Maple for Math Majors

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

15. Manipulating Data Structures with Procedures

15.1. Introduction

Almost all Maple commands are really procedures written in the Maple programming language. Most of the symbolic algebra commands (like **factor**, **combine**, **simplify**) are procedures that manipulate data structures in order to perform the symbolic manipulations. In this worksheet we look at examples of how a procedure can manipulate a data structure to accomplish a symbolic calculation. This will give us an idea of how Maple does its job.

```
> 
```

15.2. Differentiating a monomial

The derivative with respect to x of the monomial $c\,x^n$ is $c\,n\,x^{(n-1)}$ So, for example, the derivative of $3\,x^5$ is $15\,x^4$. Here is a procedure that implements this rule.

```
> diff_term := proc( term )
> 
  op(1,term)*op(2,op(2,term))*op(1,op(2,term))^(op(2,op(2,term))-
  1)
> end;
```

Before explaining how the procedure works, let us try it out.

```
> diff_term( 3*x^5 );
> diff_term( 2*t^3 );
> diff_term( a*x^n );
> diff_term( x^3 );  # We will fix this bug later.
```

The procedure **diff_term** has a single formal parameter, **term**, which the procedure assumes is a data structure representing one term of a polynomial (i.e., a monomial). The procedure does its work by breaking up the data structure named by **term** into its basic parts and then reconstructing those parts into the derivative of **term**. For example, if **term** has the value **3*x^5**, then the pieces of **term** are **3**, **x**, and **5**, and here is how those pieces are built up into the derivative **3*5*x^4**.

```
> term := 3*x^5;
> op(1,term), op(2,op(2,term)), op(1,op(2,term)),
  op(2,op(2,term))-1;
> op(1,term)*op(2,op(2,term))*op(1,op(2,term))^(op(2,op(2,term))-
  1);
```

Here is a way to watch how the procedure **diff_term** works. Let us redefine the procedure using right-quotes to delay the evaluation of certain results.

```
> diff_term := proc( term )
>   ' op(1,term)*'op'(2,op(2,term)) *
>     'op'(1,op(2,term))^('op'(2,op(2,term))-1) '
> end;
```

Notice that we delayed the evaluation of the entire command in the body of the procedure and we further delayed the evaluation of three of the **op** commands. Let us apply this modified version of **diff_term** to a monomial and watch the steps taken in the evaluation of the derivative.

```
> diff_term( 3*x^5 );
> %;
> %;
>
```

Now let us look at the bug that showed up earlier. Something went wrong when the input was **x^3**. Let us see what.

```
> term := x^3;
> op(1,term); op(2,op(2,term)); op(1,op(2,term));
  op(2,op(2,term))-1;
```

Now we see what went wrong. There is no **op(2,op(2,term))** when **term** is **x^3** (why?). We need a special case to handle monomials without a leading coefficient. Here is another version of **diff_term**.

```
> diff_term := proc( term )
>   if type( term, `^` ) then
>     op(2,term)*op(1,term)^(op(2,term)-1)
>   else
>
  op(1,term)*op(2,op(2,term))*op(1,op(2,term))^(op(2,op(2,term))-
  1)
>   fi
> end;
```

Notice how we have used a conditional statement to break up our calculation into different cases. The boolean part of the conditional statement uses the **type** command. This command is used to ask if a particular data structure is of a particular data type, and the result is either **true** or **false**. If **term** is of type **`^`**, then we are assuming that **term** is of the form **x^n** (i.e., the monomial has no coefficient), otherwise we are assuming that **term** is of the form **c*x^n**.

Let us see if this works.

```
> diff_term( x^3 );  # Yea!
> diff_term( x^1 );  # Opps.
```

We were assuming that the variable **x** has an exponent other than 1, but it need not. If the exponent is 1 then Maple (automatically) simplifies the expression to not have an exponent. We also get an

error with the following input.

```
> diff_term( 3*x^1 );
```

So we need another special case. Let us try again.

```
> diff_term := proc( term )
>   if type( term, {name,numeric}&*name ) then
>     op(1,term)
>   else
>     if type( term, `^` ) then
>       op(2,term)*op(1,term)^(op(2,term)-1)
>     else
>
  op(1,term)*op(2,op(2,term))*op(1,op(2,term))^(op(2,op(2,term))-
  1)
>     fi
>   fi
> end;
```

This version handles three special cases. The first case is new and the last two are the same cases as in the previous version. Notice that the first **`type`** command has a special data type in it, something called a **structured data type**. The structured data type **`{name,numeric}&*name`** matches any data structure which is of the form "a name or number times a name", e.g., either **`c*x`** or **`3*x`**. In both cases the derivative is the "constant" in front of the "variable". Let us try out this new version.

```
> diff_term( x^3 );
> diff_term( 3*x );
> diff_term( -x );
> diff_term( x );  # Opps again.
```

Another special case that we have to worry about. One more time.

```
> diff_term := proc( term )
>   if type( term, name ) then
>     1
>   else
>     if type( term, {name,numeric}&*name ) then
>       op(1,term)
>     else
>       if type( term, `^` ) then
>         op(2,term)*op(1,term)^(op(2,term)-1)
>       else
>         op(1,term)*op(2,op(2,term))
>               * op(1,op(2,term))^(op(2,op(2,term))-1)
>       fi
>     fi
>   fi
> end;
```

Give this version a try.

```
> diff_term( x^3 );
> diff_term( 3*x );
> diff_term( a*x );
> diff_term( x );
> diff_term( 10 );  # Oh no.
```

If we stop and think about it, this is our last special case. We are trying to differentiate terms of the form **a*x^n**, and this term can be in any of the following special forms, each of which is really a different kind of data structure with a different (structured) data type.

```
>
```

form	data type
a	numeric or name
x	name
a*x	{numeric or name}*name
x^n	`^`
a*x^n	{numeric or name}*name^{numeric or name}

```
>
```

So we really need a special case for each of the five forms that the input term can take.

```
> diff_term := proc( term )
>   if type( term, numeric ) then
>     0
>   else
>     if type( term, name ) then
>       1
>     else
>       if type( term, {name,numeric}&*name ) then
>         op(1,term)
>       else
>         if type( term, `^` ) then
>           op(2,term)*op(1,term)^(op(2,term)-1)
>         else
>           op(1,term)*op(2,op(2,term))
>                 * op(1,op(2,term))^(op(2,op(2,term))-1)
>         fi
>       fi
>     fi
>   fi
> end;
```

Let us try this out.

```
> diff_term( a*x^n );
> diff_term( x^3 );
> diff_term( 3*x );
> diff_term( a*x );
```

```
> diff_term( x );
> diff_term( 1 );
> diff_term( a );  # What should we get here?
```

