N );
> for n from 0 to N do
>   data := [ op(data), [x, sin(x)] ];
> od:
> plot( data );
>
> N := 8;
> b := 2*Pi;
> data := []:
> n := 'n':
> x := evalf( b/N );
> for n from 0 to N do
>   p := n*x;
>   data := [ op(data), [p, sin(p)] ];
> od:
> plot( data );
>
```

The following for-loop computes five equally space points on the circumference of the unit circle and puts these point in a list called **data**. Notice that once again **data** starts off as the empty list, **[]**, and as each new point **[x,y]** is computed it is put into the list using a command of the form **data:=[op(data),[x,y]]**.

```
> N := 5;
> data := [];
> for n from 0 to N do
>   data := [ op(data), [cos(evalf(n*2*Pi/N)),
  sin(evalf(n*2*Pi/N))] ];
> od;
```

Notice how the list of points grew by one point with each iteration of the loop. Now use the **plot** command to plot the list.

```
> plot( data, style=point, scaling=constrained );
```

Now plot it with lines connecting the dots.

```
> plot( data, style=line, scaling=constrained );
```

In the next for-loop we use the names **x** and **y** to represent the calculation of the coordinates of each of the points. Convince yourself that the next execution group does exactly the same calculation as the previous execution group. The reason for rewriting it like this is to try and make the execution group easier to read. (And this time we are suppressing the output from the for-loop.)

```
> N := 5;
> data := [];
> n := 'n';  # be sure that n is unassigned
> theta := evalf( n*2*Pi/N );
> x := cos(theta);
```

```
> y := sin(theta);
> for n from 0 to N do
>    data := [ op(data), [x, y] ];
> od:
```

Plot the points again, to verify that it was the same calculation.

```
> plot( data, style=line, scaling=constrained );
```

Now go back to the last execution group and change the value of **N** from 5 to 7 or 9. The execution group will plot 7 or 9 equally spaced points on the circumference of the unit circle. Try any positive integer for **N**.

```
>
```

Here is a variation on this example. The next execution group plots 5 equally spaced points on the circumference of the unit circle, but they are not computed in sequential order around the circle. Try to figure out exactly what this version computes and how it does it.

```
> N := 5;
> J := 2;
> data := []:
> n := 'n':
> theta := J*2*Pi/N;
> x := cos(n*theta);
> y := sin(n*theta);
> for n from 0 to N do
>    data := [ op(data), [x, y] ];
> od:
> plot( data, style=line, scaling=constrained );
```

Try changing **N** to 7 and **J** to 3. Try several different values for **N** and **J**.

```
>
```

Exercise: Convert the last execution group into a procedure that takes two positive integer parameters, the **N** and **J**, and draws the appropriate graph. This will make it a lot easier to try out different parameter values.

```
>
```

One more variation on this example. This version has three parameters, **N**, **J**, and **K**. Try to figure out exactly what this execution group is doing. (Hint: It does pretty much the same thing as the previous version, but it does it twice per iterate of the loop.)

```
> N, J, K := 36, 21, 9;
> data := []:
> n := 'n':
> theta1 := J*2*Pi/N;
> theta2 := K*2*Pi/N;
> x1, y1 := cos(n*(theta1+theta2)),
>           sin(n*(theta1+theta2));
```

```
> x2, y2 := cos(n*(theta1+theta2)+theta1),
>           sin(n*(theta1+theta2)+theta1);
> for n from 0 to N do
>    data := [ op(data), [x1, y1], [x2, y2] ];
> od:
> plot( data, style=line, scaling=constrained, axes=none );
>
```

Here are some values for the parameters **N**, **J** and **K** that produce nice graphs.
15, 8, 13
28,19,15
39,33,27
19,13,11

```
>
```

Exercise: Part (a) Convert the last execution group into a procedure that has three positive integer parameters and draws the appropriate graph. Call your procedure with a number of different parameters.

```
>
```

Part (b): Convert your procedure from Part(a) into a procedure that takes no input parameters and generates the three integers it needs randomly. Your procedure should use the Maple function **rand** to generate the random integers (the expression **rand(a..b)()** (with *both* sets of parentheses) generates a random integer between **a** and **b**). The procedure should print out the values of the randomly chosen integers and draw the appropriate graph. Run this procedure many times. You should get some very elegant graphs.

```
>
```

```
>
```

14.3.5. Example 5: Butterfly curve

Our fifth example uses a for-loop and a data structure (a list) to draw a fairly complex graph, a version of a butterfly curve.

The curve we will draw is not in fact a curve. Instead we will be plotting points, thousands of points, and not connecting them together with line segments. The points we will be plotting will be computed by a for-loop and placed in a list (as in the last example). Then the **plot** command will be used to draw a graph of all the points in the list (without connecting them together with line segments). Almost all of the work in this example is in computing the coordinates of all the points that we want to plot.

The loop in this example is fairly computationally intensive. You should save all of your work before executing it, in case it takes too long for it to compute on your computer (do not try this on a Pentium, use at least a Pentium II). If your computer is really fast, you can try changing **N**. Try **N:=21000**, or **N:=41900** (which should run for a pretty long time). Different values of **N**

give different butterfly curves.

```
> r := phi -> ( exp(cos(phi))-2*cos(4*phi)
  )*sin(99999999*phi)^4;
> N := 11500;                    # Number of points to compute.
> h := evalf(2*Pi/N);            # Step size between points.
> n := 'n';                      # Just to be safe.
> x := r(n*h)*sin(n*h);          # x-coord of a point on the
  curve
> y := r(n*h)*cos(n*h);          # y-coord of a point on the
  curve
> data := [];                    # Start with an empty list.
> for n from 1 to N do           # This do-loop computes the
  butterfly.
>    data := [ op(data), [evalhf( x ), evalhf( y )] ]
> od:
```

Now that we have computed our list of points, let us graph it.

```
> plot( data, style=point, symbol=point, color=black );
```

The original reference for butterfly curves is *The Butterfly Curve*, by Temple H. Fay, in The American Mathematical Monthly, Volume 96, Issue 5 (May, 1989), pages 442-443. The version of the butterfly curve in this example is from *A Study in Step Size*, by Temple H. Fay, in Mathematics Magazine, Volume 70, No. 2, April 1997, pages 116-117.

```
>
```

```
>
```

14.3.6. Example 6: Animations

Our last example is the use of a for-loop to compute the frames of an animation. Earlier we created animations using the **animate** command. But the **animate** command is limited in the kind of animations it can draw. For example, it cannot animate the graphs of equations. There is another way to create animations in which we use a for-loop to create and label a sequence of graphs (the frames) and then we use a special form of the **display** command to turn the sequence of graphs into an animation. Here is an example of this technique.

First, a for-loop is used to create and name, using dotted names, 51 two-dimensional graphs, each with a parameter slightly changed. Then the **display** command is used to "sequence" the 51 graphs into a short movie. To view the movie, after the first graph is visible, place the cursor on the graph. Some VCR type buttons will appear at the top of the Maple window. Click on the "play" button. (The for-loop takes a little while to complete.)

