Maple for Math Majors

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

13. Procedures in Maple

13.1. Introduction

An idea that we keep returning to is that there are two ways to represent mathematical functions in Maple, as either Maple expression or as Maple functions. As we showed in the last worksheet, the underlying idea of Maple expressions is a data structure. In this worksheet we show that the underlying idea of Maple functions is a procedure. Recall that our initial definition of a data structure was that it is a collection of data. Our initial definition of a procedure is that it is a collection of instructions to Maple. But just as a data structure is more than just a collection of data, a procedure is more than just a collection of instructions. In this worksheet we see what we mean by a collection of instructions, how we turn a collection of instructions into a procedure, and why a procedure is more than just the collection of instructions. We show that Maple functions are a special case of procedures, we compare mathematical functions to Maple procedures to see what they have in common and how they differ, and we see how mathematical functions can be represented by Maple procedures. We look at several examples of procedures, some that represent mathematical functions and some that do not, and we see how procedures can manipulate data structures.

```
> 
```

13.2. From execution group to procedure

Earlier we said that a procedure is a collection of instructions. But this is an over simplification. For example, an execution group is a collection of instructions to Maple but, as we will see, an execution group is not a procedure. Let us look at a simple example.

Every positive integer can be decomposed into its prime factorization. Maple has a command to do this.

```
> ifactor( 550550 );
```

Let us find a sequence of Maple commands that will compute for us the largest prime in an integer's prime factorization. The last command produced a (nested) data structure as its output. The largest prime factor is contained in the last piece of that data structure. We can use the **op** command to get that last piece.

```
> op( nops(%), % );
```

And we can use another **op** command to get the largest prime from out of that piece.

```
> op( 1, % );
```

Now let us put these commands together in an execution group that will always compute the largest prime factor contained in a positive integer *n*. To understand how this execution group "works", compare the comment to the right of each command with the command's output line below the execution group.

```
> 550550;            # Choose a positive integer.
> ifactor( % );      # Factor it.
> op( nops(%), % ); # Pick off the largest (i.e., last) factor.
> op( 1, % );        # Remove the exponent (if there is one).
> op( 1, % );        # Remove the parentheses.
```

Suppose that we only want to see the final result; we can replace the semicolons with colons to suppress all the outputs except for the final one.

```
> 550550:
> ifactor( % ):
> op( nops(%), % ):
> op( 1, % ):
> op( 1, % );
```

If we go back and change the value of the positive integer, we can do the same calculation for a different number. (Try this execution group with several different positive integers.) So we can say that this execution group takes a positive integer *n* as an "input" and returns as an "output" the largest prime number in the prime factorization of *n*.

```
>
```

So now we have a collection of Maple commands that we have combined into an execution group and the execution group performs a useful calculation for us. Since this calculation might be useful to us later on, we would like to be able to reuse the execution group. But this execution group is not very convenient to reuse. To use it later on you either have to go back in the worksheet to where the execution group is and use it or you have to cut and paste the execution group to some other place in the worksheet and then use it. This problem of not being able to conveniently reuse a collection of instructions is one motivation for the idea of a **procedure**.

What would make the execution group easier to use? First, it would be nice if the group of commands that make up the execution group could be given a name so that we could refer to them from anywhere in our worksheet, much like we can give an expression or a function a name and refer to it from anywhere in a worksheet. Second, it would be nice if we could tell the execution group, at the same time that we refer to it by its new name, what integer it is we want it to work on. Suppose we gave the execution group the name `largest_prime`. It would be nice if we could tell the group of instructions to work on the integer 550550 by using something like `largest_prime(550550)` (which should look to you a lot like a function call).

```
>
```

Here is how we do exactly what we described in the last paragraph. We convert our execution group into a procedure using the following **procedure definition**.

```
> largest_prime := proc(n)
>    ifactor( n );     # Factor the input.
>    op( nops(%), % ); # Pick off the last factor.
>    op( 1, % );       # Remove any exponent.
>    op( 1, % );       # Remove the parentheses.
> end;
```

Notice several things about the procedure definition. The procedure definition itself begins with the word **proc** (which is an abbreviation of "**proc**edure") and ends with the word **end**. The assignment operator in the first line assigns the procedure a name. We gave our procedure the name **largest_prime**. The input has moved to a different place from where it was in the execution group. The input to the procedure (which we named **n**) appears inside the parentheses after the word **proc**. The procedure definition then has four Maple commands in it, almost, but not exactly, the same four commands that are in the execution group. (Find the one difference.)

```
>
```

Now here is one very big and very important difference between the procedure definition and the execution group. When you put the cursor inside the procedure definition and hit the Enter key, you do not execute the instructions in the procedure like you would execute the instructions in an execution group. Instead, you only inform Maple about the procedure and its name. The name we choose becomes an assigned variable and the value of this variable is the definition of the procedure. If you have not yet "executed" the above procedure definition, do so now. Then the following command will remind us of the definition of our procedure.

```
> eval( largest_prime );
```

So how do we execute the instructions in the procedure? We use a **procedure call**. For example, we can call our new procedure with the input **550550**.

```
> largest_prime( 550550 );
```

But now we can also call our procedure many times with many different inputs. So we see that our procedure is very convenient to use and reuse.

```
> seq( largest_prime(i), i=550550..550560);
```

We just used our procedure ten times! The fact that we have named the instructions in the procedure makes the procedure a lot more useful than the execution group. In fact, the procedure can now be used anywhere in our worksheet.

```
>
```

Exercise: In the execution group, we changed most the semi colons at the ends of the command lines to colons so that we would only see the final result of the execution group. Notice that the commands in the procedure body all have semi colons at the end of them and yet we do not see any of the intermediate results from the procedure calls, only the final result. Go back to the procedure definition and change *all* of the semicolons to colons and then re-execute the procedure definition. What affect does this have?

```
>
```

[>

13.3. Some definitions

Let us analyze our procedure `largest_prime` from the last section in more detail. The Maple command used to define a procedure is referred to as a **procedure definition**. A procedure definition begins with `proc` and ends with `end`. We can use the assignment operator give a name to the procedure being defined. In our case we named the procedure `largest_prime`. Any variable names contained inside the parentheses after the `proc` are called **formal parameters** (they are also called input parameters, or parameter variables). A procedure can have any number (including zero) of formal parameters. In our procedure `largest_prime`, `n` is the only formal parameter. The sequence of Maple commands that goes between the list of formal parameters and the `end` is call the **procedure body**. The procedure `largest_prime` has four commands in its procedure body.

[>

When we use the name of a procedure as a Maple command (as in `largest_prime(550550)`), we refer to this as **calling the procedure** (or, a procedure call). The values that the formal parameters receive from the procedure call are referred to as **actual parameters**. So `largest_prime(550550)` means that the formal parameter `n` in the body of the procedure definition gets the actual parameter `550550`. The process of substituting actual parameters for formal parameters in a procedure call is referred to as **parameter passing**.

[>

The commands in the procedure body are not executed by Maple until the procedure is called. When you are first typing a procedure definition into Maple, it does not execute any of the commands in the procedure body (like `ifactor`). The execution of the commands in the procedure body waits until the procedure is called, and then the commands are executed with the formal parameters taking on the values of the actual parameters. The **return value** of the procedure call is the result of the last command executed by Maple during the procedure call. In our example `largest_prime`, the last command executed is always the fourth command, and so the result of this command is the result that Maple "returns" for the procedure call. The return value can be thought of as the "output" of the procedure. Notice that all the commands in our procedure body for `largest_prime` end in a semicolon but Maple does not print out the results of any of the commands except the last command (and the result of the last command is the return value). This is different than in our original execution group, where we had to put a colon at the end of each command for which we did not want to see any output.

[>

Here is a simple example of a procedure definition. This procedure is named `plus`, it has two formal parameters, and it has three commands in its procedure body. Notice that the results of the first two commands are not used in any way. When this procedure is called, Maple will execute all

three commands, but the results of the first two commands are just "thrown away". (So this is really a pretty silly example.)

```
> plus := proc(x,y)
>         x - y;
>         x * y;
>         x + y;  # This is the return value.
> end;
```

Here is a call to **plus** with actual parameters **3** and **4**.

```
> plus( 3, 4 );
>


>
```

13.4. Parameter, local, and global variables

In a procedure there are three kinds of variables, **parameter variables**, **local variables**, and **global variables**. Consider the following example, a slight variation on the last example from the last section.

```
> plus := proc(x,y)
>      local a, b;
>      global c;
>      a := x - y;
>      b := x * y;
>      c := x ^ y;
>           x + y;  # the return value
> end;
```

This procedure has two parameter variables (**x** and **y**), two local variables (**a** and **b**), and one global variable (**c**). The parameter variables (i.e., formal parameters) are place holders for the actual inputs passed to the procedure during a procedure call. The local variables can be thought of as "scratch pad variables", or "temporary variables". Local variables are used to hold temporary results that might come up while we are trying to compute our main result in the procedure. As we will see, the local variables only "live" inside the procedure call. They do not exist outside the procedure, and they do not survive from one procedure call to the next (this is why they get their name "local"). The global variable is just that, it is global to the whole Maple worksheet. As we will see, global variables inside procedure bodies are really the same as the variables we use in commands at our worksheet prompts.

Let us see how all this works with the procedure **plus** (make sure that you have executed the definition of **plus**). Here is a procedure call for **plus**.

```
> plus( 2, 5 );
```

Now what about the variables **a**, **b**, and **c**?

```
> a; b; c;
```

Notice that **a** and **b** are still unassigned but **c** has the value $2^5 = 32$. The variables **a** and **b** in our

worksheet are unassigned because the variables **a** and **b** inside the procedure **plus** are local variables. The **a** and **b** inside **plus** have no affect on the "global" **a** and **b**, the ones in our worksheet. On the other hand, the "global" variable **c** in our worksheet is the same variable as the **c** inside **plus** since it is declared to be a global variable there. Notice that a global variable can be used as a sneaky way for a procedure to "output" another result besides its official return value. Here is another call to the procedure **plus**.

```
> plus( -2, 2 );
```

Now look at the value of **c**.

```
> c;
```

The value of **c** changes with each call to **plus**.