Let us not worry about the last one for now. Instead, let us look at a slightly cleaner way to write **diff_term**. Nested if-then-else-fi statements are so common that there is a nice shorthand for them. This shorthand makes nested if-then-else-fi statements easier to read, it gets rid of many levels of indentation, and it gets rid of all the trailing **fi**'s. Here is **diff_term** rewritten using this shorthand, the if-then-elif-then-else-fi construction.

```
>
> diff_term := proc( term )
>   if type( term, numeric ) then
>     0
>   elif type( term, name ) then
>     1
>   elif type( term, {name,numeric}&*name ) then
>     op(1,term)
>   elif type( term, `^` ) then
>     op(2,term)*op(1,term)^(op(2,term)-1)
>   else
>
    op(1,term)*op(2,op(2,term))*op(1,op(2,term))^(op(2,op(2,term))-
    1)
>   fi
> end;
```

Notice how this version is much easier to read. Let us try this version to make sure that it still works.

```
> diff_term( a*x^n );
> diff_term( x^3 );
> diff_term( 3*x );
> diff_term( a*x );
> diff_term( x );
> diff_term( 1 );
>
```

Exercise: Use the delayed evaluation trick in the body of the most recent version of **diff_term** to watch how the procedure evaluates the derivatives of various monomials.

```
>
```

Exercise: Write a procedure **int_term** that finds an antiderivative of a monomial. If you model your procedure on **diff_term**, then your **int_term** should also work with "monomials" with negative coefficients. Make your version of **int_term** compute the correct antiderivative for the

special case of $\frac{1}{x}$.

[>

Now that we have a procedure that differentiates monomials, it would be nice if we could use it to define a procedure that differentiates polynomials, which are just sums of monomials. In the next section, we show how to define such a procedure. The main tool that we need is a special data structure manipulating command called **map**.

[>

[>

15.3. Differentiating a polynomial; the map command

What about differentiating a polynomial, which is a sum of monomials (i.e. terms). The derivative with respect to x of a sum of terms is the sum of the derivatives of each term. So, for example, the derivative of $3\,x^2 + 5\,x^3$ is $6\,x + 15\,x^2$. How can we use our **diff_term** procedure to accomplish this? To find out how, we need to first look at a new Maple command.

Maple provides a procedure called **map** which applies a given procedure to all of the pieces of a given data structure and returns the result in the same kind of data structure as the input. For example the next command will apply the **sqrt** function to each number in the given list and return the results in a list.

```
[ > map( sqrt, [9, 16, 25, 36, 49, 64, 81] );
```

Here is an interesting trick. If we delay the evaluation of the **sqrt** function, then we can see the intermediate step in the calculation of the **map** command.

```
[ > map( ''sqrt'', [9, 16, 25, 36, 49, 64, 80] );
```

Here we see that the **map** command produced a list of the **sqrt** function applied to every number from the original list.

The next command applies the squaring function to a set of numbers and returns the results in a set (why are there only four numbers in the result?). Notice how the squaring function that we use here is an anonymous function.

```
[ > map( x->x^2, {-3, -2, -1, 0, 1, 2, 3} );
```

The next command applies the delayed evaluation trick to see the intermediate result.

```
[ > map( ''x->x^2'', {-3, -2, -1, 0, 1, 2, 3} );
```

Here we see that **map** has produced a set containing the squaring function applied to each of the seven elements of the original set.

The next command produces a list of values of $\sin(x)$ for x from 0 to $2\,\pi$ in steps of $\pi/6$. The command does this by applying the **sin** function to every number in the list produced by the **seq** command.

```
> map( sin, [seq( (i*Pi)/6, i=0..12)] );
```

The next example returns the sum of the coefficients of the polynomial (why?).

```
> map( coeffs, x^4 + 2*x^3 + 3*x^2 + 4*x + 5 );
```

Here is what the intermediate step from this last calculation looks like.

```
> map( ''coeffs'', x^4 + 2*x^3 + 3*x^2 + 4*x + 5 );
```

The next example uses **diff_term** (from the last section) and returns the derivative of the polynomial.

```
> map( diff_term, x^4 + 2*x^3 + 3*x^2 + 4*x + 5 );
```

Here is the intermediate step.

```
> map( ''diff_term'', x^4 + 2*x^3 + 3*x^2 + 4*x + 5 );
```

The **map** command is an example of a powerful Maple command for manipulating data structures. It has a lot of uses in Maple.

Exercise: It is important to remember that the first operand of the **map** command should be a function or a procedure. For example, what if we redo the above squaring example, but we represent the squaring function with an expression instead of with a Maple function? Explain the result of the next command.

```
> map( x^2, {-3, -2, -1, 0, 1, 2, 3} );
```

This example shows that it is not always possible to interchange "mathematical functions as expressions" with "mathematical functions as Maple functions". There are times when we must work with Maple functions.

```
>
```

Let us define a new procedure **diff_poly** that uses **map** and **diff_term** (from the last section) to differentiate a polynomial (i.e. a sum of terms). We use **map** to apply **diff_term** to each term of an input polynomial and get a result that is also a polynomial.

```
> diff_poly := proc( poly )
>    map( diff_term, poly )
>  end;
```

Let us try it out.

```
> diff_poly( x^4+2*x^3+3*x^2+4*x+5 );
> diff_poly( 10+x-5*x^2+x^4 );
> diff_poly( 5*x^5-4*x^4+3*x^3-2*x^2-2*x^(-2) );
> diff_poly( a*x^n );  # Opps
> diff_poly( 3*x^n );  # Opps again
```

Now something is broken. The last two results are not right even though we had those working in the last section. Let us analyze a call to **diff_poly** with the actual parameter **a*x^n**. The **map** command in the body of **diff_poly** will apply the procedure **diff_term** to each piece of the data structure **a*x^n**.

```
> map( diff_term, a*x^n );
```

But `a*x^n` is a data structure of type `` `*` `` with operands `a` and `x^n` so `diff_poly` will return the product of `diff_term(a)` (which is `1`) and `diff_term(x^n)` (which is `n*x^(n-1)`) to get the final result `n*x^(n-1)`. Here is the delayed evaluation trick so that we can see what `map` is doing.

```
> map( ''diff_term'', a*x^n );
> %;
```

This is not what we want.

```
>
```

Exercise: Analyze the call to `diff_poly` with the actual parameter `3*x^n` to see exactly what happened in that case.

```
>
```