```
> x := 'x': y := 'y':
> for i from -20 to 30 do
>   p.i := plots[implicitplot]( x^3+y^3-5*x*y = 1-i/8,
>                x=-3..3, y=-3..3, numpoints=800,
  tickmarks=[2,2] )
```

```
> od:
> plots[display]( p.(-20..30), insequence=true );
>


>
```

14.4. Conditional statements

In the previous two sections we learned how to make Maple repeat a block of statements. In this section we learn how to make Maple skip over certain blocks of statement.

Here is a simple procedure. It takes in two numbers as parameters and it returns the larger of the two numbers.

```
> bigger := proc( a, b )
>   if a >= b then a else b fi;
> end;
```

Try it out.

```
> bigger(3, 5);
> bigger(-3, -5);
>
```

The procedure **bigger** introduces another important element of programming, the **conditional statement** (also called a **if-then-else-fi statement**) . The conditional statement allows Maple to make a choice when it is computing. A conditional statement is also sometimes called a branching statement since it gives Maple a choice between two possible branches of calculations to make. Here is Maple's syntax for a conditional statement.

if *boolean_expression* **then**
sequence_of_Maple_commands
else
sequence_of_Maple_commands
fi

The part of the if-then-else-fi statement between the **if** and the **then** is called the **conditional-part** and it is either true or false. If the conditional-part is true, then Maple executes the statements between the **then** and the **else**. These statements are called the **body of the then-part**. If the conditional-part is false, then the Maple executes the statements between the **else** and the **fi**. These statements are called the **body of the else-part**. The bodies of either the then or else part can contain any number of Maple commands. The body of the else-part is optional and if it is left off, then the conditional statement has the following simpler form, which is referred to as an **if-statement**.

if *boolean_expression* **then**
sequence_of_Maple_statements
fi

If the conditional-part of an if-statement is true, then Maple executes the body of the then-part. If the

conditional-part of an if-statement is false, then Maple does not execute any statements, and it appears as if the if-statement did nothing.

The formatting of the conditional statement, with the bodies of the then and else parts indented slightly and the words **else** and **fi** on their own lines, is not part of the syntax. But the formatting makes it a lot easier to read a conditional statement and should be used most of the time.

Here are a few examples of simple conditional statements. The next command randomly generates a zero or one and if the random number is zero, the statement outputs **heads**, otherwise it outputs **tails**. Try executing this statement several times.

```
> if rand(0..1)() = 0 then
>    head
> else
>    tail
> fi;
>
```

If we remove the else-part of the if-then-else statement, so that it becomes an if-statement, then the statement will produce no output about half of the time. Try executing this statement several times.

```
> if rand(0..1)() = 0 then head fi;
>
```

The following execution group generates a long list of random integers between 1 and 10 and then it uses an if-statement inside of a for-in-loop to determine what percentage of the random integers were 10's. What answer do you expect to get?

```
> N := 1000:
> counter := 0:
> seq( rand(1..10)(), i=1..N ):
> for i in % do  # Check for 10's.
>   if i = 10 then counter := counter+1 fi
> od;
> counter/N;
> evalf( % );
```

Try executing the execution group several times. Try changing the value of **N** to 100 or 10 or 10000.

```
>
```

In the last section of this worksheet we saw that loop statements can be very useful commands to use at the Maple command prompt. The conditional statement however is of pretty limited use at the Maple prompt. Instead it is almost always used in the body of a procedure definition or in the body of a loop. Almost all of our examples of conditional statements will be in procedure bodies.

One common use of conditional statements in mathematics textbooks is in the definition of piecewise defined functions, that is functions defined by different formulas on different parts of the domain. Here are a few examples.

Suppose we wanted to represent in Maple the mathematical function defined by $x^2 + 1$ for $x < 0$ and by $\sin(\pi x)$ for $0 \le x$. Here is how we can do it using a procedure containing a conditional statement.

```
> f := proc( x )
>   if x < 0 then x^2+1 else sin(Pi*x) fi
> end;
```

Notice how there is only one boolean expression, even though there are two pieces to the function. The second piece of the function, the $\sin(\pi x)$ part, should apply whenever $0 \le x$. But we only have $0 \le x$ whenever the boolean expression **x<0** is false, which automatically puts us in the else-part of the conditional statement. So only one boolean expression is needed for a piecewise defined function with two pieces. But this is not how traditional mathematics books would typeset the definition of this function. Mathematics books almost always write out a boolean expression for each piece of the function, like this.

$$f(x) = \begin{cases} x^2 + 1 & x < 0 \\ \sin(\pi x) & 0 \le x \end{cases}$$

What appears in a mathematics book is more like the following Maple procedure.

```
> f := proc(x)
>   if x < 0 then x^2+1 else
>     if x >= 0 then sin(Pi*x) fi;
>   fi
> end;
```

This version has two boolean expressions because it has two if-statements. This is not the preferred way of defining our function in Maple for several reasons. First of all, it is not as easy to read as the previous version. With only one conditional statement we know that the else-part is mutually exclusive of the then-part. With two conditional statements it is not obvious that the two statements are mutually exclusive. A reader must carefully examine the boolean expressions to determine if they are meant to be mutually exclusive or not. Secondly, the version with one conditional is computationally more efficient than the version with two conditional statements. Every call to the second version must evaluate two boolean expressions while every call to the first version only needs to evaluate one boolean expression. This can make a difference if the function is going to be called thousands (or millions or even billions of times) of times.

Exercise: Define a function **g** similar to the last definition of **f** in such a way that the two boolean expressions are not mutually exclusive. Draw a graph of **g**. If the boolean expressions are not mutually exclusive, how does that affect the graph of **g**?

```
>
```

Let us graph our function **f** to see what it looks like.

```
> plot( f, -1..1, discont=true, color=red );
```

It is worth noting that the following command does *not* work, even though it looks perfectly OK. The function **f** is converted to an expression by evaluating it at **x**, and then the form of the **plot** command for expressions is used.

```
> plot( f(x), x=-1..1 );
```

What went wrong is that Maple is using its rule of full evaluation, so Maple tries to evaluate all of the operands in the **plot** command *before* actually calling the **plot** procedure. When Maple tries to evaluate **f(x)** there is an error, since the actual parameter **x** is an unassigned variable and the conditional statement in **f** has no way to determine if the boolean expression **x<0** is true or not.

```
> f(x);
```

The following command does work, since it prevents the evaluation of **f(x)** until the **plot** procedure actually starts sticking numbers as actual parameters into the formal parameter **x** in **f(x)**.

```
> plot( 'f(x)', x=-1..1 );
```

In an optional section later in this worksheet we will say more about this situation with **f(x)**.

```
>
```

Exercise: Here is what seems to be a reasonable definition for the function **f**.

```
> f := proc(x)
>   if x < 0 then x^2+1 fi;
>   if x >= 0 then sin(Pi*x) fi;
> end;
```

But it is not correct. Look at the graph of this version of **f**.

```
> plot( f, -1..1 );
```

What happened to the left half of **f**, the part for **x<0**? Here is a hint. What is the return value for the procedure call **f(-2)**?

```
> f(-2);
>
```

Exercise: Consider the following two functions.