It is worth mentioning that parameter variables are a kind of local variable since they have no affect except during a procedure call. For example, notice that the variables **x** and **y** are still unassigned in the worksheet.

```
> x; y;
```

The parameter variables **x** and **y** that were given the values **-2** and **2** in the last procedure call were local to the procedure call, so they have no affect on the global **x** and **y** in our worksheet.

We will see several uses for local and global variables later in this worksheet and in the following worksheets.

```
>
```

```
>
```

13.5. Another example

Here is another example of converting an execution group into a procedure. Suppose we have a large integer and we want to know what one of its digits is. For example, suppose we want to know what is the 15th digit (from the right) of the integer 47!.

```
> 47!;
```

You could just start counting from the last digit on the right, but that is tedious, error prone, and will not work for finding the 257th digit in 200!.

```
> 200!;
```

The length command can be used to find out just how many digits there are in an integer. The next command shows that 200! has a lot more than 257 digits.

```
> length( % );
```

Let us find a sequence of Maple commands that will find the *i*'th digit (from the right) of an integer *n* . Take for example *n* to be 98765 and *i* to be 3 (so the answer is 7).

```
> n := 98765;
```

```
> i := 3;
```

The integer *n* has its decimal point to the right of its first digit. The following command moves the decimal point to the left of the i'th digit of *n*.

```
> evalf( 10^(-i)*n );
```

The next command returns the **frac**tional part of the last result (the part to the right of the decimal point), leaving the digit we want just on the right of the decimal point.

```
> frac( % );
```

The next command moves the decimal point over to the right one place, leaving the digit we want by itself just to the left of the decimal point.

```
> 10*%;
```

Finally, the next command **trunc**ates off the fractional part of the last result leaving us with just the digit we want.

```
> trunc( % );
```

Now let us put these commands together in an execution group. (Notice that we really do not need the **evalf** in the third command.)

```
> n := 98765:
> i := 3:
> 10^(-i)*n:
> frac( % ):
> 10*%:
> trunc( % );
```

This execution group performs a useful calculation, but it is not easy enough to use and reuse. So now let us put our commands into the body of a procedure. We will call the procedure **get_digit** and this procedure will take in two numbers as inputs.

```
> get_digit := proc(n,i)
>     10^(-i)*n;
>     frac( % );
>     10*%;
>     trunc( % );
> end;
```

Let us try it out with a procedure call.

```
> get_digit( 98765, 3 );
```

Now we can easily find the 257'th digit of 200!.

```
> get_digit( 200!, 257 );
```

Of course, there is no easy way to verify this result. The best that you can do is to test this procedure on a lot of verifiable inputs until you have convinced yourself that the procedure is always correct.

```
>
```

Exercise: Recall that in our original sequence of commands we had an **evalf** function. Supppose we put it back in the body of our procedure.

```
> get_digit2 := proc(n,i)
>     evalf( 10^(-i)*n );
>     frac( % );
>     10*%;
>     trunc( % );
> end;
```

Show that the procedure is now incorrect. Explain what can go wrong.

```
> 
```

Here is an application of our new procedure that takes a positive integer **n** and returns an expression sequence made up of the integer's digits.

```
> n := 30!;
> seq( get_digit(n, length(n)-i), i=0..length(n)-1 );
```

The following command will tell you which of the digits 0 to 9 are used in the integer **n**. (Why does this work?)

```
> { % };
```

So the digit 7 did not appear in **n**.

The next two commands add up the digits in the number **n**.

```
> L := seq( get_digit(n, length(n)-i), i=0..length(n)-1 );
> add( i, i = L );
```

(Note: The expression **i=L** in the **add** command means that **i** successively takes on each of the values in the expression sequence **L**.)

The command that returned an expression sequence of the digits from an integer **n** is itself a useful command, but it is tedious to keep typing it in. So let us turn this command into another procedure that we will call **digits**.

```
> digits := proc(n)
>   local i;
>   seq( get_digit(n, length(n)-i), i=0..length(n)-1 );
> end;
```

Notice that what we have here is a procedure that we defined that uses another procedure that we defined. This is typical of Maple programming. We build up interesting new procedures out of simpler procedures that we previously defined.

Now for some fun. Here are a few surprising facts about numbers that we can verify with our new procedure. Each of the numbers below is an example of what some people call Narcissistic Numbers because each number is somehow related to its own digits in some unusual way.

Here is an interesting fact about the integer 2^{70}.

```
> n := 70;
> L := digits( 2^n );
> add( i, i = L ); # Add up the digits of 2^n.
```

So the digits of 2^{70} add up to 70. Try this for 2^n for some other n.

The number 40,585 has the following amazing property.

```
> n := 40585;
> L := digits( n );
```

```
> add( i!, i = L );  # Add up the factorials of the digits of n.
```
So the number 40,585 is the sum of the factorials of its digits (again, try this for some other integers). Such numbers have been called factorians. See the book *Keys to Infinity*, p.169-171, by Clifford Pickover.

The number 3435 is the sum of each of its digit raised to its own power.
```
> n := 3435;
> L := digits( n );
> add( i^i, i = L );  # Add up the digits raised to their own
  power.
>
```

Exercise: The number 438579088 has the same property as 3435 if you let 0^0 be 0 (which Maple does not). Can you think of a way to verify this property of 438579088 using the last two commands? (Hint: Do not change the last two commands. Make a small change in 438579088.)
```
>
```

The following number is called a **digital invariant**. It is the sum of each of its digits raised to the number-of-digits power.
```
> n := 82693916578;
> L := digits( n );
> add( i^nops([L]), i = L );  # The power is the number of
  digits.
```
Here is a huge digital invariant.
```
> n := 115132219018763992565095597973971522401;
> L := digits( n );   # There are 39 digits.
> add( i^nops([L]), i = L );
>
```

Consider the following example and try to figure out what narcissistic property this number has.
```
> n := 2646798;
> L := digits( n );
> add( L[i]^i, i = 1..length(n) );
>
```

All of the examples of Narcissistic Numbers used here came from the Narcissistic Numbers web page, http://www.geocities.com/~harveyh/Narciss.htm.
```
>
```

```
>
```

13.6. Maple functions are procedures

A function in Maple is really like a procedure. Below we define a procedure, a Maple function, and an expression, each of them equivalent to the mathematical function $f(x) = x^2 + 2x - 1$.

```
> f1 := x^2+2*x-1;    # The mathematical function as an
  expression,
> f2 := x->x^2+2*x-1; # as a Maple function,
> f3 := proc(x)       # as a procedure.
>    x^2+2*x-1
> end;
```

Let us see how Maple remembers the definitions of **f1**, **f2**, and **f3**.

```
> eval( f1 );     # f1 was defined as an expression.
> eval( f2 );     # f2 was defined as a function.
> eval( f3 );     # f3 was defined as a procedure.
```

Now we will check the data types of **f1**, **f2**, and **f3**.

```
> whattype( eval(f1) );   # f1 was defined as an expression.
> whattype( eval(f2) );   # f2 was defined as a function.
> whattype( eval(f3) );   # f3 was defined as a procedure.
```

Notice that Maple considers both **f2** and **f3** to be of type **procedure**, so Maple treats functions as procedures.

```
>
```

```
>
```

13.7. How a mathematical function is like a procedure

Here is a simple procedure that implements the mathematical function $f(x, y) = x + y$. The procedure has two formal parameters and only one command in the body of the procedure. Whatever that command computes is the return value of the procedure when the procedure is called.

```
> plus := proc(x,y)
>    x+y;
> end;
```

Here are a few calls to this procedure.

```
> plus( 3, 4 );
> plus( -3, 4 );
```

We can "compose" procedure calls just as we can compose functions. The following composition will compute the sum of three numbers.

```
> plus( 5, plus(3,4) );  # Add three numbers.
```

The actual parameters in a procedure call may be unassigned variables.

```
> plus( variable1, variable2 );
```

In the next example, how are the variables **x** and **y** in the procedure call related to the formal parameters **x** and **y** in the definition of the procedure?

```
> plus( x+y, x-y );
```

The procedure call **plus(x+y, x-y)** can be a bit confusing. Here, the formal parameter **x** (in the procedure definition) gets the actual parameter **x+y** (from the procedure call), and the formal

parameter **y** gets the actual parameter **x-y**. You should think of the formal parameters as "place holders" for the actual parameters. And remember, the variables **x** and **y** in the procedure call are global variables while the formal parameter **x** and **y** in the body of the procedure are local variables. So the variables **x** and **y** in the procedure call are not the same **x** and **y** that are in the procedure body.

```
>
```

This is just like in algebra and calculus. If $f(x) = x^2 + 2x - 1$, then what is $f(x+h)$? It is $(x+h)^2 + 2(x+h) - 1$. The equation $f(x) = x^2 + 2x - 1$ is a "procedure definition", f is the name of the "procedure", and the x in this equation is a "formal parameter". Then $f(x+h)$ is a "procedure call", and $x+h$ is the "actual parameter" for the "formal parameter" x.