So the problem is that `diff_poly` assumes (unwisely) that its input is a sum of terms. We need a conditional statement inside `diff_poly` to check the data type of the input. If the input is a `` `+` `` data structure then it is a sum of terms, otherwise we will assume it is a single term.

```
> diff_poly := proc( poly )
>   if type(poly, `+`) then
>     map( diff_term, poly )
>   else
>     diff_term( poly )
>   fi
> end;
```

Let us try this new and improved version.

```
> diff_poly( a*x^n );
> diff_poly( 3*x^n );
> diff_poly( 3+2*x+a*x );
> diff_poly( 10+x-5*x^2+x^4 );
> diff_poly( 5*x^5-4*x^4+3*x^3-2*x^2-2*x^(-2) );
```

Notice that the last example was not even a polynomial (why?), but it worked. Let us try our differentiation procedure on a randomly chosen polynomial in *x*.

```
> randpoly( x );
> diff_poly( % );
```

So far so good. However, the next three examples all give erroneous answers.

```
> diff_poly( x^3 + z^2 );      # This result doesn't really make
  sense.
> diff_poly( x*y^2 + y*x^2 );  # Neither does this one.
> diff_poly( cos(x) );         # And this doesn't work.
```

Our `diff_poly` procedure was not designed to handle these kinds of inputs, e.g., polynomials in several variables or arbitrary functions. But our procedure should not return erroneous results or cryptic error messages for inputs it was not designed for. It would be better if our procedure had Maple do some "type checking" of the procedure's input so that Maple could inform users of the

proper kind of input. We want to make sure that the procedure's input is a polynomial in one variable. Here is a version of **diff_poly** that only accepts univariate polynomials. The word **polynom** means a polynomial data type to Maple and **polynom(name)** means a polynomial of a single variable.

```
> diff_poly := proc( poly::polynom(name) )
>   if type(poly, `+`) then
>     map( diff_term, poly )
>   else
>     diff_term( poly )
>   fi
> end;
```

Let us see if this helps.

```
> diff_poly( cos(x) );
> diff_poly( x^2+y^2 );
```

It seems to help, but now something else is wrong.

```
> a*x^n;
> diff_poly( % );  # now this is broken.
```

Our strategy for designing the **diff_term** and **diff_poly** procedures is not really very good. That is why we are having trouble getting some things just right. But the examples in this section have given you an idea of how a procedure can manipulate a data structure that represents a polynomial to build up a new data structure that represents the polynomial's derivative. We have seen that we can use type checking to test the kind of inputs our procedure accepts but we have also seen how subtle and difficult it can be to really get things right when we want to do a symbolic mathematical manipulation.

```
>
```

Exercise: Use the **map** command and your **int_term** procedure from the last section to write a procedure **int_poly** that finds an antiderivative of a polynomial.

```
>
```

Let us see how Maple's own differentiation command works. If we are differentiating an expression, Maple requires that we specify the independent variable that we wish to differentiate with respect to. This requirement solves some of the problems we are having with **diff_poly**, and it is one of the main difference between our **diff_poly** and Maple's **diff**. (The other main differences are that **diff** works on any expression, not just polynomials, and **diff** does not have any bugs in it.)

```
> diff( 3*x^2, x );
> diff( 3*t^2, t );
> diff( 3*t^2, x );
> diff( a*x^n, x );  # Does this give the right answer?
> diff( a, x );
> diff( a, a );
> diff( x^3 + z^2, x );
```

```
> diff( exp(x)*cos(x), x );
> diff( f(x)*g(x), x );  # diff knows the product rule.
> diff( f(g(x)), x );    # And diff knows the chain rule.
>
```

In a later section of this worksheet we will see how to modify our **diff_poly** procedure to make it act more like Maple's **diff** command. Before doing that, let us look at some examples of real Maple procedures that are written in the Maple programming language.

```
>
```

```
>
```

15.4. Some real Maple procedures

The examples from the previous two sections demonstrate two main points. First, to show how some of Maple's data structure manipulating commands are used. Second, to show that by using data structure manipulating commands, we can write interesting commands that do mathematics symbolically. The basic ideas behind these examples is in fact the basis for how much of Maple itself is written. In this section we show how most Maple commands are really procedures written in the Maple programming language and that many of these procedures do data structure manipulations like the ones that we did in the previous two sections.

```
>
```

If most Maple commands are really procedures written in the Maple programming language, then we should be able to print out the definition of a command and see how it is written. So let us try to look at the definition of a real Maple command.
To do this we need to use a special command, Maple's **interface** command. Let us try to look at the definition of the **factor** command.

```
> print( factor );
```

That did not tell us much. The next command will help.

```
> interface( verboseproc=2 );
```

Now let us try printing the definition of **factor** again (the definition is pretty long).

```
> print( factor );
```

If you look carefully at the Maple code in the **factor** command, you will see that it is similar in structure to our **diff_term** procedure. There are a lot of nested if-then-else-fi statements that check for specific data types.

```
>
```

Here is the definition of the **simplify** command.

```
> print( simplify );
```

If you want to see a really long Maple program, look at the source code to **plot** (recall that key combination Ctrl-Z can be used to make the output go away after it has been displayed).

```
> print( plot );
```

Looking at the source code of Maple commands can be intriguing. For example, what are those two very large integers doing in the source code for **isprime**?

```
> print( isprime );
>
```

We cannot examine the source code for every one of Maple's commands. Some commands are "built in" commands, which means that they are written in the C programming language instead of in Maple's own programming language. You cannot see the source code for these built in commands. Here are a few examples of built in commands.

```
> print( diff );
> print( expand );
> print( eval );
```

Maple knows when a procedure is built in because a built in procedure has the **builtin** option in the third operand of its **procedure** data structure.

```
> op( 3, eval(diff) );
>
```

Exercise: The command **nextprime** has a very short Maple program.

```
> print( nextprime );
```

Explain in detail exactly how this program works. In particular, which lines of code are responsible for each of the following results?

```
> nextprime( -5 );
> nextprime( 19 );
> nextprime( hello );
> nextprime( 19/2 );
>
```

Exercise: Both this and the next exercise use the definition of the **ln** procedure.

```
> print( ln );
```

Find the part of the code for **ln** that implements the rule $\ln(b^a) = a\ln(b)$. (Hint: There are actually two different places in the code that implement this rule, but for different data types of a and b.)

```
>
```

Exercise: What do the third and fourth elif-parts do in the code for the **ln** function? What logarithm rules do they implement?

```
>
```

Exercise: Examine the code for the function **sqrt** and find the part that is responsible for rationalizing the square root of fractions such as the following.

```
> sqrt(1/2), sqrt(1/3), sqrt(3/5);
> print( sqrt );
>
```