```
> f := proc(x) if x>=0 then x^2 else 0 fi end;
> g := proc(x) if x>=0 then x^2 fi end;
```

Their graphs look similar.

```
> plot( f, -10..10 );
> plot( g, -10..10 );
```

But they are not the same function. Explain how they differ.

```
>
```

Here is an interesting variation on the idea of a piecewise defined function. We will create a randomly defined piecewise function. The following procedure, like the piecewise defined function above, evaluates **f** by choosing between two expressions, **x^2+1** and **sin(Pi*x)**. But instead of basing the choice of the expression on the value of the input **x**, this version bases the choice on a random number generated within the procedure.

```
> f := proc( x )
>   if rand(0..1)()=0 then x^2+1 else sin(Pi*x) fi
> end;
```

So the value of the function **f** at any point will be one of two randomly chosen numbers. Try

executing the following command several times.

```
> f(5), f(5), f(5), f(5), f(5);
```

Here is what a graph of this function might look like. Notice that each time the function is graphed, we get a different graph (why?).

```
> plot( f, -1..1 );
> plot( f, -1..1 );
>
```

Exercise: What causes the vertical bands of red in the graph? Here is a hint.

```
> plot( f, -1..1, style=point, symbol=circle, numpoints=50,
  adaptive=false );
>
```

Suppose that we want to represent in Maple a piecewise defined function with three pieces. For example, suppose we want to represent the function g defined by

$x^2 + x$ for $x \le 0$, by $\sin(x)$ for $0 < x < 3\,\pi$, and by $x^2 - 6\,x\,\pi + 9\,\pi^2 - x + 3\,\pi$ for $3\,\pi \le x$,

or, to use a notation similar to (but not exactly like) standard mathematical notation,

$$g(x) = \begin{cases} x^2 + x & x \le 0 \\ \sin(x) & x < 3\,\pi \\ x^2 - 6\,\pi\,x + 9\,\pi^2 - x + 3\,\pi & 3\,\pi \le x \end{cases}$$

Here is a procedure that computes this function.

```
> g := proc(x)
>   if x <= 0 then
>     x^2 + x
>   else
>     if x < 3*Pi then
>       sin(x)
>     else
>       x^2-6*x*Pi+9*Pi^2-x+3*Pi
>     fi
>   fi
> end;
>
```

This procedure uses a conditional statement as the body of the else-part of another conditional statement. We call these **nested conditional statements**. Notice three things. First, notice how three levels of indentation are used to help show the structure of the procedure body, in particular, the way the second conditional statement is the else-part of the first conditional statement. Second, there are only two boolean expressions even though there are three pieces of the function. The third piece of the function acts as the "default" piece and it applies whenever the first two pieces do not. Third, notice how the second boolean expression does not say `0<x<3*Pi` (as it would probably be written in a mathematics book). There are two reasons for not writing the boolean expression this way. First, it is syntactically incorrect in Maple (see the next section for more about the syntax of boolean

expressions). Secondly, it is partially redundant with the first boolean expression. Since we are in the else-part of the first (outer) conditional statement, we know that `x<=0` is false, so it must be that `x` is positive, so there is no need to have the boolean expression check again if `0<x`.

Let us plot this function.

```
> plot( g, -3..3*Pi+3 );
>
```

Exercise: Why was the right hand endpoint of this plot set to `3*Pi+3`? If we make the left hand endpoint `-4`, what would be a good choice for the right hand endpoint? What would the graph of `g` look like if we graphed it over a large domain, say from -100 to 100? Why?

```
>
```

Nested conditional statements are fairly common, so Maple has a special abbreviation for them. Here is the definition of `g` using this abbreviation.

```
> g := proc( x )
>   if x <= 0 then
>     x^2 + x
>   elif x < 3*Pi then
>     sin(x)
>   else
>     x^2-6*x*Pi+9*Pi^2-x+3*Pi
>   fi
> end;
>
```

This version of `g` has only *one* conditional statement in it (notice the single `fi` at the end of the procedure body and the use of fewer levels of indentation). There is no nesting of conditional statements in this version of `g`. Where the previous version of `g` had a conditional statement in the else-part of the outer if-then-else-fi statement, the abbreviated version has an elif-part (`elif` is an abbreviation for "`el`se `if`") followed by an else-part. This form of the conditional statement is called an **if-then-elif-then-else-fi** statement. There can be as many **elif-then** clauses as you want in an if-then-elif-then-else-fi statement. For example, if we want to represent a piecewise defined function with four pieces in its definition, then we can use two elif-then clauses (in a single if-then-elif-then-else statement).

```
> h := proc( x )
>   if x <= 1 then
>     x
>   elif x <= 2 then
>     x^2
>   elif x <= 3 then
>     6-x
>   else
```

```
>     x^2-6*x+12
>   fi
> end;
>
```

When you read a conditional statement like this, it is important to remember that the boolean expressions are "cumulative". So for example, the **x^2** part is not used just when **x<=2** is true as the **elif** clause just before it might seem to imply. The **x^2** part is only used when **x<=2** is true *and* **x<=1** is false. The test for **x<=2** only comes after the test for **x<=1** fails. Similarly, the test for **x<=3** only comes after both **x<=1** fails and **x<=2** fails. And the **x^2-6*x+12** part is used only if each of the three tests **x<=1**, **x<=2**, and **x<=3** all fail.

```
>
```

Let us look at graphs of these last two functions.

```
> plot( g, -3..3*Pi+3 );
> plot( h, -1..5 );
>
```

The syntax for a **if-then-elif-then-else-fi** statement should be pretty clear by now.

if boolean_expression **then**
  sequence_of_Maple_commands
elif *boolean expression* **then**
  sequence_of_Maple_commands
else
  sequence_of_Maple_commands
fi

Remember that there can be as many elif-then clauses as needed in this form of the conditional statement.

```
>
```

The next example shows how we might need two if-then-else-fi statements nested inside of an if-then-else-fi statement, one in each of the then and else parts. This procedure finds the largest of three numbers.

```
> bigger3 := proc( a, b, c )
>   if a >= b then
>     if a >= c then a else b fi;
>   else
>     if b >= c then b else c fi;
>   fi;
> end;
```

Try this procedure out.

```
> bigger3( 1, 2, 3 );
> bigger3( 3, 2, 1 );
```

```
> bigger3( 1, 3, 2 );
>
```
This procedure could use the **elif** abbreviation for one of the nested conditional statements, but that probably would not make the procedure any easier to understand.
```
>
```

Exercise: Rewrite the procedure **bigger3** using an elif-clause in the outer conditional statement.
```
>
```

There is another way to implement **bigger3**. This version uses nested calls to our procedure **bigger** instead of nested conditional statements. (Recall that **bigger** was defined at the very beginning of this section.)
```
> bigger3 := proc(a, b, c)
>    bigger( bigger(a, b), c );
> end;
```
This is a common technique in programming. Use one procedure as part of the definition of another procedure. If we compare the two versions of **bigger3**, notice how the inner call to **bigger** plays the same role as the outer conditional statement, and the outer call to **bigger** plays the role of *both* the inner conditional statements.
```
> bigger3(-3, 4, 2);
>
```

We can also create a procedure **bigger4** that finds the largest of four numbers. We can do this two ways, one way using nested conditional statements and another way using nested procedure calls to **bigger3** and/or **bigger**. The nested conditional statement version of **bigger4** will be quite messy but you should try writing it. Here are several ways of writing **bigger4** using nested procedure calls.
```
> bigger4 := proc(a, b, c, d)
>    bigger( bigger(a, b), bigger(c, d) );
> end;
```

```
> bigger4 := proc(a, b, c, d)
>    bigger( bigger( bigger(a, b), c ), d);
> end;
```

```
> bigger4 := proc(a, b, c, d)
>    bigger( bigger3(a, b, c), d );
> end;
```
There are still several other ways of making **bigger4** out of **bigger** and **bigger3**. What are they?
```
>
```

Exercise: Write a version of **bigger4** that uses only nested conditional statements.