Notice that mathematically, $f(x) = x^2 + 2x - 1$, $f(y) = y^2 + 2y - 1$, and $f(z) = z^2 + 2z - 1$ are all definitions of the same function f. Each one just uses a different formal parameter in the definition of the function. Similarly, if we rename the formal parameters in the definition of **plus**, that does *not* change the procedure. It will still be the same procedure, because it will do exactly the same thing. In other words, the following definition of **plus** is not really any different from the first definition.

```
> plus := proc(u,v)
>   u + v;
> end;
```

Here is a way to use Maple to confirm this claim.

```
> evalb( proc(u,v) u+v end = proc(x,y) x+y end );
```

```
>
```

Exercise: Make a small change in the last command so that the two procedures are no longer the same.

```
>
```

One last note. To Maple, the three expressions **x^2+2x-1**, **y^2+2y-1**, and **z^2+2z-1** are not the same expression, since they contain different variable names (that is, the data structures contain different data). But remember, Maple treats expressions and functions differently.

```
>
```

```
>
```

13.8. Anonymous procedures

When we define a procedure we almost always give it a name. But we do not have to. We can have unnamed procedures just as we can have unnamed functions or expressions. Unnamed procedures are called **anonymous procedures**. Here is an anonymous procedure that takes a string and a positive integer as input and returns the string truncated to the integer number of letters.

```
> proc(x::string, n::posint)
>   local i;
>   seq( x[i], i=1..n );
>   cat( % );
> end;
```

Right now the procedure is anonymous. But we can still call this procedure.

```
> %( "this is a long name", 11 );
```

We can give the procedure a name, so that it is no longer anonymous.

```
> shorten := %%;
```

Now we can call the procedure by its new name.

```
> shorten( "Aren't we having fun?", 15 );
```

Here is an anonymous procedure that reverses the letters in a string. The following command defines the anonymous procedure and then calls it, all in a single command.

```
> x := (proc(x) local i; cat(seq(x[-i], i=1..length(x))) end)(
  "try it out" );
```

Notice that the assignment operator was acting on the result of the call to the anonymous procedure.

```
> x;
```

```
>
```

The last few commands were meant to emphasize that defining and naming a procedure are two distinct steps and procedures do not have to have names to be used. Anonymous procedures can be used anywhere anonymous functions can be used. But anonymous functions are more common than anonymous procedures.

```
>
```

Exercise: Pick apart the two anonymous procedures from this section and make sure that you understand how they work. Try converting them into execution groups, so that you can see the results of each step in the procedure body.

```
>
```

```
>
```

13.9. Procedures and data structures

The next few examples are meant to show how knowledge of data structures can be useful for writing procedures.

Here are a few simple procedure definitions. These three procedures compute the average of two, three, and four input numbers respectively.

```
> avg2 := proc(x,y)
>     (x+y)/2;    # Compute the average of two numbers.
> end;
```

```
> avg3 := proc(x,y,z)
>     (x+y+z)/3;  # Compute the average of three numbers.
> end;

> avg4 := proc(a,b,c,d)
>    (a+b+c+d)/4;  # Compute the average of four numbers.
> end;
```

Here are a few calls to our procedures.

```
> avg2( 10, 3 );
> evalf( % );
> avg3( -1, 34, 12 );
> avg4( 50, 32, 100, 1 );
> evalf( % );
> avg4( 3, 3, 3, 3 );
>
```

What if we want to compute the average of 27 numbers? Do we have to write a procedure that has 27 formal parameters? The next procedure solves the problem of how we can average any number of numbers without having to write an infinite number of procedures. This procedure has only one input parameter, but that input is a data structure (in this case a list) that can hold any number of numbers. This is an example of using a data structure to solve a programming problem.

```
> avg := proc(L)
>    local i, N, S;            # Local variables.
>    N := nops( L );           # How many numbers we are
  averaging.
>    S := add( L[i], i=1..N ); # Add up the numbers in the list.
>    S/N;                      # This is the return value.
> end;
```

The procedure **avg** determines how many numbers are in the list **L** and puts the result in **N** (a local variable). It uses the **add** command to add up the numbers in the list and stores the sum in the local variable **S**. Then it divides **S** by **N** to get the average. (Notice the use of comments inside the procedure to help explain what is being done.) This example uses three local variables. Notice how the local variables are used to hold temporary results that come up while we are trying to compute our main result. Remember that the local variables only "live" inside a procedure call. They do not exist after the procedure has returned and they do not survive from one procedure call to the next.

```
>
```

Here are some calls to **avg**.

```
> avg( [2,3,4,5,6,7] );
> evalf( % );
> L := [12, 32.3, Pi, 67, 100, exp(2), 5.5, 44, 66, 100] ;
```

```
> avg( L );
> evalf( % );
```
Remember, the global variable **L** in this last procedure call is not the same variable as the parameter variable **L** in the definition of **avg**.
```
>
```

What do you think **avg** will do with the following input, a list of lists?
```
> L := [ [1,2], [2,3], [3,4] ];
> avg( L );
```
Now try **avg** on a list of polynomials.
```
> L := [ 1+x^2, 3+2*x^2+5*x^3, 4+2*x^2 ];
> avg ( L );
```
The last two commands worked but the next one does not. The next input causes trouble since the elements of the list are sets.
```
> L := [ {1,2}, {2,3}, {3,4} ];
> avg( L );
>
```

Exercise: Why could our procedure average the numbers from a list of lists and return a list of averages, but it could not average a list of sets to produce a set of averages? Hint: The key idea has to so with addition. Does it make sense to add two lists of numbers together, as in **[1,2,3]+[4,5,6]**? Does it make sense to add together two sets of numbers, as in **{1,2,3}+{4,5,6}**?
```
>
```

When we wrote the procedure **avg**, we were thinking that the list **L** represented a list of numbers. It turns out that the procedure works on lists of some other data types but it also does not work with lists of certain data types. There is a sophisticated way for us to force our procedure to work only in the way that we originally thought of it, as an average of a list of numbers. We will use what is called a **type declaration** in the list of formal parameters for **avg** to tell Maple exactly what kind of data types the actual parameters are supposed to be. When the procedure is called, Maple will check the data type of the actual parameters and see if they are of the correct type (this is referred to as **type checking**). If the actual parameters are not of the data type declared for the formal parameters, then Maple will return an appropriate error message.

Here we tell Maple that the formal parameter **L** represents a "list of numbers".
```
> avg := proc(L::list(numeric)) # Tell Maple exactly what L
  represents.
>    local i, N, S;
>    N := nops( L );
>    S := add( L[i], i=1..N );
>    S/N;
```

```
> end;
```
Now let us try this new definition of **avg** on the bad input.
```
> L := [ {1,2}, {2,3}, {3,4} ];
> avg( L );
```
The error message we got here is a bit more informative than it was before. It tells us what kind of input **avg** was expecting as opposed to the kind of input we passed to it.
```
>
```

Now consider this next example.
```
> avg(2,3,4,5,6,7);  # What's wrong with this input?
```
Notice that only the number **2** was passed to the procedure **avg** (read the error message carefully). All the other elements of the expression sequence in the function call were ignored since **avg** was defined to take only *one* input. And since **2** is not a list, we got the error message. This last command demonstrates two things. First of all, Maple has the unusual property of allowing procedure calls with more parameters than the procedure was written to accept. We will see in the next section why Maple chooses to behave this way. Compare this with mathematical functions; if the function f is defined by $f(x) = 2x - 1$, then the "function call" $f(\pi, 2)$ is considered an error. But in Maple, the 2 would be ignored and only the π would be passed to the function (i.e., procedure). Maple will complain however if a procedure is passed too few parameters as the following procedure call demonstrates.
```
> avg3(0,1);   # avg3 expects 3 inputs
```
The other thing that we learn from the command
```
> avg(2,3,4,5,6,7);
```
is that our solution to the problem of averaging an arbitrary number of numbers is not really optimal. The last command should have been written like this:
```
> avg( [2,3,4,5,6,7] );
```
But those brackets in the function call are awkward. Compare the following sequence of commands.
```
> avg2(2,3);
> avg3(2,3,4);
> avg4(2,3,4,5);
> avg( [2,3,4,5,6] );
```
Why should the last command have to be typed in differently from the previous three? Looking at it from this point of view, the brackets solved one problem but they created another problem. How are users of our procedure supposed to remember that the correct usage of **avg** is the non intuitive
```
> avg( [2,3,4,5,6] );
```
instead of the more obvious, but incorrect,
```
> avg(2,3,4,5,6);
```
In the next section we will see how Maple provides a way to solve this dilemma caused by the way we wrote **avg**.
```
>
```

Exercise: Try to determine, before executing them, how Maple will interpret each of the following

procedure calls. Do this exercise with both definitions of **avg**, the first definition without type checking and the second definition with type checking. (You will need to change all of the colons to semi colons if you want to see the results.)

```
> avg([2],3,4,5,6,7):
> avg([2,3,4],5,6,7):
> avg([2,3,4],[5,6,7]):
> avg([[2,3,4],[5,6,7]]):
> avg(2,[3,4,5,6,7]):
>

>
```

13.10. Procedure data structure

We have said that almost everything in Maple is a data structure. So it should not be a surprise that procedures are themselves another kind of data structure and that they have the data type **procedure**. Here is a simple (and somewhat silly) example.

```
> f := proc(x::numeric,y)
>   local u, v;
>   global w;
>   option trace;
>   description "a silly example";
>   u:=x; v:=y; w:=u+v+5;
> end;
```

Let us check the data type of **f**.

```
> whattype( f );
```

Oops, forgot about last name evaluation.

```
> whattype( eval(f) );
```

Since our procedure is a data structure, we should be able to use the **op** command to examine the operands of this data structure, that is, use **op** to see what kind of data is stored in a **procedure** data structure.