Exercise: This exercise is about the source code for the **sin** function.

```
> print( sin );
```

Find the part of the code for **sin** that is responsible for each of the following calculations.

```
> sin(-x);
> sin(arctan(y));
> sin(z+Pi/2);
```

Recall that the remember table for **sin** allows Maple to return the following symbolic results.

```
> sin(Pi/5);
> sin(2*Pi/5);
```

If we examine the remember table for **sin**, we see that there is no entry in it for **3*Pi/5** and yet Maple knows the following symbolic result.

```
> sin(3*Pi/5);
> op( 4, eval(sin) );
```

Use the source code for **sin** to explain exactly how Maple was able to calculate the symbolic result for **sin(3*Pi/5)**.

```
>
```

Exercise: This exercise uses the source code for **ln** again.

```
> print( ln );
```

Click on the source code for **ln** so that it is highlighted. Then use the Copy item in the Edit menu to put this code in the clipboard. Now use the Paste item from the Edit menu to paste the code for **ln** into the next prompt right after the **my_ln :=**. Now edit the error message for **ln(0)** (you need to find this error message first). Change it to anything you want. Put a semi colon at the very end of the code. Then hit the enter key (with the cursor still in the new code for **ln**) to redefine the **ln** function and give the redefinition the name **my_ln** (the name **ln** is "protected", so it is difficult, but not impossible, to use that name). Then execute the **my_ln(0)** command at the second prompt below to see your new error message.

```
> my_ln :=
> my_ln(0);
```

Using this technique, any Maple command, except for the built in ones, can be modified to your heart's content. This allows people with specialized mathematical needs to modify Maple for their own purposes. It also allows users to fix bugs in Maple. Occasionally, in the Maple Users Group e-mail mailing list, people will post bug fixes for Maple that one can implement by modifying a command's code, just as you did here.

```
>
```

How does Maple know which procedure definitions can be seen by default and which ones need **verboseproc** set to 2? Recall that a **procedure** data structure has an options operand, which is the third of seven operands in a **procedure** data structure. One of the possible options is called **Copyright**. If a **procedure** data structure has a **Copyright** option in its third operand, then we need **verboseproc** set to 2 in order to see the procedure's definition.

```
> op( 3, eval(simplify) );
```
Since **simplify** has the **Copyright** option, we need **verboseproc** set to 2 to view its definition. All of the commands in the Maple library have the **Copyright** option. We do not however need **verboseproc** set to 2 to be able to see the **procedure** data structure for a procedure. Every procedure data structure can be examined using the **op** command (recall that a procedure's definition is not part of a **procedure** data structure).
```
>
```

Exercise: Create a simple procedure and give it the **Copyright** option. Recall that options, like **Copyright**, **remember**, and **trace**, are declared right after the local and global variable declarations. Try to view the definition of your procedure with **verboseproc** set to 1 and 2.
```
>
```

Finally, let us make sure that we set the **interface** back to its default setting.
```
> interface( verboseproc=1 );
```

```
>
```

15.5. "Reversing" a polynomial

We know that a polynomial like $10\,x^3 - 3\,x^2 + 2\,x - 5$ is represented in Maple by a data structure. Let us write a procedure that takes as its input a data structure representing a polynomial and manipulates the data structure so as to reverse the order of the coefficients in the polynomial, so $10\,x^3 - 3\,x^2 + 2\,x - 5$ will become $-5\,x^3 + 2\,x^2 - 3\,x + 10$.

But before trying that, let us write a procedure that reverses the elements of a simpler data structure, a list.
```
> reverse_list := proc( my_list )
>   local i, N;
>   N := nops(my_list);
>   [ seq( op(N-i, my_list), i=0..N-1) ]
> end;
```
Let us try it out before we explain it.
```
> reverse_list( [1,2,3,4,5] );
> [ seq( 2^i, i=-4..4) ];
> reverse_list( % );
> empty_list := [];
> reverse_list( empty_list );
```

To explain **reverse_list** let us start with a simple list.
```
> simple_list := [a,b,c];
```
There are three elements in this list. Notice what the following commands do.
```
> op(3-0, simple_list); op(3-1, simple_list); op(3-2,
```

```
   simple_list);
```
We can use the **seq** command to abbreviate this.
```
> seq( op(3-i, simple_list), i=0..2 );
```
Put square brackets around the last command so that the result is a list.
```
> [ seq( op(3-i, simple_list), i=0..2 ) ];
```
Finally, let us notice that **3** is the same as **nops(simple_list)** and **2** is the same as **nops(simple_list)-1**.
```
> N := nops(simple_list);
> [ seq( op(N-i, simple_list), i=0..N-1 ) ];
```
Now try a less simple **simple_list**.
```
> simple_list :=
  [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z];
> N := nops(simple_list);
> [ seq( op(N-i, simple_list), i=0..N-1 ) ];
```
(Why is there a 17 in the reversed list? How can we get rid of it?)
```
>
```

Here is a slightly different way to write the command that reverses a list. What has been changed and why?
```
> [ seq( op(-ii, simple_list), ii=1..N ) ];
```
Here is an even more brief way to write the command that reverses a list.
```
> [ seq( simple_list[-ii], ii=1..N ) ];
```
Here is still another way to write this.
```
> [ simple_list[-ii]$ii=1..N ];
>
```

Now let us return to our original problem and use a similar idea with a polynomial. The main difference here is that the data structure is a bit more complicated.
```
> reverse_coef := proc( poly )
>   local i, j, N;
>   N := nops(poly);
>   [ seq( op(1, op(N-i+1, poly))*op(2, op(i, poly)), i=1..N ) ];
>   add( op(j, %),  j=1..N );
> end;
```
Let us test this on some polynomials.
```
> 101*x + 102*x^2 + 103*x^3 + 104*x^4 + 105*x^5;
> reverse_coef( % );
> 5*x^2 + 2*x + 9*x^3;
> reverse_coef( % );
```
We can simplify our procedure a bit. Instead of creating a sequence and then adding the terms of the sequence to form the final polynomial, we can use the **add** command to simultaneously create the sequence of terms and add them.
```
> reverse_coef := proc( poly )
```

```
>   local i, N;
>   N := nops(poly);
>   add( op(1, op(N-i+1, poly))*op(2, op(i, poly)), i=1..N );
> end;
```

Let us test this again.

```
> 101*x + 102*x^2 + 103*x^3 + 104*x^4 + 105*x^5;
> reverse_coef( % );
> 5*x^2 + 2*x + 9*x^3;
> reverse_coef( % );
```