```
> 
```

Exercise: Write a version of **bigger4** that uses procedure calls nested inside of conditional statements.

```
> 
```

For the sake of completeness, what about finding the maximum of an arbitrary number of numbers. This is like our problem of finding the average of an arbitrary number of numbers. We solve it by using a list (which is a data structure) as the input to our procedure.

```
> biggest := proc( list::list(numeric) ) # Type check the input.
>   local candidate, i;
>   candidate := list[1];  # Make an initial guess.
>   for i from 2 to nops(list) do
>     if list[i] > candidate then
>       candidate := list[i]; # Update our guess.
>     fi;
>   od;
>   candidate;  # Return our final answer.
> end;
```

Now try it out.

```
> biggest( [1,2,3,4,5,100,6] );
```

Let us test our procedure on a randomly generated list of integers.

```
> random_list_of_integers := [ seq( rand(), i=1..12 ) ];
> biggest( random_list_of_integers );
> 
```

Exercise: Change the definition of **biggest** to use a call to procedure **bigger** in place of the if-statement.

```
> 
```

Exercise: If you read the optional section on the special local variables **args** and **nargs**, then write a version of **biggest** that does not need the brackets in its procedure call.

```
> 
```

```
> 
```

14.5. Boolean expressions

Boolean expression are used mostly with while-loops and conditional statements. In this section we will look at more examples that use boolean expressions and we will define the syntax for boolean expressions.

Recall our first definition of **bigger3**.

```
> bigger3 := proc( a, b, c )
>   if a >= b then
>     if a >= c then a else b fi;
>   else
>     if b >= c then b else c fi;
>   fi;
> end;
```

Here is another way to write it.

```
> bigger3 := proc( a, b, c )
>   if a >= b and a >= c then
>     a;
>   elif b >= a and b >= c then
>     b;
>   else
>     c;
>   fi;
> end;
```

Try this new version out.

```
> bigger3(1, 2, 3);
> bigger3(3, 2, 1);
> bigger3(1, 3, 2);
```

The new version uses a single conditional statement (with an elif-part) and it uses a more sophisticated kind of boolean expression. Let us define several terms used with boolean expression.

```
>
```

Boolean expressions are expressions that evaluate to be either **true** or **false**. We will say that **true** and **false** are the two possible **boolean values**. Boolean expressions can be contrasted with arithmetic (or algebraic) expressions which are expressions that evaluate to a number. Just as arithmetic expressions are made up of basic arithmetic operators (e.g., **+**, **-**, *****, **/**, **^**) and functions that return a number (i.e., real valued functions), **boolean expressions** are made up of the three **logical operators** (**and**, **or**, **not**), the **relational operators** (**<**, **<=**, **>**, **>=**, **=**, **<>**), and functions that return **true** or **false** (i.e., **boolean functions**). We have already mentioned the relational operators and we have seen examples of using boolean functions like **isprime**. But we have not mentioned the basic logical operators **and**, **or** ,and **not**, so we will go over them now.

We call **and** and **or** **binary boolean operators** and we call **not** a **unary boolean operator**. A binary operator is an operator that acts on two operands (like **+**, **-**, *****, **/**, and **^**) with one operand placed on either side of the operator. But **+**, **-**, *****, **/**, and **^** are arithmetic operators that operate on numbers, and the result that they return is a number. The operators **and** and **or** operate on boolean values and return a boolean value, so they are boolean operators. A unary operator acts on only one operand which is sometimes placed before the operator and sometimes placed after the operator

(placing the operand after the unary operator symbol is more common). A common example of a unary operator is the arithmetic operator `-` for negation (not to be confused with the binary operator `-` for subtraction) which is place before the operand. Another unary operator is the factorial operator `!` which is placed after its operand. The `not` operator is placed before its boolean operand.

```
> 
```

Since boolean operators operate on boolean values, and there are only two boolean values, it is possible to make a table of a boolean operator's value for every possible input. We call these **truth tables**. Here is the truth table for `or`.

```
   x      y    x or y
false | false | false
false | true  | true
 true | false | true
 true | true  | true
```

Here is the truth table for `and`.

```
   x      y    x and y
false | false | false
false | true  | false
 true | false | false
 true | true  | true
```

Here is the truth table for `not`.

```
   x    not x
false |  true
 true |  false
```

We can verify some of these entries with simple Maple commands.

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

Exercise: Here are several examples of boolean expressions. Explain their values.

```
> isprime(7) and not 7<5;
> member(d, [a, b, [c,d]]) or a=b;
> has([a, b, [c,d]], d) and 7<5;
> (1/3 + 1/3 = 2/3) and type(a+b, algebraic);
> true <> false or 0!=1;
> not(x=y) and 0>-infinity;
```

Note: The parentheses in the last example are not part of a function call since `not` is not a function. The parentheses are for grouping, and `not(x=y)` should be thought of like the arithmetic expression `-(x+y)` (recall that both `not` and negation are unary operators).

```
> 
```

Exercise: The expression `-(x+y)` is the same as `(-x)+(-y)`. Is `not(x=y)` the same as `(not x)=(not y)`?

```
> 
```

Exercise: Give examples of some other binary operators in Maple. Give examples of some other unary operators. For your examples, specify the data type of the operands and the return value of the operator.

```
> 
```

Notice that the following is not a boolean expression in Maple, though it would be a true statement in a mathematics book.

```
> 3 <= 4 <= 5;
```

Here is how the last example should be expressed in Maple.

```
> 3 <= 4 and 4 <= 5;
```

Similarly we cannot say the following in Maple.

```
> 1 = 0! = 1!;
```

Instead we must put it this way.

```
> 1 = 0! and 0! = 1!;
```

This last result may seem surprising. We were expecting to get `true`. For example, if we modify the expression just a bit, then it does return `true`.

```
> 1 = 0! and 2 = 2!;
> 
```

Let us analyze how Maple evaluates the boolean expression `1=0! and 0!=1!`, since it brings up an important fact about equations and boolean expressions. Since `0!` evaluates to `1`, Maple evaluates (using the rule of full evaluation) the equation `1=0!` to be the equation `1=1`. Similarly for `0!=1!`.

```
> 1=0!;
> 0!=1!;
```

So the expression `1=0! and 0!=1!` evaluates to `1=1 and 1=1`. Then Maple automatically simplifies the expression `1=1 and 1=1` to be `1=1` (the expression `x and x` automatically simplifies to `x` for any expression `x` (why?)).