The **op** command, however, works very strangely with **procedure** data structures, so looking at the data in a procedure is a bit confusing. First, since Maple uses last name evaluation for procedures, the following command gives us the data type of the name **f**, not the data type of **f**'s value, the procedure.

```
> op( 0, f );
```

The next command gives us the data type of the procedure which is the value of **f**.

```
> op( 0, eval(f) );
```

But in the next two commands, **op** switches to full evaluation and shows us the definition of the procedure named by **f**.

```
> op( f );
> op( 1, f );
```

But the definition of the procedure is not the data stored in the **procedure** data structure. To get at the data stored in the **procedure** data structure, we use the following command.

```
> op( eval(f) );
```

There are actually seven operands in a **procedure** data structure, as the next command shows.

```
> nops( eval(f) );
```

The next sequence of commands shows what is in each of the seven operands. A couple of these data items are currently empty in the data structure for **f**.

```
> op( 1, eval(f) );
> op( 2, eval(f) );
> op( 3, eval(f) );
> op( 4, eval(f) );
> op( 5, eval(f) );
> op( 6, eval(f) );
> op( 7, eval(f) );
```

Notice that the first operand of a **procedure** data structure is an expression sequence of formal parameters. If a formal parameter has a type declaration, then the type declaration is held in a **`::`** data structure.

```
> op( 1, [op( 1, eval(f) )] );
> whattype( op( 1, [op(1, eval(f))] ) );
```

Notice also that the definition of the procedure is nowhere in the **procedure** data structure.

```
>
```

To summarize, **op(f)** returns the definition of the procedure and **op(eval(f))** returns the contents of the **procedure** data structure.

```
> op( f );
> op( eval(f) );
```

There are seven operands in a **procedure** data structure and we access them using **op(i, eval(f))** with **i** from 1 to 7. The first operand is the list of formal parameter names. The second operand is the list of local variable names. The third operand is a list of options. The fourth operand is called the remember table (we will discuss this operand in the next section). The fifth operand is a descriptive string. The sixth operand is the list of global variable names. And the seventh operand is called the lexical table.

Exercise: Explain what the following procedure does and how it does it. In particular, explain the reasons for the right-quotes.

```
> pds := proc(p::procedure)
>   local i;
>   for i from 1 to 7 do print('op'(i, 'eval'(p))=op(i, eval(p)))
  od;
> end;
>
```

Modify the procedure **pds** so that it displays a more descriptive message on the left hand side of

each equal sign.
```
> 
```

Of the seven operands in a procedure data structure, we already know what three of them represent. They are the formal parameters, the local variables, and the global variables. Of the remaining four, the remember table is the most important and we will discuss it in the next section. The descriptive string seems pretty self explanatory. The lexical table lists what are called "lexically scoped variables". These are variables that can only occur when one procedure is defined inside the body of another procedure. We will say more about this in a later (optional) section. The options operand we discuss next.

A procedure definition can contain an optional **option** section. There are eight different properties that can be declared in the **option** section, **remember**, **builtin**, **system**, **operator**, **arrow**, **trace**, **package** and **Copyright**. Right now we want to describe just three of these, the **trace**, **operator** and **arrow** options. The **remember** option will be described in the next section. The **builtin** and **Copyright** options will be discussed in the worksheet on Maple programming. The other options we will not mention any further.

If a procedure is defined with both the **operator** and **arrow** options, then the procedure acts exactly as if it were defined using the arrow notation.
```
> g := proc(x) option operator, arrow; x^2+2*x-1 end;
> eval( g );
```
Notice that if we define a function using the arrow notation
```
> h := x -> x^2-2*x+1;
```
and then examine the option operand of the function's **procedure** data structure
```
> op( 3, eval(h) );
```
it will have both the **operator** and **arrow** options. A procedure can have one of the **operator** or **arrow** options without the other, but we will not go into what that means.

Exercise: Try defining a procedure with just the **operator** option or just the **arrow** option. Do they act like functions defined using the arrow notation?
```
> 
```

Now we shall mention the **trace** option. This is an option that is used for debugging a procedure. When we write procedures that are a bit complicated, it is very possible that we can make a mistake in the definition of the procedure and the procedure does not do what we expect it to do. Finding mistakes can be difficult, and the **trace** option is meant to help.

When a procedure has the **trace** option, Maple will print out quite a bit of information about the procedure whenever it is called. Here is an example with our silly function **f**, which has the **trace** option.
```
> f( 1, 2 );
```

Let us define another procedure with the `trace` option that calls the procedure `f`.

```
> g := proc(x,y) local u,v; option trace; u:=f(x,x); v:=f(y,y);
  u+v; end;
```

Now let us call `g` to see what Maple produces for us.

```
> g( 2, 3 );
```

What Maple is doing is giving us a "trace", or a history, of everything that goes on inside of the procedures `g` and `f`. Hopefully this information will be of use when we are trying to find out what we did wrong in the definition of a procedure. Here is another example. The following procedure takes as input a name and a positive integer and it truncates the name to the integer number of letters.

```
> shorten := proc(x::name, n::posint)
>   local i, y;
>   option trace;
>   y := convert(x, string );
>   y := seq( y[i], i=1..n );
>   y := cat( y );
>   convert( y, name );
> end;
```

Let us call this procedure.

```
> shorten( `a name that is too long`, 6 );
```

Notice how we can see all of the steps that occur inside the procedure as they are executed.

```
>
```

Exercise: The following procedure was meant to reverse the letters in a name, so `reverse_name(hello)` was supposed to output `olleh`. But there is a bug in the procedure. Give the procedure the trace `option` and find the bug.

```
> reverse_name := proc(x::name)
>   local i, y;
>   y := convert( x, string );
>   y := seq( x[-i], i=1..length(y) );
>   y := cat( y );
>   convert( y, name );
> end;
> reverse_name(hello);
>
```

Exercise: Explain the relationship between a `procedure` data structure and `function` data structure. The following two help pages might help a bit.

```
> ?type,procedure
> ?type,function
>
```

```
>
```

13.11. Remember tables

In this section we look at an ingenious feature of Maple's design that has two important, and seemingly unrelated, consequences for Maple. This feature, remember tables, improves the computational efficiency of Maple and it also gives Maple some of its symbolic abilities.

A **remember table** is part of a `procedure` data structure. It is the fourth operand (out of seven) in the data structure. But a procedure will only have a remember table if the procedure is defined with the `remember` option. Here is an example. First define a simple procedure without the `remember` option.

```
> f := proc(x)
>   x^2-2*x+1
> end;
```

Now look at its procedure data structure.

```
> op( eval(f) );
```

There are no options and no remember table.

```
> op( 3, eval(f) );
> op( 4, eval(f) );
```

Now let us redefine `f` with the `remember` option.

```
> f := proc(x)
>   option remember;
>   x^2-2*x+1;
> end;
```

Now look at its procedure data structure.

```
> op( eval(f) );
```

Now there is one option but it seems that there is still no remember table.

```
> op( 3, eval(f) );
> op( 4, eval(f) );
```

So the `remember` option is in the data structure but there is no remember table yet. Let us evaluate `f` at some input.

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

Now look at the procedure data structure.

```
> op( eval(f) );
```

There is the remember table.

```
> op( 4, eval(f) );
```

When a procedure has the `remember` option, every time the procedure is called, Maple records the input of the procedure call and the resulting output as an ordered pair in an equation data structure in the remember table. So the procedure call `f(2)` with the result of `1` is recorded in `f`'s remember table as the equation `2=1`. From now on, anytime we make the procedure call `f(2)`, Maple will get the result for the procedure call from the remember table, instead of re-evaluating the procedure.

Let us put some more values in **f**'s remember table.

```
> f(3), f(1000), f(Pi), f(sin(Pi/4)), f(sin(Pi/7)), f(z);
```

Let us look in the remember table.

```
> op( 4, eval(f) );
```

As we use the function, the remember table keeps growing. If we redefine the function, the remember table is wiped clean.

```
> f := proc(x)
>   option remember;
>   x^5 + 100;
> end;
```

Check the remember table again.

```
> op( 4, eval(f) );
>
```

Now we look at what makes remember tables so great, but we will also look at a problem that they can cause.

First of all, remember tables help Maple work more efficiently. That is, they speed up calculations. Here is a simple example. Let us ask Maple to compute the 45,000th prime number.

```
> ithprime( 45000 );
```

That calculation should have taken a few seconds. Now ask Maple to do it again.

```
> ithprime( 45000 );
```

The second time, the result is instantaneous because Maple pulled it out of the remember table for the **ithprime** procedure. If you wish you can look at the remember table for **ithprime** and find **45000=545747** in it, but the remember table for **ithprime** is, by default, very large so you may not want to bother displaying it.

```
> op( 4, eval(ithprime) ):
```

If we **restart** Maple, then all of the remember tables are restored to their original values. So after a **restart**, calculating **ithprime(45000)** will again take several seconds.

```
> restart;
> ithprime( 45000 );
>
```

Other examples of remember tables used to speed up calculations are the remember tables for **simplify** and **expand**. By default, these commands have empty remember tables.

```
> op( 4, eval(simplify) );
> op( 4, eval(expand) );
```

Let us do some algebra.

```
> seq( factor(1-x^i), i=1..20 );
```

Check the remember tables for **simplify** and **expand** again.

```
> op( 4, eval(simplify) );
> op( 4, eval(expand) );
```

Notice how **expand** picked up a lot of values. Reset the remember tables.

```
> restart;
```

Now do some more algebra.

```
> solve( 1+x+x^2+x^3+x^4+x^5=0, x );
```

And check the remember tables for **simplify** and **expand**.

```
> op( 4, eval(simplify) );
> op( 4, eval(expand) );
```

Notice how much stuff accumulated in the remember table for **expand** from just that one calculation.

```
>
```