Let us try the procedure on a randomly chosen polynomial.

```
> randpoly( z );
> reverse_coef( % );
```

What went wrong? Here are the crucial lines from the body of the procedure with some delayed evaluation added so that we can observe how the terms are evaluated.

```
> poly := randpoly( z );
> N := nops(poly);
> add( 'op'(1, op(N-i+1, poly))*''op''(2, op(i, poly)), i=1..N )
  ;
> %;
> %;
```

If we look at each of the six **op** commands in the last output, we see that the very last term in the sum has an **op** command with an invalid index. So there was a problem with the constant term in the polynomial. We assumed that every term in the polynomial has the form $c\,x^n$ but a constant term does not have that form. In fact, here is another polynomial that will cause an error.

```
> 3*x^3+x^2-5*x;
> reverse_coef( % );
```

The polynomial had a term of the form x^2 without an explicit coefficient. Let us see in detail how the procedure evaluated the polynomial.

```
> poly := 3*x^3+x^2-5*x;
> N := nops(poly);
> add( 'op'(1, op(N-i+1, poly))*''op''(2, op(i, poly)), i=1..N )
  ;
> %;
> %;
```

We see that the middle term **x^2** was not handled properly at all. Neither the coefficient nor the variable part were extracted properly from this term. (Why did the final result have only two terms?)

In our procedure **reverse_coef** we need to be more careful about selecting the coefficient part and the variable part from each term of the polynomial. We need our procedure **reverse_coef** to have a form like the following, where we need to define new procedures **coef_part** and **var_part**.

```
> reverse_coef := proc( poly )
```

```
>   local i, N;
>   N := nops(poly);
>   add( coef_part(op(N-i+1, poly))*var_part(op(i, poly)), i=1..N
  );
> end;
```

Here is what a call to `reverse_coef` looks like at this point.

```
> 3*x^3+x^2-5*x;
> reverse_coef( % );
> randpoly( x, degree=10, terms = 3 );
> reverse_coef( % );
```

Here is a definition for `coef_part`.

```
> coef_part := proc(term)
>   if type(term, constant) then term
>   elif type(term, `^`) then
>     1
>   else
>     op(1, term)
>   end
> end;
```

And here is a definition for `var_part`.

```
> var_part := proc(term)
>   if type(term, constant) then 1
>   elif type(term, `^`) then
>     term
>   else
>     op(2, term)
>   end
> end;
```

Try them out.

```
> 3*x^3+x^2-5*x;
> reverse_coef( % );
> 5*x^5+2*x^3+x^2-6*x;
> reverse_coef( % );
>
```

Exercise: Each of `coef_part` and `var_part` contains a single if-then-elif-else-statement in its body. Explain in detail what kinds of monomial terms each of the three clauses deals with.

```
>
```

We still have a problem with `reverse_coef`. Consider the next example.

```
> 3*z^3+2*z^2+z;
> reverse_coef( % );
>
```

Exercise: The error message in the last example identified the procedure `var_part` as the source of the problem. Here is a rewritten `var_part`. What was wrong with the previous version and why does this version fix the problem?

```
> var_part := proc(term)
>   if type(term, constant) then 1
>   elif type(term, name) then term
>   elif type(term, `^`) then
>     term
>   else
>     op(2, term)
>   end
> end;
```

But we still have a problem with `reverse_coef`. Consider the last example again.

```
> 3*z^3+2*z^2+z;
> reverse_coef( % );
```

There is still a problem with the procedure `coef_part`. Find the problem and fix it.

```
>
```

There is one last issue to consider with our procedure `reverse_coef`. Consider the next example.

```
> x^2-2*x-3;
> reverse_coef( % );
```

The result from `reverse_coef` is correct, but it may not look like what we were expecting. In Maple, the way a polynomial gets displayed is session dependent and it can change from one session to another. In my version of Maple, the last result has the constant term at the beginning of the polynomial instead of at the end where we usually expect it to be. We do not have much control over how Maple decides to order the terms of a polynomial. We can use the `sort` command to order the terms in descending order of degrees.

```
> sort( % );
```

Usually, when we enter a polynomial, Maple will display the polynomial with the terms in the order that we entered them in.

```
> 4*x^3-7+5*x^8;
```

But now consider the next example

```
> 5*x^8+4*x^3-7;
```

In my session, Maple "remembered" the order of the terms of this polynomial from the last entry and displayed the polynomial as before. So Maple will sometimes change the order of the terms of a polynomial.

Now what should be the result of the next command? The actual result will depend on how Maple orders the terms of the input polynomial.

```
> reverse_coef( 5*x^8+4*x^3-7 );
```

The result from `reverse_coef` is correct (what did it do?). But this behavior of Maple's with

respect to ordering the terms of a polynomial can make it very difficult to predict the results of **reverse_coef**. One way to avoid this confusion would be for **reverse_coef** to **sort** its input polynomial and also **sort** its return polynomial just before returning it.

```
[ >
```

Exercise: Add **sort** commands to the definition of **reverse_coef** as described in the last paragraph. Try out the new version of the procedure.

```
[ >
```

Exercise: Maple has a command **coeffs** that can be used to return the coefficient of a monomial. We should be able to use this command to replace our procedure **coef_part** in the definition of **reverse_coef**. But the **coeffs** command needs to be told the variable with respect to which we want the coefficient.

```
[ > coeffs( 12*x*y, x );
[ > coeffs( 12*x*y, y );
```

Our procedure **reverse_coef** is designed to work with any polynomial in one variable, so we do not know ahead of time what variable is in the polynomial. However, the Maple command **indets** (for "**indet**erminate**s**") will return the set of variables used in an expression. Rewrite **reverse_coef** to use **indets** and **coeffs** instead of **coef_part**.

```
[ >
```

Exercise: Notice in the online documentation that the Maple command **coeffs** has a "call by name" parameter that can be used to return the part of a monomial that the coefficient is "in front of".

```
[ > coeffs( 12*x*y, x, 't' );
[ > t;
[ > coeffs( 12*x*y, y, 't' );
[ > t;
```

In the last exercise you used **coeffs** to eliminate the procedure **coef_par** from the definition of **reverse_coef**. Now use **coeffs** to also eliminate the procedure **var_part**.

```
[ >
```

Exercise: Rewrite **reverse_coef** so that it works with multivariate polynomials. Define **reverse_coef** so that it expects a second parameter that specifies the variable with respect to which it reverses the coefficients. Consider whether you should use Maple's **collect** and **sort** commands.

```
[ >
```

```
[ >
```

15.6. Teaching Maple new tricks

One feature that makes Maple so powerful for doing symbolic mathematics is that many of Maple's

most important commands are "extensible". What this means is that we can extend the capabilities of these commands by writing special "interface" procedures. In this section we give examples of extending the capabilities of the **expand** and **simplify** commands.