```
> x and x;
> 1=1 and 1=1;
```

But why does the equation `1=1` not evaluate to `true`, since it certainly is true? The reason has to do with a dual role that equations (and inequalities) play in Maple. Equations are used both as boolean expressions and as algebraic equations.

```
> type(1=1, boolean);
> type(1=1, equation);
```

For example, it would be very inconvenient if Maple would take an equation like the following and automatically evaluate it as a boolean expression.

```
> a*x^2+b*x+c=0;
```

Of course, the above equation is a boolean expression, but that is usually not what we have in mind when we type it in. Here is how Maple would evaluate it as a boolean expression.

```
> evalb( % );
```

Since equations in their role as algebraic equations are so common, Maple has a rule that it will not evaluate an isolated equation (or inequality) as a boolean expression unless explicitly told to do so by using the **evalb** command. But if an equation (or inequality) is part of a larger boolean expression, or contained in the boolean part of a while-loop or conditional statement, then Maple will automatically evaluate the equation (or inequality) as a boolean expression.

So now we know why Maple evaluates the expression **1=0! and 2=2!** as **true** but it evaluates the similar expression **1=0! and 0!=1!** as **1=1**.

```
> 1=0! and 2=2!;
> 1=0! and 0!=1!;
```

Even though the expression **1=0! and 0!=1!** starts out as a boolean expression, it automatically simplifies to an equation, and Maple will not evaluate this isolated equation as a boolean expression unless we explicitly tell it to do so by using the **evalb** command.

```
> 'evalb'( 1=0! and 0!=1! );
> %;
>
```

Parentheses play an important role in boolean expressions just as they do in arithmetic expressions. For example, the following two expressions are equivalent.

```
> x = y and z;
> (x = y) and z;
```

But they are not equivalent to the next expression.

```
> x = (y and z);
```

(Why did Maple not evaluate this last expression as true or false?)

There is a whole algebra to boolean expressions that specifies the order of precedence for all of the boolean operations, associativity rules for each boolean operation, and algebraic identities for boolean expressions. For example, here are some automatic simplifications that Maple knows for the algebra of boolean expressions.

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

The last two simplifications are known as DeMorgan's Rules.

```
>
```

Most of the boolean expressions that we need for conditional statements and while-loops are formed

using simple combinations of **and**, **or**, **not**, the relational operators, and a few boolean functions like **type**, **isprime**, **has**, and **member**. So we will not got into any more detail about the algebra of boolean expressions.

```
>
```

```
>
```

14.6. For-loop like commands

The three commands **seq**, **add**, and **prod** act very much like for-loops. In a sense they are abbreviations of special purpose for-loops. Let us look at a few examples of each of these commands

We have seen the **seq** command before. It is used to create expression sequences. Here is an example.

```
> seq( ifactor(n), n=1..10 );
```

Compare this last command to the following for-loop.

```
> for n from 1 to 10 do ifactor( n ) od;
```

Both commands did roughly the same thing. An index variable **n** was incremented, in steps of 1, from an initial value of **1** to a final value of **10** and for each value of the index variable the procedure **ifactor(n)** was evaluated. The main difference between the two commands is that the **seq** command produced one result, an expression sequence, but the for-loop produced 10 separate results. Here is a way to rewrite the for-loop so that it produces an expression sequence.

```
> result := NULL:  # Start with an empty exprseq.
> for n from 1 to 10 do
>   result := result, ifactor(n) # Append an operand to the
  exprseq.
> od:
> result;  # Show the final exprseq.
```

Except for the fact that this execution group needed to use an extra variable, the execution group produces the same result as the above **seq** command.

```
>
```

Here is another example. This produces, more or less, one line of Pascal's triangle.

```
> seq( op(1,n), n=expand((a+b)^12) );
```

In this case, instead of the index variable counting from an initial value to a final value, the index variable steps through the operands of a data structure. The following for-in-loop produces the same expression sequence as the **seq** command.

```
> result := NULL:
> for n in expand( (a+b)^12 ) do
>   result := result, op(1,n)
> od:
> result;
```

Notice how much more clear it is to use the **seq** command. Besides being easier to read and write, the **seq** command is also more computationally efficient than the equivalent for-loop. The for-loop produces a lot of intermediate results (which we hide by using a colon) but the **seq** command is implemented in a way that avoids all the intermediate expression sequences.

```
> 
```

Now let us turn to the **add** command. The following command will add up the first ten squares.

```
> add( n^2, n=1..10 );
```

This can also be done with a for-loop.

```
> result := 0:
> for n from 1 to 10 do
>   result := result + n^2
> od:
> result;
```

The main difference between the two is that the for-loop needed an extra variable and it produced a lot of intermediate results. The **add** command is also faster. Try adding up the first one million squares using first the **add** command and then the for-loop.

```
> 
```

Recall that earlier we said that for-loops can be used to implement the sigma notation used in mathematics. The **add** command is a direct analogue in Maple to sigma notation. An **add** command of the form

```
add( f(n), n=a..b )
```

means exactly the same thing as

$$\sum_{n=a}^{b} f(n)$$

and their for-loop equivalent is

```
result := 0:
for n from a to b do
  result := result + f(n)
od:
result;
```

By the way, look at the interesting output from these nested "loop" commands. They sum up the first 10 squares, then the first 100 squares, then the first 1000 squares, etc. Can you explain the pattern?

```
> for k from 1 to 6 do
>   add( n^2, n=1..10^k )
> od;
```

The **mul** command is much like the **add** command, it just uses multiplication instead of addition. So the following command will find the product of the first ten squares.

```
> mul( n^2, n=1..10 );
```
Here is the equivalent for-loop.
```
> result := 1:
> for n from 1 to 10 do
>   result := result * n^2
> od:
> result;
>
```
The **mul** command is a direct analogue in Maple to mathematical product notation. A **mul** command of the form

```
mul( f(n), n=a..b )
```

means exactly the same thing as the standard mathematical product notation

$$\prod_{n=a}^{b} \mathrm{f}(n)$$

and their for-loop equivalent is

```
result := 1:
for n from a to b do
  result := result * f(n)
od:
result;
```

There are two Maple commands that are related to the **add** and **mul** command but they are not abbreviations for for-loops. They are the **sum** and **product** commands. These two commands do *symbolic summation* and *symbolic multiplication*. For example, consider the next command, which will sum up the first **j** squares.
```
> sum( 'n^2', 'n'=1..j );
```
This command gave us a symbolic answer for the sum of the first **j** squares for *any* value of **j**. The result from the **sum** command can be simplified quite a bit.
```
> simplify( % );
> factor( % );
```
The **sum** command can even sum up infinite series. Here is a geometric series.
```
> sum( 'r^n', 'n'=1..infinity );
```
The product command does much the same thing for products.
```
> j:='j':
> product( 'n^2', 'n'=1..j );
```
The product of the first **j** squares is given symbolically in terms of a special function, Γ, called the gamma function. Let us test this result with **j** equal to 99.
```
> mul( n^2, n=1..99 );
> GAMMA(99+1)^2;
```
The **product** command can also do infinite products. The following command uses the "inert" form of the **product** command (i.e. **Product**) to display a typeset version of the product

notation on the left hand side of an equal sign, and on the right of the equal sign is the regular **product** command to evaluate the infinite product.