If you are doing massive amounts of calculations with Maple, large numbers of intermediate results, which will quite often be used again and again, build up in lots of remember tables. This can save Maple from having to recalculate some intermediate result many times and can lead to a lot of time saving. But the price for this speed efficiency can be memory inefficiency. The remember tables can start to occupy a lot of the computer's memory. This can explain why some Maple calculations can need hundreds of megabytes of computer memory to run well. The trade off between time and memory efficiency is one of the most common themes running through computer science and computational mathematics. More often than not, the way to speed up a calculation is to use more memory. Maple's remember tables are actually nothing more than what computer scientists call a **cache** and the idea of a cache is ubiquitous in the world of computers. The CPU in your computer has a cache, the hard drive has a cache, your web browser has a cache, the servers your computer is connected to have caches. A cache is a place for a computer to store recently accessed data where it is cheaper (i.e. faster) to get it from than where the data originally was stored. Data is cached because of the belief that if you accessed the data recently, then you will probably access it again soon. In the case of Maple, the "original source" of the data is computing it, and then it is stored "locally" in the remember table. If you take the example of your web browser, the original source of the data is a web page on the Internet. Once you access a web page, your browser stores a copy of it on the computer's hard drive. If you go back to the web page, which is very likely, the browser will retrieve the local copy cached on the (very fast) hard drive rather than retrieving the original over the (much slower) Internet again. So your browser uses hard drive space to save you time. (But when people have very full disk drives or small disk drives, the size of the web browser's cache can become a problem).

```
>
```

Now here is the problem with remember tables, and this is a problem with all caches. Stale data. Sometimes the values in the remember tables (or in a cache) no longer represent the correct results (or the original information). Here is an example. We will define two functions, one of which calls the other, and we will give the calling function a remember table.

```
> h := proc(x)
>   x+2
> end;
> g := proc(x)       # g calls h
```

```
>   option remember; # and g remembers
>   h(x)+1
> end;
```
Now evaluate **g** at some point.
```
> g( 2 );
```
Now change the definition of **h**.
```
> h := proc(x)
>   x+100
> end;
```
Call **g** again with the same input.
```
> g( 2 );
```
That is not the correct answer. But it is the value that is cached in **g**'s remember table.
```
> op( 4, eval(g) );
```
So what do we do? We could **restart** Maple, but that may ruin a lot of other calculations that we are working on and be very inconvenient. There is a special Maple command called, not surprisingly, **forget** that clears a procedure's remember table. But this procedure is not part of any package and it is not loaded into Maple by default.
```
> forget( g );
```
In this case, returning unevaluated is the way we know that **forget** is not defined to Maple yet. So we need to load it before we can use it.
```
> readlib(forget);
> forget( g );
```
forget does not return a value, but it did clear **g**'s remember table.
```
> op( 4, eval(g) );
```
So now let us recalculate **g(2)**.
```
> g( 2 );
```
There are a lot of ways that old values in remember tables can cause Maple to return erroneous results. If you are getting strange results from Maple and are suspicious that Maple is getting bad results from remember tables, just use **restart** to clear all the remember tables. If you know exactly which procedure has the corrupted remember table, then you can use **forget**.
```
>
```

Exercise: Why did we need two functions for our example of a problem with a remember table? Why did we not just have one function **g**, evaluate it at a point, change **g**, and then evaluate it again at the same point?
```
>
```

Exercise: In the case of a web browser, what could cause a piece of data in the cache to become incorrect? An interesting feature of web browsers is that they have a way to automatically detect when cached information has gone stale. It would be nice if Maple also had a feature like that.
```
>
```

Now let us turn to the other great thing about remember tables. Remember tables provide Maple with some of its symbolic abilities. Here is an example.

```
> sin(Pi/5);
```

How did Maple know that? It pulled it out of the remember table for **sin**. Many Maple functions come with predefined remember tables. Let us look at **sin**'s.

```
> op( 4, eval(sin) );
```

Now notice something else. There is no entry in **sin**'s remember table for the input **3*Pi/5**. But Maple can still figure it out.

```
> sin(3*Pi/5);
```

There is a value for **sin(2*Pi/5)** in the remember table and the **sin** function knows how to make use of this value to compute **sin(3*Pi/2)**. We will see how the **sin** function does this in the worksheet on Maple programming.

Notice that Maple knows, from the remember table, what **sin(Pi/12)** is.

```
> sin(Pi/12);
```

This result can be simplified to $\frac{\sqrt{6}-\sqrt{2}}{4}$. I do not know why Maple stores this result in the form that it does.Let us set **sin(Pi/12)** to this simplified value.

```
> sin(Pi/12) := (sqrt(6)-sqrt(2))/4;
```

Check the remember table again.

```
> op( 4, eval(sin) );
```

The simplified value is now in the remember table. Now **restart** Maple.

```
> restart;
```

Check **sin**'s remember table again.

```
> op( 4, eval(sin) );
```

sin(Pi/12) is back to what it was. So **restart** does not just clear remember tables, it also restores default remember tables back to their default values.

Here are some other remember tables containing symbolic results. Look them over carefully to get a sense of the kind of symbolic results that Maple keeps track of. In particular, notice the number of results involving infinity and also complex numbers.

```
> op( 4, eval(ln) );
> op( 4, eval(exp) );
> op( 4, eval(arctan) );
>
```

Now let us teach Maple something that it does not know. If you look in a good trigonometry book, you will find the identity

$$\sin\left(\frac{\pi}{60}\right)=\frac{\sqrt{12-2\sqrt{15}-4\sqrt{5}+6\sqrt{3}}-\sqrt{20-2\sqrt{15}+4\sqrt{5}-10\sqrt{3}}}{8}$$

Maple does not know this.

```
> sin(Pi/60);
```

So let us teach Maple.

```
> sin(Pi/60) :=
>   ( sqrt(12-2*sqrt(15)-4*sqrt(5)+6*sqrt(3))
>   - sqrt(20-2*sqrt(15)+4*sqrt(5)-10*sqrt(3)))/8;
```

Now Maple knows the identity.

```
> sin(Pi/60);
```

And Maple is a fast learner. It now also knows the following. (We will see how later.)

```
> sin(59*Pi/60);
> sin(61*Pi/60);
>
```

Exercise: Maple automatically learned one other value when we taught it `sin(Pi/60)`. What was it?

```
>
```

Exercise: Teach Maple the symbolic values for `cos(Pi/60)` and `sin(Pi/30)`. Hint: Use a couple of trig identities.

```
>
```

Exercise: Look in a trigonometry book and find out how to derive the symbolic value for `sin(Pi/60)`. Hint: You can derive the value for `sin(Pi/60)` using values from `sin`'s default remember table.

```
>
```

Exercise: Look in a trigonometry book and find out how to derive the symbolic value for `sin(Pi/5)` that is in `sin`'s default remember table.

```
>
```

```
>
```

13.12. Return values and side effects

13.13. The args expression sequence (optional)

In a previous section we solved the problem of how to write a procedure that would compute the average of any number of input numbers. We solved the problem using a data structure (a list). But our solution was not all that appealing. Recall that our solution required that the procedure call have brackets in it so that the procedure would be passed only one argument, the list. The brackets made the procedure awkward to use. In this section we will see how to remove the need for the brackets.

Here is the procedure `avg` as we defined previously (without the type checking).

```
> avg := proc(L)
>   local i, N, S;
```

```
>   N := nops( L );
>   S := add( L[i], i=1..N );
>   S/N;
> end;
```

Here is an example of a call to this procedure.

```
> avg( [2,3,4,5,6,7,8] );
```

What we would like to be able to do is to call the **avg** procedure like this.

```
> avg(2,3,4,5,6,7,8);
```

Notice that in the proper procedure call the input is a list and in the improper procedure call the input is an expression sequence. Both lists and expression sequences are data structures capable of holding any number of numbers. Can we rewrite the definition of **avg** to have an expression sequence as the formal parameter instead of a list? The answer is no. We cannot have a formal parameter represent an expression sequence data structure like we had a formal parameter represent a list data structure. The reason has to do with the way Maple passes parameters. Every procedure call is of the form

procedure_name(*expression-sequence-of-actual-parameters* **)**

and the **proc** command that defines the procedure is of the form

procedure_name := proc(*expression-sequence-of-formal-parameters* **)** *Maple-statements* **end**.

Maple will take the elements of the *expression-sequence-of-actual-parameters* in the procedure call and match those elements up, one for one, with the elements of the *expression-sequence-of-formal-parameters* in the procedure definition. There is no way to have Maple take some of the elements from the expression sequence of actual parameters and pass them into a single formal parameter that represents an expression sequence.

```
>
```

However, writing procedures that take an arbitrary number of parameters is a common thing to do in Maple, so Maple provides a special solution to the problem we are up against. Maple provides every procedure with a special local variable called **args** that holds the entire *expression-sequence-of-actual-parameters* from a procedure call, and Maple also provides another special local variable **nargs** that holds the number of actual parameters in **args** (so **nargs** is the same as **nops([args])**).

Here is how we can use **args** and **nargs** to rewrite the procedure **avg**.

```
> avg := proc()
>   local i, S;
>   S := add( args[i], i=1..nargs );
>   S/nargs;
> end;
```

Notice how strange this seems at first. The procedure **avg** is now defined to take no parameters! But it does take parameters and it gets those parameters through the **args** expression sequence. (Notice how we are still using a data structure to solve the problem of how to pass an arbitrary number of

numbers to our procedure. Instead of using a list data structure, now we are using an expression sequence data structure. And instead of the data structure being help in a formal parameter, now it is held in a special local variable.)

Here are some procedure calls to our new version of **avg**.

```
> avg(2,3,4);
> avg(2,3,4,5,6,7);
>
```

Exercise: Notice what happens when we call **avg** with no parameters.

```
> avg();
```

Modify the definition of **avg** so that it returns a better error message when it is called with no arguments.

```
>
```

The procedures **avg2**, **avg3**, and **avg4** that we defined previously each represent a mathematical function, i.e., a real valued function of two, three, and four variables respectively (you can easily write a mathematical formula for each of these functions). The version of **avg** that we just defined does *not* represent a simple mathematical function. There is no easy way to define a domain for **avg** using common mathematical language and, similarly, there is no easy way to use mathematical notation to write a formula for **avg**. This example shows how the computer science notion of a procedure can extend the mathematical notion of a function.