Many well known mathematical functions have special algebraic properties that Maple is aware of and can make use of in its symbolic manipulations. Here are some examples that use the **expand** command.

```
> abs(x*y);
> expand( % );
> exp(x+y);
> expand( % );
> sin(a+b);
> expand( % );
```

Maple's knowledge of these properties is built into the **expand** procedure. If we want to teach **expand** a new identity for some function, it would be impractical to try to modify the code for **expand** (especially since **expand** is a built in command and so it is not written in Maple's own programming language). So we need a way to inform **expand** of an identity that is not already built into it. Let us look at an example of how we do this.

```
>
```

Let us suppose there is a function h for which the following identity is true.

$$\mathrm{h}(x+y)=\frac{\mathrm{h}(x)}{\mathrm{h}(y)^2}$$

Let us teach **expand** about this identity. Maple provides an "interface" to the **expand** procedure that allows us to extend its capabilities. The way that we use this interface to teach the **expand** procedure about our function h is we define a procedure called **`expand/h`**. When we call the **expand** procedure and it finds a function call of the form **h(u)** (for any expression **u**), the **expand** procedure looks to see if there is a definition for the procedure **`expand/h`** and if there is one, **expand** calls **`expand/h`(u)**, which should return the appropriate expansion of **h(u)**, and then **expand** replaces the expression **h(u)** with the result returned by **`expand/h`**.

```
>
```

Before doing the more complicated example of h, let us start off with a simple example. Suppose we want to teach **expand** about a function g that has the (silly) property that g(x) expands to g($2\,x$)/2 for any input x. Here is a definition for the procedure **`expand/g`** that implements this simple expansion.

```
> `expand/g` := proc( u )
>   'g'(2*u)/2;
> end;
```

So now whenever the **expand** procedure comes across a function call of the form **g(u)** (for any expression **u**), **expand** makes the procedure call **`expand/g`(u)**, which returns the expression

`g(2*u)/2`, and then `expand` replaces the original expression `g(u)` with its expansion `g(2*u)/2`.

```
> expand( g(a) );
> expand( % );
> expand( exp(g(x^2)) );
```

So far so good. But the next two examples demonstrate a problem with our procedure `` `expand/g` ``.

```
> expand( g(exp(x+y)) );
> expand( g(g(y)) );
```

Notice that `expand` did not expand the exponential inside of `g` nor did it expand the `g` inside of `g`. Here is how we can fix this. We put an extra (recursive) call to `expand` inside of the return value of `` `expand/g` ``. This causes `expand` to keep on working on whatever is inside the expanded function call `g(2*u)/2`.

```
> `expand/g` := proc( u )
>   'g'(expand(2*u))/2;
> end;
```

Let us try this new version.

```
> expand( g(exp(x+y)) );
> expand( g(g(y)) );
```

Why did that not work? Let us try an experiment. Restart Maple.

```
> restart;
```

Define `` `expand/g` `` again.

```
> `expand/g` := proc( u )
>   'g'(2*expand(u))/2;
> end;
```

And now try it again.

```
> expand( g(exp(x+y)) );
> expand( g(g(y)) );
> expand( % );
```

The problem was that the procedure `expand` was "remembering" (using a remember table) the previous, erroneous, results about `g`. Look at the following remember table for `expand`. Notice how it contains the results from the last two calls to `expand`.

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

When we redefine `` `expand/g` ``, this does not cause the remember table for `expand` to be erased. Subsequent calls to `expand` can get their results directly from the remember table without ever calling the redefined `` `expand/g` ``.

```
>
```

Here are two examples in which `expand` produces a result that we are probably not expecting.

```
> expand( exp(g(x)+z) );
> expand( sin(g(x)+z) );
```

Notice that in each of these examples the expansion of `g` occurs inside the expansion of some other

function. Notice also that it appears as if `expand/g` gets called twice in each example instead of just once as we might have expected. We can verify that `expand/g` is being called twice by tracing calls to this procedure.

```
> trace( `expand/g` );
```

Now let us re-execute one of the last two examples.

```
> expand( exp(g(x)+z) );
```

The trace did not work because expand got that result out of its remember table. We need to erase expand's remember table again.

```
> readlib(forget)( expand );
```

Now re-execute the example.

```
> expand( exp(g(x)+z) );
```

Now we see that `expand/g` really was called twice. If we also trace the expand and `expand/exp` procedures, then we can get a bit of an idea why `expand/g` was called twice. In order to do this trace, we first need to forget expand's remember table.

```
> readlib(forget)( expand );
```

Now set the procedures that we want to trace.

```
> trace( expand, `expand/exp`, `expand/g` );
```

Re-execute the example (and then untrace the procedures).

```
> expand( exp(g(x)+z) );
> untrace( expand, `expand/exp`, `expand/g` ):
```

By carefully reading the above trace, we see that the first call of `expand/exp`, with operand g(x)+2, calls expand on this operand, and this causes the first call of `expand/g`. Then `expand/exp` actually expands the exponential. Then after the exponential is expanded, `expand/exp` calls `expand/exp` on each term of the expanded exponential. This leads to a second call of `expand/g`. So `expand/g` is called twice because the `expand/exp` procedure expands all of its operands both before and after it expands the exponential. The reason that expand keeps trying to expand operands is that it is hard to know ahead of time in just what order the operands in an expression should be expanded. For example, in the following expression, the inner function needs to be expanded before the outer function can be expanded.

```
> sin(cos(x+y));
> expand( % );
```

But in the next example we cannot expand the inner functions, but after the outer function has been expanded, more expansion still needs to be done on the result.

```
> sin(2*arctan(x+y));
> expand( % );
```

Here is the same expression with a lot of delayed evaluation. This allows us to see the order that the expansion is done in. First, the sin is expanded.

```
> expand( sin(2*''arctan('x+y')'') );
```

Next is some automatic simplification.

```
> %;
```

Get rid of one level of delayed evaluation.

```
> %;
```

Now another expansion is still needed.

```
> expand( % );
```

As we will see in the example below, when writing a procedure that interfaces with **expand**, it is important to make sure that every possible expansion gets done and this is usually accomplished with a number of recursive calls to **expand**.

```
>
```

At this point we seem to need to restart Maple, otherwise the **forget** procedure does not work for **expand**. It seems that using **trace** on the **expand** procedure somehow gets Maple confused.

```
> restart;
```