```
> Product( 1-1/(4*'n'^2), 'n'=1..infinity )
>        = product( 1-1/(4*'n'^2), 'n'=1..infinity );
>
```

Here is one tricky difference between **add**, **mul** on the one hand and **sum**, **product** on the other hand. Let us give the variable **i** a value (it does not matter what value).

```
> i := 0;
```

Now use **i** as the index variable in an **add** and a **mul** command.

```
> add(i, i=1..10);
> mul(i, i=1..10);
```

Let us check the value of **i** now.

```
> i;
```

It is still zero. The value of **i** did not affect, and was not affect by, the index **i** in the **add** or **mul** commands. The index variable in an **add** or a **mul** command is local to that command and does not have anything to do with the global variable with the same name (just like local variables in procedures). Now try the **sum** and **product** commands.

```
> sum( i, i=1..10 );
> product( i, i=1..10 );
```

The index variable in the **sum** and **product** commands are global variables. Here is how to fix the last two commands.

```
> sum( 'i', 'i'=1..10 );
> product( 'i', 'i'=1..10 );
```

Now check the value of **i**.

```
> i;
```

Strangely enough, even though the index variable in the **add** and **product** commands is the global variable **i**, **i** still retains its value from before the commands were executed.

The moral of this is that you need to be more careful when using **sum** and **product** than when you use **add** or **mul**. Always use **add** and **mul** if you do not need the extra abilities of **sum** and **product**.

```
>
```

Exercise: Compare the way Maple handle the "index variable" in **add**, **mul**, **sum**, and **product** with the way Maple handles the index variable in a for-loop.

```
>
```

```
>
```

14.7. Statements vs. expressions (optional)

Suppose we have a list **L** of numbers and we want to compute their average. Here is how we might

do this.

```
> L := [2,3,4,5,6,7,8];
> total := 0:
> for i in L do total := total+i od:
> avg := %/nops(L);
```

Here is something that you cannot do in Maple even though it seems perfectly reasonable. The last result variable (%) in the fourth line of the execution group holds the result of the for-loop from the third line. Why not save a line and put the for-loop directly in the numerator of the expression that computes the average?

```
> L := [2,3,4,5,6,7,8];
> total := 0:
> avg := (for i in L do total := total+i od)/nops(L);
```

Maple does not like that. Why not? The reason is that we cannot use the for-loop (which is a statement) as if it were an expression.

```
>
```

Maple makes a distinction between expressions and statements. For the most part in Maple, you cannot use statements where you are supposed to use expressions. For example, you cannot put a statement after an assignment operator and you cannot put statements in expression sequences.

```
> x := y := 5;
> y := if x<0 then -x else x fi;
> 0, x:=1, y:=2, 3;
```

The most common kinds of statements in Maple are repetition statements, conditional statements, assignment statements, and the **restart** statement. Almost everything else in Maple is an expression (and every expression is also a statement).

```
>
```

Here is another example of how Maple treats statements and expressions differently. It also shows how the rules about where you can use a statement are not very clear. Suppose we want to represent in Maple the mathematical function that is equal to $\sin(x)$ for x less than 0 but is equal to $\sin(\pi x)$ for x greater than 0. Here is one way that we can do this.

```
> f := x -> if x<0 then sin(x) else sin(Pi*x) fi;
```

Here we used the arrow notation with a conditional statement in the body of the Maple function. On the other hand we could also do the following.

```
> g := x -> `if`(x<0, sin(x), sin(Pi*x));
```

Here we used the arrow notation with a conditional *operator* in the body of the Maple function. The **conditional operator** is an expression version of the conditional statement. So **f** and **g** are defined using a statement and an expression, respectively, after the arrow. Notice how Maple seems to store the definition of these two functions differently.

```
> eval( f );
> eval( g );
```

Now here is an arrow operator with a repetition statement after the arrow.

```
> h := n -> for n from n while not isprime(n) do n+1 od;
```
So Maple allowed a conditional statement after the arrow but not a repetition statement. The function **h** was meant to return the first prime after the positive integer **n** as the next execution group does.
```
> n := 10010:
> for n from n while not isprime(n) do n+1 od:
> %;
>
```

Exercise: Rewrite **h** as a procedure.
```
>
```

```
>
```

14.8. Print levels, printlevel, and print commands (optional)

When you place for-loops inside if-statements or if-statements inside for-loops, things do not work out quite the way you might expect them too. Here is an example.
```
> if 1=1 then
>   for i from 0 to 10 do i od
> else
>   for i from -10 to 0 do i od
> fi;
```
There was no output from the command. But we know that the first for-loop was executed. The problem is that Maple has a notion of **print levels**. A for-loop inside of an if-statement is at the second print level. By default, Maple only prints out results from commands at the first print level. There are two ways to solve this problem. The easiest is to use the **print** command, which always prints no matter what print level it is at.
```
> if 1=1 then
>   for i from 0 to 10 do print(i) od
> else
>   for i from -10 to 0 do print(i) od
> fi;
```
Now go back and try changing **1=1** to **1=0**.
```
>
```

The other solution is to change the value of the **printlevel** variable. Right now the **printlevel** variable has the value 1, which is its default value. So Maple only prints out the results of commands at the first print level.
```
> printlevel;
```
The following command tells Maple to print the results of all commands at both the first and second print levels.
```
> printlevel := 2;
```

Let us try it out.

```
> if 1=1 then
>   for i from 0 to 10 do i od
> else
>   for i from -10 to 0 do i od
> fi;
```

Try changing **printlevel** back to 1 and executing the last if-statement again.

```
> printlevel := 1;
>
```

The same thing happens for if-statements inside of for-loops. (Try this with **printlevel** equal to 1.)

```
> for i from 0 to 10 do
>    if i >= 5 then i fi
> od;
```

How many lines of output should this command have produced? To see the output we can either use a **print** command inside the if-statement, or set **printlevel** higher than 1. Here is the version using the **print** command.

```
> for i from 0 to 10 do
>    if i >= 5 then print(i) fi
> od;
```

Now set **printlevel** to 2.

```
> printlevel := 2;
```

Re-execute the for-loop (without the **print** command).

```
> for i from 0 to 10 do
>    if i >= 5 then i fi
> od;
>
```

Notice that with **printlevel** equal to 2, there is still no output from the following command.

```
> if 1=0 then
>   for i from 1 to 5 do
>      for j from 1 to i do j od
>   od
> else
>   for i from -5 to 0 do
>      for j from -5 to i do j od
>   od
> fi;
```

Since this command had two nested for-loop nested inside of a conditional statement, we need **printlevel** equal to 3 to be able to see the output (or we could use **print** commands).

```
> printlevel := 3;
```

Now go back and re-execute the last if-statement to see its output.

```
> 
```

Changing the value of **printlevel** and using **print** commands seem to be equivalent ways to solve the problem of seeing your results when you have for-loops and conditional statements inside of each other. But these two techniques for seeing your results are *not* equivalent. There is a subtle difference. Consider the following example.