Here are two more procedures that make simple use of the **args** and **nargs** variables. The first simply returns its actual parameter list and the second counts how many actual parameters it was called with.

```
> quote_them := proc()
>   args;
> end;
> count_them := proc()
>   nargs;
> end;
```

Here are some examples of their use.

```
> quote_them( a, b, c, d, e, f, g, h, i, j );
> count_them( a, b, c, d, e, f, g, h, i, j );
> quote_them( a, [b, c], {d, e}, f=g, h..i, j );
> count_them( a, [b, c], {d, e}, f=g, h..i, j );
```

Why was the last result 6?

```
>
```

Exercise: A lot of Maple commands use the **args** expression sequence so that the command can accept an arbitrary, or at least a variable number, of parameters. Take for example the **plot**

command. Convince yourself that various **plot** invocations can have a different number of parameters in their procedure calls. Look at the following examples. How many parameters are in each use of the **plot** command? (Hint: Count commas at the "top level" of the parameter sequence.)

```
> plot( x->x^2 );
> plot( {x->sin(x), x->cos(x)} );
> plot( x^2, x=-10..10 );
> plot( x->x^2, axes=none );
> plot( [sin(x), cos(x), x=-2*Pi..2*Pi], style=point );
> plot( [cos(x), sin(x)], x=-2*Pi..2*Pi, color=[blue, black] );
> plot( x->x^2, 0..5, 0..5, color=green );
> plot( [ln, x->x], 0..5, color=[blue,red],
>       style=point, adaptive=false, numpoints=10 );
```

Double check your answers by replacing each occurrence of the name **plot** with the name **count_them** and see what the return values are.

```
>
```

```
>
```

13.14. Recursive procedures (optional)

In this section we will look at an idea that is important to both mathematics and computer science, the idea of **recursion**. Recursion means defining something in terms of itself. We will see, for example, that many common mathematical ideas can be given recursive definitions. But our main interest will be in recursive procedures. A procedure is recursive if the procedure calls itself somewhere in its body, which would mean that the procedure is defined in terms of itself. Recursive procedures are important in computer programming because they often provide very compact and elegant solutions to a programming problem. But, unfortunately, the compactness and elegance of recursive procedures comes at a steep price. Recursive procedures tend to be slow and they tend to consume massive amounts of computer memory when they execute. We will see this with a couple of our examples.

```
>
```

Here is a famous example of recursion in mathematics. If n is a positive integer, the symbol $n!$ (read "n factorial") is defined to be the product of all the positive integers less than or equal to n. Using mathematical notation we would write

$$n! = n*(n-1)*(n-2)*...*3*2*1.$$

So for example 3! = 3*2*1 = 6, and 5! = 5*4*3*2*1 = 120. There is another way to define $n!$, a recursive way. The recursive definition is

$$n! = n\,(n-1)!.$$

This recursive definition defines factorials in terms of factorials. The factorial symbol appears on both side of the equal sign in the definition. Here is how we could try to compute 5! using this definition.

$$5! = 5*(4!) = 5*(4*(3!)) = 5*(4*(3*(2!))) = 5*(4*(3*(2*(1!)))) = 5*(4*(3*(2*(1*(0!)))))$$

But how did we know when to stop using the formula $n! = n(n-1)!$? We certainly do not want to write 5*(4*(3*(2*(1*(0*(-1!)))))) as the next step in the calculation. Besides implying that this calculation will go on for ever, it also seems to imply that the result is zero. We need something to bring the "recursion" to a halt. For the factorial function, that something is the rule that 0!=1. With that rule the calculation of 5! becomes

$$5! = 5*(4!) = 5*(4*(3!)) = 5*(4*(3*(2!))) = 5*(4*(3*(2*(1!)))) =$$
$$5*(4*(3*(2*(1*(0!)))))=5*4*3*2*1=120.$$

So the complete recursive definition of the factorial function has two parts

$$n! = n\,(n-1)! \quad \text{and} \quad 0! = 1.$$

Now we can take this recursive definition and implement it as a recursive procedure in Maple.

```
> recursive_factorial := proc(n)
>   if n=0 then
>     1
>   else
>     n*recursive_factorial(n-1)
>   fi;
> end;
```

Notice how both parts of the recursive definition appear in the recursive procedure. The procedure is recursive because in the 5'th line of the procedure it calls itself, and that is one part of the recursive definition of $n!$. The other part of the recursive definition is the conditional statement that checks if the input is zero (we will talk about the details of conditional statements in the next worksheet).

Let us compute some factorials using this recursive procedure.

```
> recursive_factorial(5);
> recursive_factorial(85);
```

There is a factorial function built into Maple so we can check our results.

```
> 85!;
>
```

Unfortunately, recursion is not always the best way to implement a procedure. For example, the next command does not work because recursion forces Maple to use so much of its computer memory that Maple runs out of space to store results.

```
> recursive_factorial(779);
```

But the built in Maple function for computing factorials has no trouble computing this result. It is not a recursive procedure.

```
> 779!;
> length( % ); # In case you are curious.
>
```

Here is a trick that will let helps us to visualize the workings of this recursive procedure. In the body of the procedure, use right-quotes to delay the evaluation of the recursive procedure call.

```
> recursive_factorial := proc(n)
>   if n=0 then
>     1
>   else
>     n*'recursive_factorial(n-1)'
>   fi;
> end;
```

Now call the procedure and then keep evaluating the output until it is fully evaluated.

```
> recursive_factorial(5);
> %;
> %;
> %;
> %;
> %;
>
```

Here is another way in which we can kind of watch the recursion take place We put in our procedure the line **option trace**. This causes Maple to "trace" all the steps that the procedure goes through as it calculates. (The **option trace** line should go in a procedure body after any local and global variable declarations.)

```
> recursive_factorial := proc(n)
>   option trace;
>   if n=0 then
>     1
>   else
>     n*recursive_factorial(n-1)
>   fi;
> end;
```

Now call the procedure and study the output.

```
> recursive_factorial(5);
>
```

Here is a non-mathematical, non-programming example of recursion, a recursive acronym. Let the acronym CALC stand for Can Anyone Like CALC. If we continue to expand the acronym, it will grow forever, since the acronym refers to itself. This infinite regress has to be avoided in mathematical or programming examples of recursion. In fact, we saw earlier an example in Maple of recursion with an infinite regress when we did the following:

```
> x := 'x';  # Unassign x.
> x := x+1;  # Give x a recursive (i.e., self-referential)
  definition.
> x;         # Maple gives up after a certain number of
  recursions.
> x := 'x';  # Get rid of the infinite recursion.
```

```
> 
```

When we write recursive procedures, we always have to do something to avoid infinite recursion. For example, if we take our mathematical definition of factorial $n! = n\,(n-1)!$ too literally, then we end up with the following procedure.

```
> recursive_factorial := proc(n)
>   n*recursive_factorial(n-1)
> end;
```

Let us try this version out.

```
> recursive_factorial(5);
```

It does not work. Why is the mathematical definition of factorial not infinitely recursive? Because we also have the rule that 0!=1, and that definition stops the recursion after a finite number of steps.

```
> 
```

Exercise: Lots of simple mathematical ideas can be given recursive definitions. For example, if a is a real number and n is a positive integer, then a^n and $n\,a$ can be given the following recursive definitions.

$$a^n = a\,a^{(n-1)}$$

and

$$n\,a = a + (n-1)\,a$$

What are the proper "terminating" conditions for these definitions?

```
> 
```

Here is another simple recursive procedure. This procedure adds up all the numbers in a list. Notice how we avoid the problem of infinite recursion by using an if-then-else-fi statement to check for a "terminating" condition.

```
> add_list := proc(x::list)
>   if nops(x)=0  # This check prevents infinite regress;
>   then          # if there is nothing in the list,
>     0           # then we return 0.
>   else
>     x[1] + add_list(x[2..-1])  # Here's the recursion.
>   fi;
> end;
```

Give it a try.

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

In the next worksheet we will see how to write a non recursive procedure for adding up a list of numbers.

```
> 
```

Here is `add_list` with delayed evaluation of the recursive procedure call.

```
> add_list := proc(x::list)
```

```
>   if nops(x)=0  # This check prevents infinite regress;
>   then          # if there is nothing in the list,
>     0           # then we return 0.
>   else
>     ''''''x[1]'''''' + 'add_list(x[2..-1])'  # Here's the
  recursion.
>   fi;
> end;
```

Notice that we also added a lot of delayed evaluation to the **x[1]** term in front of the recursive call. This is there to make the output from the following procedure call more illustrative of what is going on.

```
> add_list( [1,2,3,4,5] );
> %;
> %;
> %;
> %;
> %;
> %; %; %; %; %;
```

Here is **add_list** with **option trace**.

```
> add_list := proc(x::list)
>   option trace;
>   if nops(x)=0  # This check prevents infinite regress;
>   then          # if there is nothing in the list,
>     0           # then we return 0.
>   else
>     x[1] + add_list(x[2..-1])  # Here's the recursion.
>   fi;
> end;
```

Look carefully at the following output. Notice how it shows that the addition gets done from right to left in the list, not from left to right.

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

Exercise: The definition of **add_list** with delayed evaluation seems to imply that the addition is done from left to right in the list. The definition of **add_list** with **option trace** seems to imply that the addition is done from right to left in the list. Which is correct? Explain why one of these procedures gives an incorrect impression.

```
>
```

Exercise: Consider the following recursive version of **add_list**.

```
> add_list := proc(x::list)
```

```
>   if nops(x)=1
>   then
>     x[1]
>   else
>     x[1] + add_list(x[2..-1])  # Here's the recursion.
>   fi;
> end;
```
Explain how it is different from the previous version. Which is better and why?
```
>
```

Exercise: Write a recursive procedure call **prod_list** that computes the product of a list of numbers.
```
>
```

In the next worksheet, in the section on conditional statements, we will write a procedure **biggest** that finds the largest number in a list of numbers. Here we write a recursive version of this procedure. First, we need a procedure that finds the larger of just two numbers.
```
> bigger := proc(a,b)
>   if a >= b then a else b fi;
> end;
```
Now we can use **bigger** to write a version of **biggest** that is recursive. The procedure **biggest** will take in only one input, a list (which can be arbitrarily long). The recursive idea behind **biggest** is contained in the following sentence:

> The **biggest** number in the list [17, 3, 23, 45, 87, 67] can be defined as the **bigger** of 17 and the **biggest** number from the list [3, 23, 45, 87, 67].