Now let us turn to the function h and teach **expand** the identity $\mathrm{h}(x+y)=\frac{\mathrm{h}(x)}{\mathrm{h}(y)^2}$. Here is a first attempt at defining **`expand/h`**.

```
> `expand/h` := proc( u )
>   if type(u, `+`) then
>     'h'(op(1,u))/'h'(op(2,u))^2;
>   else
>     'h'(u)
>   fi
> end;
```

Let us try this out.

```
> h(x+y);
> expand( % );
> exp(h(x+y))+h(z)/h(z^2-3);
> expand( % );
```

So far so good. But the next few examples show that we need to do some work on **`expand/h`**.

```
> h(x+y+z);
> expand( % );
> h(exp(x+y)+h(a+b));
> expand( % );
> h(abs(x*y));
> expand( % );
```

Of the last three examples, the first showed that we need to be more careful of how we handle a sum in the definition of **`expand/h`**, and the last two examples show that **expand** is not being allowed to work on the operand of a function call **h(u)**. The second problem is easy to fix. Here is a redefinition of **`expand/h`** that calls **expand** recursively.

```
> `expand/h` := proc( u )
>   if type(u, `+`) then
>     'h'(op(1,expand(u)))/'h'(op(2,expand(u)))^2
>   else
>     'h'(expand(u))
```

```
>   fi
> end;
```
Let us clear the remember table for **expand**.
```
> readlib(forget)( expand );
```
Now let us retry some of the above examples.
```
> h(exp(x+y)+h(a+b));
> expand( % );
> h(abs(x*y));
> expand( % );
```
The first example now works fine because we put calls to **expand** in the then-part of the conditional statement inside the body of **`expand/h`**, and the second example now works correctly because of the call to **expand** in the else-part. But now consider the following two examples.
```
> h(sin(x+y));
> expand( % );
> h(cos(2*x)+1);
> expand( % );
```
Neither of these examples is correct. In the function call **h(sin(x+y))**, **h** has only one operand, so there is no need to apply the identity for **h**. But after **expand** applies a trig identity to the single operand, the operand for **h** becomes a sum, and then the identity for **h** should be applied. So in the case where the operand for **h** is not a sum, we need to check if the operand becomes a sum after **expand** is called to expand the operand. And in the function call **h(cos(2*x)+1)**, the operand is a sum, so **`expand/h`** executes **'h'(op(1,expand(u)))/'h'(op(2,expand(u)))^2** with **u** equal to **cos(2*x)+1**. But here is what **expand(u)** returns.
```
> expand( cos(2*x)+1 );
```
So here we have the case of an operand that is a sum, but it is no longer a sum after it is expanded by **expand**. These two examples show that what we really should be doing in **`expand/h`** is expanding the operand from the call **h(u)** before we do any analysis of its structure and have the conditional statement analyze the structure of the expanded operand. Here is another version of **`expand/h`**.
```
> `expand/h` := proc( u )
>   local temp;
>   temp := expand( u );
>   if type(temp, `+`) then
>     'h'(op(1,temp))/'h'(op(2,temp))^2
>   else
>     'h'(temp)
>   fi
> end;
```
In this version of **`expand/h`**, the local variable **temp** holds the result of expanding the operand from the function call **h(u)**. If **temp** is not a sum, then the else-part just returns **h** composed with

the results from **expand**. And if **temp** is a sum, then we apply the identity. Before we can try this out we need to clear **expand**'s remember table again.

```
> readlib(forget)( expand );
```

Now retry the last two examples.

```
> h(sin(x+y));
> expand( % );
> h(cos(2*x)+1);
> expand( % );
```

If we really want to apply the identity for h to the sum **cos(2*x)+1**, without expanding this sum first, then we can use right-quotes to delay the evaluation of the **cos(2*x)** term.

```
> expand( h(''cos(2*x)''+1) );
> %;
```

Or, better yet, we can use a **cos** option to **expand** to prevent **expand** from expanding the cosine sub-expression.

```
> readlib(forget)( expand );
> expand( h(cos(2*x)+1), cos );
>
```

Now let us turn to the problem of handling sums. The following result is obviously not correct.

```
> expand( h(a+b+c) );
```

Our definition of **`expand/h`** assumes that if the expanded operand from the function call **h(u)** is a sum, then it only has two terms, which need not be the case. We need a way to handle sums of three or more terms. We need to be able to replace this command,

'h'(op(1,temp))/'h'(op(2,temp))^2,

with something like

'h'(op(1,temp))/'h'(*second-through-last-terms-of-***temp)^2**.

Unfortunately, there is no obvious way to get the second through last items from a **`+`** data structure. With a **list** data structure we can use the following index notation.

```
> [a,b,c,d,e][2..-1];
```

But this notation does not work with **`+`** data structures.

```
> (a+b+c+d+e)[2..-1];
```

But here is a trick that will do what we want. The **subsop** command allows us to substitute for items in a data structure, much like **subs**, but **subsop** allows us to specify the index of the operand that we want to substitute for. Here are a few examples.

```
> subsop( 2=Pi, a+b+c+d+e );
> subsop( 5=Pi, a+b+c+d+e );
> subsop( 2=7, x+x^2+x^3 );
```

So here is the trick that we want. If we use **subsop** to substitute 0 for a term of a sum, then that term goes away because of automatic simplification.

```
> subsop( 1=0, a+b+c+d+e );
> subsop( 4=0, a+b+c+d+e );
```

So let us use this trick in **`expand/h`**.

```
> `expand/h` := proc( u )
>   local temp;
>   temp := expand( u );
>   if type(temp, `+`) then
>     'h'(op(1,temp))/'h'(subsop(1=0,temp))^2
>   else
>     'h'(temp)
>   fi
> end;
```

Clear **expand**'s remember table again.

```
> readlib(forget)( expand );
```

Try this version out.

```
> expand( h(a+b+c) );
> expand( % );
> expand( h(a+b+c+d) );
> expand( % );
> expand( % );
```

Here is another example.

```
> h(cos(x+y)+sin(x+y));
> expand( % );
```

Well, things are better now, but still not correct. The identity for h needs to be applied now to the term in the denominator. Here is how we do that. We call **expand** on the term in the denominator, which will just call **`expand/h`** again for us and apply the identity.

```
> `expand/h` := proc( u )
>   local temp;
>   temp := expand( u );
>   if type(temp, `+`) then
>     'h'(op(1,temp))/expand('h'(subsop(1=0,temp))^2)
>   else
>     'h'(temp)
>   fi
> end;
```