```
> if 1=1 then for i from 1 to 2 do (x+y)^i od fi;
```

This command did not print out a result but it did produce a result.

```
> %;
```

Now use a print command to see the output of the for-loop (notice that the loop now goes from **1** to **4**).

```
> if 1=1 then for i from 1 to 4 do print( (x+y)^i ) od fi;
```

Now what is the value of the last result variable **%**?

```
> %;
```

Why does **%** have the value $(x+y)^2$ and not $(x+y)^4$? Let us do a small experiment.

```
> 1 + 1;
> if 1=1 then for i from 1 to 4 do print( (x+y)^i ) od fi;
> %;
```

The loop had no effect on the last result variable. The loop was executing **print** commands. Does the **print** command change the last result variable?

```
> print( "hello" );
> %;
```

So the **print** command itself does not have any effect on the last result variable. Now we see why changing the value of **printlevel** is not the same as using **print** commands. How the loop effects the last result variable is different if we use **print** commands.

```
> 
```

Here is another example of the difference between using **print** commands and changing the value of **printlevel**. Recall that the following loop prints out Pascal's triangle.

```
> for i from 0 to 8 do
>   seq( binomial(i, j), j=0..i );
> od;
```

Let us put this loop inside the body of a procedure.

```
> pascal := proc(n)
>   local i, j;
>   for i from 0 to n do
>     seq( binomial(i, j), j=0..i );
>   od;
> end;
```

Now let us try our procedure.

```
> pascal(8);
```

What happened? We only got the last line of Pascal's triangle. Recall that a procedure has a return value and the return value is the last command executed in the body of the procedure. So the last line of Pascal's triangle was the return value of the procedure call and the other lines of Pascal's triangle were computed but thrown away. To see all of Pascal's triangle we could use a higher value of **printlevel**.

```
> printlevel := 6;
> pascal(8);
```

Notice that the procedure call produced a lot of output, but the return value of the procedure call is still the last result computed.

```
> %;
```

But putting **printlevel** this high can be very inconvenient since we will be getting a lot of unwanted output now from a lot of the Maple commands. So lets us return **printlevel** to its normal value.

```
> printlevel := 1;
```

A better way to see all of Pascal's triangle is to use a **print** command in the body of the procedure.

```
> pascal := proc(n)
>   local i, j;
>   for i from 0 to n do
>     seq( binomial(i, j), j=0..i );
>     print ( % );
>   od;
> end;
```

Now lets us try this version.

```
> pascal(8);
```

Now here is an interesting question? What was the return value of this procedure call? Was it the last line of Pascal's triangle? The last result variable (i.e. **%**) should hold the return value for us at this point.

```
> %;
```

Why did we get that? Let us try an experiment.

```
> 2 + 2;
> pascal(3);
> %;
```

The procedure call had no effect on the last result variable. Did the procedure have a return value? Let us try another experiment.

```
> xxx := pascal(3);
> xxx;
> assigned( xxx );
```

The procedure did seem to have some kind of return value, but it seems to be a value without a value. The return value of a procedure is the last command in the procedure body and in the case of this version of **pascal** the last command would be the **print** command. What is the return value of a **print** command? Is it what the command prints?

```
> whattype( print("hello") );
```

The return type of `print` is exprseq, so its return value is not what it prints out (which has type `string`). So what is the return value of the `print` command? The `print` command returns a special result called `NULL`, which is actually a name for the empty expression sequence.

```
> whattype( NULL );
> evalb( NULL = op( [] ) );
> evalb( print("hello") = NULL );
```

The last result variable, `%`, has the property that it always ignores the result `NULL`. This is why the `print` command did not have any effect on the last result variable. Here is another example of print returning `NULL`.

```
> nothing_there := [ print('x'), print('y'), print('z') ];
```

The list on the right hand side of the assignment operator evaluates to the empty list because the value of each `print` command is `NULL` (the empty expression sequence).

```
>
```

Here is something that might seem confusing. If a command like `print("hello")` returns the value `NULL` but it also prints out `"hello"`, then what do we call the printed out `"hello"`? The printout from the `print` procedure is a special case of what is called a **side effect**. A side effect is any result from a procedure other than its return value. Notice that the phrase "output from a command" is, unfortunately, kind of vague. Sometimes it refers to the value returned by a command, and sometimes it refers to a side effect of a command. (And some "commands" are procedures while other "commands" are statements, which further confuses the idea of the "output from a command".)

```
>
```

So putting `print` commands in a procedure can change what the return value of the procedure is. On the other hand, changing the value of `printlevel` does not have any effect on the return value of a procedure (but it can produce a lot of unwanted "output"). So there is a difference between using `print` commands and changing `printlevel`.

```
>
```

```
>
```

14.9. Procedures that return unevaluated or return NULL (optional)

In the section on conditional statements, we defined procedures that used if-then-else-fi statements to implement piecewise defined functions. But there were some subtle problems with those procedures that we left unresolved in that section. Here is an example.

```
> f := proc(x) if x < 0 then x else x^2 fi end;
```

Here are two problems with this function definition. First, if we convert this function into an expression `f(x)`, then we cannot graph the expression the way we would expect to.

```
> plot( f(x), x=-1..1 );
```

This problem is pretty easy to avoid. Either do not use an expression to graph the function, or use

right-quotes to delay the evaluation of the expression in the **plot** command.

```
> plot( f, -1..1 );
> plot( 'f(x)', x=-1..1 );
```

But both of these solutions are avoiding a deeper problem. Here is an example of the deeper problem. Suppose we wish to work with our function **f** in some kind of symbolic way, such as working with its difference quotient.

```
> (f(x+h)-f(x))/h;
```

We cannot work with **f** in any kind of symbolic way.

```
> f(x+h); f(x);
```

One of the whole points of Maple is to be able to work with functions symbolically, so this is a serious problem. The solution to this problem uses a subtle and important technique that is referred to as **a procedure returning unevaluated**. Here is the correct way to define the procedure **f**.

```
> f := proc(x)
>   if type(x, numeric) then
>     if x < 0 then x else x^2 fi
>   else
>     'f'(x)
>   fi
> end;
```

This version of **f** uses type checking to determine if the input to **f** is some kind of number. If the input is not a number, then it must be some kind of symbolic input, so in that case the function just returns itself "unevaluated", i.e., **'f'(x)**. On the other hand, if the input is some kind of number, then the function is evaluated just as before. Here are some examples.

```
> (f(x+h)-f(x))/h;
```

The next example is partly a symbolic use of **f** and partly a numeric use of **f**.

```
> (f(2+h)-f(2))/h;
```

Now we can graph **f** as an expression the way we expect to be able to.

```
> plot( f(x), x=-1..1 );
```

The situation with **f** is now far better than it was before. The new version of **f** behaves pretty much the way we would expect a function to behave symbolically.

```
>
```

Whenever a function is defined using a procedure, especially if the procedure uses a conditional statement (or while loop), the procedure body should begin with a type check on the input to see if it is **numeric**, and if not, return the function call unevaluated.