This definition of **biggest** was self-referential since **biggest** appeared in the definition of **biggest**. But the first appearance of **biggest** was referring to a list with six numbers in it and the second appearance of **biggest** was referring to a list with five numbers in it (the first list with the first number deleted from it). The following procedure implements this recursive definition of **biggest**.
```
> biggest := proc(x::list)
>   if nops(x)=1 then   # This check prevents infinite recursion.
>     x[1]
>   else
>     bigger(x[1], biggest(x[2..-1]))
>   fi;
> end;
```
Let us test it on a short list of numbers.
```
> biggest( [17, 3, 23, 45, 87, 67] );
```

Let us test **biggest** on a list of randomly chosen integers. The following command creates the list (change the colon to a semicolon to see the list). If the list is much longer, then we get a "too many

levels of recursion" error from Maple when we call **biggest**. (At least I did on my computer. The maximum size of the list depends on how much memory your computer has.) This is because recursive procedures are very inefficient and they almost always run slower and use more memory than non-recursive procedures. You can check this by testing the non-recursive version of **biggest** from the next worksheet. Test that version of **biggest** with any size list of randomly chosen integers.

```
> random_list_of_integers := [ seq( rand(), i=1..520 ) ]:
> biggest( random_list_of_integers );
>
```

Exercise: Use long lists of random numbers to find out how long a list of numbers the recursive **add_list** can add. After you have seen how to add a list of numbers non recursively in the next worksheet, see how long a list of numbers the non recursive **add_list** can add.

```
>
```

Let us define **biggest** with delayed evaluation of the recursive procedure call.

```
> biggest := proc(x::list)
>   if nops(x)=1 then   # This check prevents infinite recursion.
>     x[1]
>   else
>     bigger(x[1], 'biggest'(x[2..-1]))
>   fi;
> end;
```

Let us try this version.

```
> biggest( [17, 3, 23, 45, 87, 67] );
```

We can fix this problem by modifying the definition of **bigger**. The following version of **bigger** will "return unevaluated" when it is called with non numeric operands. This technique is discussed in the next worksheet.

```
> bigger := proc(a,b)
>   if type( a, numeric ) and type( b, numeric ) then
>     if a >= b then a else b fi;
>   else
>     RETURN( 'procname(args)' )
>   fi;
> end;
```

Now we can call the version of **biggest** with delayed evaluation of the recursive procedure call.

```
> biggest( [17, 3, 23, 45, 87, 67] );
> %; %; %; %; %;
>
```

Now let use **option trace** to watch **biggest** and **bigger** do their work.

```
> bigger := proc(a,b)
```

```
>   option trace;
>   if a >= b then a else b fi;
> end;
> biggest := proc(x::list)
>   option trace;
>   if nops(x)=1 then
>     x[1]
>   else
>     bigger(x[1], biggest(x[2..-1]))
>   fi;
> end;
> biggest( [17, 3, 23, 45, 87, 67] );
>
```

Here is still another way to visualize what is going on as **biggest** executes recursively. We will keep **option trace** in **biggest** and remove it from **bigger**, and we will prevent the evaluation of **bigger** in the body of **biggest**.

```
> bigger := proc(a,b)
>   if a >= b then a else b fi;
> end;
> biggest := proc(x::list)
>   option trace;
>   if nops(x)=1 then
>     x[1]
>   else
>     'bigger'(x[1], biggest(x[2..-1]))
>   fi;
> end;
```

Now call this version of **biggest**. If you carefully study and think about all of this output, it should help you build up a sense of how recursive procedures really execute.

```
> biggest( [17, 3, 23, 45, 87, 67] );
```

Now we can execute all these calls to **bigger**.

```
> %;
>
```

Exercise: Write a self-contained version of **biggest** that does not need the procedure **bigger**.

```
>
```

Here is an interesting application of recursion to periodic functions. We say that a function f from the real line to the real line is periodic with period p if $0 < p$ and $f(x + p) = f(x)$ for all real numbers x. Notice that this is almost a recursive definition since it defines f in terms of itself. But this definition is not recursive because it does not contain any kind of a terminating condition. As it

stands, this definition defines the notion of being periodic for any function, but it does not define any specific periodic function. Here is how we adapt this definition to give a recursive definition of a specific periodic function. Let g be a function defined on the interval from a to b, where $a < b$ and $b - a = p$. Here is a recursive definition of a function f which is a periodic extension of g to the whole real line.

$$f(x) = \begin{cases} g(x) & a \le x \text{ and } x < b \\ f(x-p) & b \le x \\ f(x+p) & x < a \end{cases}$$

Notice that the definition of f is recursive because f is defined in terms of f, and g plays the role of the terminating condition in this recursive definition. Let us use a specific example to see how this definition works. Let $a = 0$, $b = 1$, and $p = 1$. Let g be defined by

$$g(x) = 4x^2 - 4x + 1 \quad \text{for } 0 \le x \text{ and } x < 1.$$

Define the periodic function f using the recursive definition above. Now consider how we evaluate $f(5.5)$. According to the definition of f we would do the following

$$f(5.5) = f(4.5) = f(3.5) = f(2.5) = f(1.5) = g(.5) = 0.$$

How would we evaluate f(-3.5)?

```
> 
```

Let us implement these definitions using Maple. First define **a**, **b**, **p** and **g**.

```
> a := 0; b := 1;
> p := 'b'-'a';
> g := x -> piecewise(a<=x and x<b, 4*x^2-4*x+1);
```

Here is what **g** looks like as an expression.

```
> g(x);
```

Now define **f**.

```
> f := proc(x)
>   if a<=x and x<b then
>     g(x)
>   elif b<=x then
>     f(x-p)
>   elif x<a then
>     f(x+p)
>   fi
> end;
```

We can evaluate **f**.

```
> f(5.5), f(-3.5), f(127.8);
```

But because **f** is defined recursively, there is a limit to how large of a value we can plug into it.

```
> f(1027.8);
```

Here are graphs of **g** and its periodic extension **f**.

```
> plot( g, -2..2, scaling=constrained, discont=true, color=red );
> plot( f, -2..2, scaling=constrained );
```

```
> 
```

Let us try using `f` with a couple of well known periodic functions.

```
> a := 0; b := 2*Pi;
> g := x -> piecewise( a<=x and x<b, 2*cos(x) );
> plot( g, -4*Pi..4*Pi, scaling=constrained, discont=true,
  color=red );
> plot( f, -4*Pi..4*Pi, scaling=constrained );
```

```
> a := 0; b := 2*Pi;
> g := x -> piecewise( a<=x and x<b, 2*sin(x) );
> plot( g, -4*Pi..4*Pi, scaling=constrained, discont=true,
  color=red );
> plot( f, -4*Pi..4*Pi, scaling=constrained );
```

Notice that we do not need to redefine `f`. We just need to redefine `a`, `b`, and `g`, and then `f` will automatically be the periodic extension of `g` with period `p=b-a`.

```
> 
```

Exercise: With `a`, `b`, and `g` defined as in the last example, try to figure out what causes the following error.

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

Hint: The following should evaluate to `true`.

```
> evalb( 2 < 4*Pi );
> 
```

Exercise: Suppose we make `a=0`, `b=p>0`, and we define `g` on the interval from `0` to `p`. What conditions on `g` would make `f` a continuous function? What conditions on `g` would make `f` an even function? What conditions on `g` would make `f` an odd function?

```
> 
```

Exercise: The following definition for the periodic extension of `g` would seem reasonable enough (everything that is inside the `piecewise` function is exactly like the conditional statement inside the body of `f`). But it does not work.

```
> f_wrong := x ->
>  piecewise( a<=x and x<b, g(x), b<=x, f_wrong(x-p), x<a,
  f_wrong(x+p) );
```

Let us try it. Define a function `g` to extend.

```
> a := 0; b := 1; p := 'b' - 'a';
> g := x -> piecewise( a<=x and x<b, 4*x^2-4*x+1 );
```

Now try to evaluate the extension.

```
> f_wrong(1/2);
```

The problem is that `piecewise` uses full evaluation so it always tries to evaluate every expression

inside its parentheses. Figure out why this causes the infinite recursion.

```
>
```

However, here is something very strange. The `plot` command can graph `f_wrong`, even using this incorrect definition! How does the `plot` command manage to evaluate `f_wrong` when Maple cannot evaluate `f_wrong` at the top level? I have no idea.

```
> plot( f_wrong, -2..2, scaling=constrained );
>
```

Exercise: Suppose we have `a=0` and `b=p>0` and we define `g` for `x` between `0` and `p`. Then the periodic extension `f` may be an even function or an odd function or it may be neither even nor odd. There is a way to define a `2*p` periodic extension of `g` that is always even, no matter what `g` is. Consider the following recursive definition of `f_even`.

```
> f_even := proc(x)
>   if 0<=x and x<p then
>     g(x)
>   elif -p<=x and x<0 then
>     g(-x)
>   elif p<=x then
>     f_even(x-2*p)
>   elif x<-p then
>     f_even(x+2*p)
>   fi
> end;
```

Let us try it with a particular `p` and `g`.

```
> p:=3*Pi/4;
> g := x-> piecewise( x>0 and x<p, 4*sin(x) );
```

Here are graphs of `g` and it `2*p` periodic even extension.

```
> plot( g, -4*Pi..4*Pi, scaling=constrained, discont=true,
  color=red );
> plot( f_even, -4*Pi..4*Pi, scaling=constrained );
```

Here is what the graph of `f`, the `p` periodic extension of `g`, would look like for the same `g`.

```
> a:=0: b:=p:
> plot( f, -4*Pi..4*Pi, scaling=constrained, discont=true,
  color=red );
```

The function `f_even` is called the "even periodic extension of `g` with period `2*p`". Explain how `f_even` creates the even, `2*p` periodic version of `g`.

```
>
```

Write a recursive procedure `f_odd` that defines a `2*p` periodic extension of `g` that is odd for any `g` defined on the interval from `0` to `p`.

```
>
```

```
>
```

13.15. Evaluation rules and procedures (optional)

13.16. Procedures that return procedures (optional)

13.17. Working with execution groups and procedures

In this section we go over some practical tips on how to work with execution groups and procedure definitions. In particular, we will go over how to create and edit an execution group and how to create and edit a procedure definition.