```
>
```

Here is another, slightly different example of a function returning unevaluated. Consider the following three procedures. Each one is an implementation of the mathematical function 1/*x*. Each procedure uses the technique we just went over in the last example. The three procedure differ only in how they react to an input of 0.

```
> f := proc(x)
```

```
>   if type(x, numeric) then
>     if x<0 then
>       1/x
>     elif x>0 then
>       1/x
>     elif x=0 then
>       ERROR("0 is a singularity.")
>     fi
>   else
>     'f'(x)
>   fi
> end;
```

```
> g := proc(x)
>   if type(x, numeric) then
>     if x<0 then
>       1/x
>     elif x>0 then
>       1/x
>     fi
>   else
>     'g'(x)
>   fi
> end;
```

```
> h :=  proc(x)
>   if type(x, numeric) then
>     if x<0 then
>       1/x
>     elif x>0 then
>       1/x
>     elif x=0 then
>       'h'(0)
>     fi
>   else
>     'h'(x)
>   fi
> end;
```

Each procedure demonstrates a different technique for dealing with an invalid input. Let us try each procedure. The first one, **f**, outputs an error message when the input is 0.

```
> f(0);
```

The second example, **g**, just ignores the input 0 and does not return anything (be sure to look over the definition of **g** to see how it ignores 0).

```
> g(0);
```
The third example, **h**, returns an unevaluated function call when the input is 0.
```
> h(0);
```
So we see three different ways to deal with "invalid" inputs to a function, and one of them uses a function that returns unevaluated. Of the three methods, the first and third are both reasonable. Sometimes one is used and sometimes the other is used.
```
> ln(0);
> ln(-2);
```
The reason for returning **ln(-2)** unevaluated is that it can, under certain circumstances, be given a value (though in calculus books it is considered undefined). Similarly, the function **h** returns **h(0)** unevaluated since it is very possible that we may later define a value for **h** at 0 (we are free to define a function any way we want). Mathematically, we can say that the difference between **f** and **h** is that **f** is defined such that 0 is not in its domain, and **h** is defined so that 0 is in its domain, but we have not yet specified what its value is there.
```
>
```

The way that **g** handles the input 0 is not, however, a very good idea. Like **f**, **g** implements a function for which 0 is not in its domain. But the way **g** is defined can lead to some pretty mysterious error messages. Here is an example.
```
> k := h + g;
> k(0);
```
Compare that with using **f** in place of **g**.
```
> k := h + f;
> k(0);
```
At least this error message makes some sense. So what is **g**'s error message referring to? Consider the following.
```
> h(0) + NULL;
```
When Maple tried to evaluate **k(0)** using **g**, first it got **h(0)+g(0)**, and then this evaluated to **h(0)+NULL**, and we just saw that this caused the error message. So why did **g(0)** evaluate to **NULL** and what is **NULL**? **NULL** is the special symbol for the empty expression sequence. The empty expression sequence is Maple's idea of nothing, and **g** returned nothing for the input 0, so Maple considers **NULL** to be the value of **g** at 0. But **NULL** cannot be added to anything, hence the error message. So it is not really a good idea for a function to return nothing (i.e., **NULL**) for inputs that are not in its domain. An explicit error message is preferable.

But there are times when Maple commands will return nothing. Here is an example of a Maple command that returns a **NULL** value.
```
> solve( sin(x)^2+x^5=0, x );
```
The **solve** command could not find a (symbolic) solution so it returned nothing. It can be argued, from a user interface point of view, that this is another example where returning nothing is not a good idea and that an explicit message would have been better, but at least it shows that Maple does have commands that can return **NULL**.

```
> 
```

Here is another example of a command that returns **NULL**. The **print** function produces an output but its output is not its value (the output is a side effect).

```
> 1 + print(5);
```

The **print** command outputted the 5 but 5 was not its value. The **print** command's value was **NULL** which is why we see that same error message again from the sum.

```
> 
```

Finally, here are two exercises from the section on conditional statements that we reproduce here because you should now be able to give them a more detailed solution.

Exercise: Here is what seems to be a reasonable definition for the function **f**.

```
> f := proc(x)
>   if x < 0 then x^2+1 fi;
>   if x >= 0 then sin(Pi*x) fi;
> end;
```

But it is not correct. Look at the graph of this version of **f**.

```
> plot(f, -1..1);
```

What happened to the left half of **f**, the part for **x<0**? Here is a hint. What is the return value for the procedure call **f(-2)**?

```
> f(-2);
> 
```

Exercise: Consider the following two functions.

```
> f := proc(x) if x>=0 then x^2 else 0 fi end;
> g := proc(x) if x>=0 then x^2 fi end;
```

Their graphs look similar.

```
> plot(f, -10..10);
> plot(g, -10..10);
```

But they are not the same function. Explain how they differ.

```
> 
```

```
> 
```

14.10. Online help for control statements

Here is the main help page about repetition statements.

```
> ?for
```

Here is the main help page about conditional statements (and the conditional operator).

```
> ?if
```

Here is further information about boolean expressions.

```
> ?boolean
```

```
> ?evalb
> ?type,logical
```
One way to discover some of Maple's boolean functions is to use the "Topic Search" menu item in the Maple Help menu and enter into the "Topic:" field just the two letters "is". This will bring up a list of several Maple functions that begin with "is" and which are boolean functions.

In the section on boolean expressions we mentioned that equations (and inequalities) are considered as both boolean expressions and as algebraic equations, and as boolean expressions they are evaluated differently than other kinds of boolean expressions. These facts are mentioned briefly in the first two bullet items in the first of the following three help pages, and in the third bullet item in each of the other two help pages.
```
> ?evalb
> ?boolean
> ?equation
```

The next two commands bring up the help pages for the **seq**, **add**, and **mul** commands. In these help pages are precise descriptions of how these commands are related to for-loops.
```
> ?seq
> ?add
```
The next two commands bring up the help pages of the **sum** and **product** commands. Neither of these help pages mentions for-loops, since these commands are much more sophisticated than just a for-loop
```
> ?sum
> ?product
```

For an explanation of print levels and the **printlevel** variable, read the following help page.
```
> ?printlevel
```
Here is the help page for the **print** command.
```
> ?print
```
And here is a page that mentions what the name **NULL** represents.
```
> ?NULL
```

Variables like **printlevel**, **Digits**, and **%** play a special role in Maple. They are called **environment variables**. The following command brings up a general page about environment variables.
```
> ?environment
```
Another name for **%** is **ditto**; here is a page about it.
```
> ?ditto
```
Notice how in one help page **%** is referred to as an environment variable, but in the other help page **%** is referred to as the ditto operator. Variables and operators are *very* different things. This just goes to show you how inconsistent the documentation can be.

Maple has a built in way for a procedure to return unevaluated. This is described in each of the following two help pages. The documentation refers to this as "fail return".

```
> ?RETURN
> ?procname
```

Control statements are just one kind of Maple statement. The next command calls up an index to all of Maple's statements. Notice however that this page in kind of misleading. For example, **proc** and **function** are listed on this page as "statements", but the help pages for **proc** and **function** clearly state that these are expressions.

```
> ?statement
```

```
>
```