An execution group is a collection of consecutive Maple input lines that are grouped together so that all of the Maple commands on the input lines are executed together when the cursor is placed anywhere within the lines and the Enter key is hit. The easiest way to create an execution group is to first use the key combination Ctrl-j to create however many input lines that are needed. Then place the cursor on the first of the input lines and use the F4 key (or the "Edit -> Split or Join -> Join Execution Groups" menu item) to combine the input lines into an execution group. Each time you hit the F4 key, the prompt below the current execution group is merged into the current execution group. Try this with the prompt at the end of this paragraph. Place the cursor at the prompt, type Ctrl-j three or four times, and then combine the separate input lines into a single execution group. (You can place some Maple commands after the prompts either before or after you combine them into an execution group.)

```
> 
```

After you create an execution group you will quite often need to edit it. If you just need to edit one of the lines in an execution group, there is not much of a problem. The non obvious part is when you want to enter a new line somewhere into the middle of an execution group. For example, suppose we want to enter a new line after the second line in the following execution group.

```
> x^2-1;
> factor( % );
> expand( % );
```

There are several ways to do this. One way is to place the cursor at the beginning of the third line (just before **expand**) and hit the F3 key (or use the "Edit -> Split or Join -> Split Execution Group" menu item). This will split off the third line from the two above it. Then either place the cursor in the third line and type Ctrl-k, or place the cursor in the first or second line and type Ctrl-j. Either way, you get a new input line between the **factor** and **expand** commands (and remember, you can always use Ctrl-z to undo what you just did if you want to try doing something different). Then move the cursor back to the first or second line and hit F4 twice to combine all of the lines into a single execution group.

The above steps for adding a line to an execution group are kind of tedious. Here is another way to do it. In the execution group at the end of this paragraph, place the cursor just to the left of the third prompt, *between* the prompt and the bracket that marks the execution group. You can do this by

either (carefully) clicking the mouse just to the left of the prompt, or by placing the cursor just to the right of the prompt and then using the left arrow key to move the cursor to the left side of the prompt (and one more strike of the left arrow key will move the cursor to the end of the second line). With the cursor to the left of the prompt, hit the Enter key. A new line will appear in the execution group just above the line where the cursor was (and if you hit Enter again, the commands in the execution group will be executed).

```
> x^2-1;
> factor( % );
> expand( % );
```

Here is still a third way to add a line to an execution group. In the execution group at the end of this paragraph, place the cursor at the *end* of the second line. Then, while holding down the shift key, hit the Enter key (so you are typing the Shift-Enter key combination). A new line will appear below the second line. But notice that this line does not have its own prompt on its left hand edge. This new line is really a continuation of the second line.

```
> x^2-1;
> factor( % );
> expand( % );
```

Now you know how to combine lines into an execution group, split lines off of an execution group, and create new lines within an execution group. So let us turn to procedure definitions.

If a procedure definition is short, it can fit on a single line, even if it may have a few commands in it.

```
> test := proc(n::integer) 13*n; factor(n); end;
```

But even when a procedure is short, it is usually better to enter the procedure definition on several lines to make the definition easier to read. So what we want to go over is how to create and then edit a multi line procedure definition.

The following prompt has the beginning of a procedure definition after it. Place the cursor anywhere in this line and hit the Enter key. Notice that Maple gives you a new prompt, and also a warning about a premature end of input. Just ignore the warning and type the first command from the above procedure definition (the command **`13*n;`**) after the new prompt. Then hit enter again. Maple gives you another new prompt, and the warning is still there. Enter the second command from the above procedure definition (the command **`factor(n);`**) after the new prompt. Hit Enter once again and you get another new prompt (along with the warning). Now enter the closing of the procedure definition (the **`end;`**) after the prompt and hit Enter one more time. The last enter causes Maple to digest the procedure definition.

```
> test := proc(n::integer)
```

If you did not do it while you were entering the procedure definition, go back and give the two lines in the body of the definition a little bit of indentation after the prompts (two or three spaces is enough). Proper indentation goes a long way towards making procedure definitions more readable.

The above exercise shows how easy it is to enter a procedure definition. Editing a procedure

definition is no different than editing an execution group. If you want to enter a new line into the middle of a procedure definition, you can either use F3, Ctrl-j, and F4 to create the new line, or you can place the cursor to the left of a prompt and hit enter, or you can place the cursor at the end of a line and type Shift-Enter. Any one of these three methods will work and each one has its advantages and disadvantages.

The techniques mentioned here for working with execution groups and procedure definitions also work for the control statements discussed in the next chapter. In particular, if you want to create a multi-line for-loop or while-loop, you just do it the way we created a procedure definition. After you have typed the key word **for** on a line, you can keep hitting the Enter key to create new input lines until you enter the closing key word **od** (or **end do**). Once Maple sees the closing keyword (with a semi colon or colon after it), it digests the definition and executes the loop. Similarly for while-loops and if-statements. And if you should nest one control statement inside of another, then Maple needs to see closing keywords (in the right order) for both of the control statements before Maple will digest and execute the commands.

```
>
```

13.18. Online help for procedures

We covered quite a few ideas in this worksheet. The most important are

(i) defining a procedure in Maple,
(ii) formal and actual parameters,
(iii) what a procedure call is,
(iv) what a return value is,
(v) local and global variables.

Here are some references to Maple's online help about these ideas. (All in all, there is not a whole lot of online help about procedures.) The following command calls up an overview of procedures in Maple.

```
> ?procedures
```

The next command calls up an overview of Maple functions and their relationship with procedures.

```
> ?functional
```

The next help page has some information about the various options that can be declared in a procedure definition.

```
> ?options
```

The remember option has it own help page.

```
> ?remember
```

The next page describes how Maple does parameter passing in a procedure call.

```
> ?parameter
```

This page has some information about return values for procedures.

```
> ?RETURN
```

The next page gives some information about type checking of the actual parameters passed to a procedure.

```
> ?procedure,paramtype
```

When doing type checking of actual parameters, the types used are usually structured types (as opposed to the more basic surface types).

The next command brings up a Maple worksheet, not a help page, that details how Maple handles local and global variables. This example worksheet is informative but not easy reading.

```
> ?examples,lexical
```

The next page is the help page that describes the local variables **args** and **nargs**, i.e., the expression sequence of actual parameters passed to a procedure from a procedure call and the number of actual parameters.

```
> ?args
```

Many Maple commands have options that are given in the form of equations (like the **plot** command's option **style=point**). Maple has a special boolean function, **hasoption**, that makes it easy for a procedure to find these options in its **args** expression sequence.

```
> ?hasoption
```

Along with **args** and **nargs**, there is a third special local variable for every procedure call, **procname**, the name by which the procedure was invoked.

```
> ?procname
```

The next command brings up a Maple worksheet that is part of the New User's Tour. This worksheet is a *very* brief introduction to procedures.

```
> ?newuser,topic10
```

The next command brings up a help page that is an index to many of the Maple commands that are specific to writing procedures and doing Maple programming.

```
> ?index,procedures
```

If you are writing your own procedures and the procedures get a bit long or complicated, then you are sure to have a few "bugs" in your procedure definitions. Finding mistakes (i.e. bugs) in a procedure can be difficult. Maple provides some tools for "debugging" your procedure. These tools can also be used to watch, or examine, how a correct procedure does its work. Maple's main tool for finding mistakes in a procedure is its "interactive debugger". The next three commands call up help pages about the debugger.

```
> ?debugger
> ?showstat
> ?stopat
```

Another useful command for debugging a procedure, or for just watching a procedure work, is the **printlevel** facility. This lets you see, for example, the results of all the commands in a procedure, not just the results of the return statement.

```
> ?printlevel
```

When procedures start to get complicated, they quite often do their work by calling other procedures. The **trace** facility lets you see how one procedure calls another procedure.

```
> ?trace
```

The following help page describes a special command that returns a very abstract representation of a procedure data structure and a procedure definition. This command is meant for very sophistcated Maple programming, but it can also be used for getting an understanding of how Maple represents procedures. Using this command you can notice, for example, that all variable names are really stripped out of the body of a procedure definition. The names of local variables and formal parameters are not really needed by a procedure. But they are stored in the procedure data structure (mostly for debugging purposes).

```
> ?procbody
```

The next help page gives some information about how to read the representation of a procedure returned by **procbody**.

```
> ?procmake
```

And the next help page describes another command that returns another kind of an abstract representation of a Maple procedure.

```
> ?codegen,maple2intrep
```

It is very difficult to write long procedures directly into a Maple worksheet. When people write long procedures (that is, anything more than about 10 or 15 lines) it is usually easier to write the procedure in file using a text editor and then read the file into Maple so you can test the procedure. The following help page describes how to read a procedure into Maple from a file.

```
> ?procfile
```

```
>
```