Clear **expand**'s remember table again.

```
> readlib(forget)( expand );
```

Try this version out.

```
> expand( h(a+b+c) );
> expand( h(a+b+c+d) );
> h(cos(x+y)+sin(x+y));
> expand( % );
>
```

Exercise: If you carefully study our version of **`expand/h`**, you will see that it is treating

addition as right associative, that is it treats **a+b+c** as **a+(b+c)**. We did this to make the procedure easier to write. But addition is usually considered to be left associative. Here is a way to force left associative expansion of **h(a+b+c)**. Notice that the final result is different from above.

```
> expand( h(''(a+b)''+c) );
> expand( % );
```

Modify the procedure **`expand/h`** so that addition is treated as left associative.

```
>
```

Exercise: Teach **expand** the following expansion rule for the function k.

$$\mathrm{k}(x\,y) = y\,\mathrm{k}(x)$$

Treat multiplication as left associative, so $\mathrm{k}(x\,y\,z) = y\,z\,\mathrm{k}(x)$. Also, be sure you get the result $\mathrm{k}(|x\,y|) = |y|\,\mathrm{k}(|x|)$.

```
>
```

The **simplify** command also has an interface like the **expand** command. If we want to teach **simplify** a special simplification rule for a function f, then we need to define a procedure **`simplify/f`**. But there is a big difference between the interface to **expand** and the interface to **simplify**. Given a function call like **f(u)**, **expand** will make the call **`expand/f`(u)**, but **simplify** makes a call of the form **`simplify/f`(***expression***)** where *expression* is an expression containing the function call **f(u)**. It is not hard to figure out why **simplify** needs to call **`simplify/f`** with expressions containing **f**. Many simplifications involve more than one occurrence of a function or even the occurrence of several different functions. For example, the following simplification requires two occurrences of the function **exp**.

```
> simplify( exp(x)*exp(y) );
```

The following simplification requires the presence of two different functions, **sin** and **cos**.

```
> simplify( cos(2*x)+sin(x)^2 );
```

So the **simplify** command needs to provide a **`simplify/f`** procedure with a lot of information about the context that **f** appears in. Here is a simple **`simplify/f`** procedure that we can experiment with. This procedure prints out the value of its input **context**, so that we can see how **simplify** called **`simplify/f`**, and then it returns **context** unmodified.

```
> `simplify/f` := proc( context )
>   print( `calling simplify/f with parameter` = context );
>   context
> end;
```

Now let us use this procedure to watch some examples of **simplify** calling **`simplify/f'**. In the first two examples, **`simplify/f`** is called twice, each time with a different expression containing **f**. (Notice that we put colons at the ends of these examples to suppress the final output from **simplify**).

```
> simplify( x+f(y*f(x)) ):
> simplify( exp(f(y)^2) ):
```

In the next example, **`simplify/f`** is called twice, and one of the calls occurs after **simplify**

does some simplification.

```
> simplify( exp(x)*exp(f(z)) ):
```

In each of the next two examples, `` `simplify/f` `` is called three times.

```
> exp(a)*exp(f(u))+exp(b+f(u));
> simplify( % ):
> tan(exp(f(w)));
> simplify( % ):
```

If you should go back and re-execute the above examples, all of them will produce no output the second time they are executed. This is because **simplify** will use its remember table the second time it is called with the same input. This is also why we kept changing the variables in each example, to keep **simplify** from using its remember table. If you want to experiment with some of the above examples, you will probably want to clear **simplify**'s remember table just before each execution of **simplify**.

```
> readlib(forget)( simplify );
```

The above examples should convince you that working with the interface to **simplify** will be quite different from working with the interface to **expand**.

```
>
```

Let us try doing a simple example of a `` `simplify/f` `` procedure. Let us teach **simplify** that f is a linear function, that is, let us teach **simplify** to implement the identity $f(x)+f(y)=f(x+y)$. Here is a first attempt at `` `simplify/f` ``.

```
> `simplify/f` := proc( context )
>   local cntxt, new_operand, simplified, i;
>   cntxt := context;
>   new_operand := 0;
>   simplified := false;
>   if type( cntxt, `+` ) then
>     cntxt := convert( cntxt, list ); # it's easier to work with
  a list
>     for i from 1 to nops(cntxt) do   # look for calls to f
>       if type(cntxt[i],function) and evalb(op(0,cntxt[i])='f')
  then
>         new_operand := new_operand + op(1,cntxt[i]);
>         cntxt[i] := 0; # remove the term from the sum
>         simplified := true
>       fi
>     od;
>     cntxt := convert( cntxt, `+` );  # convert back to a sum
>     if simplified then cntxt := 'f'(new_operand) + cntxt fi
>   fi;
>   cntxt  # this is always the return value
> end;
```

This procedure works by first checking if the context that **f** is in is a sum. If the context is not a sum, then the procedure does nothing. If the context is a sum, then the procedure checks each term of the sum to see if it is a function call to **f** (but first the sum is converted into a list, since it is a bit easier to manipulate lists, e.g., we can use index notation with lists but not with sums). If a term of the context is a call to **f**, then the operand of that call is added to a local variable, **new_operand**, that holds a running total of the operands to all the calls to **f** in the context. After the operand of the call is added to the running total, the call to **f** is removed from the sum (by making the term 0) and a flag is set. After all of the terms in the context have been checked, the context (minus any calls to **f**) is converted back into a sum and then the new call to **f** (if there is one), with the new operand, is added on.

```
>
```

Let us try this procedure out.

```
> f(x)+f(y)+f(z);
> simplify( % );
> f(x) + sin(x) + f(3) + exp(x);
> simplify( % );
```

In the next example, notice that **simplify** simplifies the operand of the first call to **f** before it calls **`simplify/f`**.

```
> simplify( f(exp(x)*exp(y)) + f(z) );
```

In the next example, **simplify** needs to do a (symbolic) simplification before **`simplify/f`** can do its simplification.

```
> arctan(tan(f(x)))+f(y);
> simplify( %, symbolic );
```

But here is an example where **`simplify/f`** does not do all of the simplifications that it could do.

```
> f(f(x))+f(y)+f(f(z));
> simplify( % );
```

There is still more simplification that can be done, as the next command shows.

```
> simplify( % );
```

We can solve this problem by recursively calling **simplify** on the new operand for the simplified call to **f**.

```
>
> `simplify/f` := proc( context )
>   local cntxt, new_operand, simplified, i;
>   cntxt := context;
>   new_operand := 0;
>   simplified := false;
>   if type( cntxt, `+` ) then
>     cntxt := convert( cntxt, list ); # it's easier to work with
  a list
>     for i from 1 to nops(cntxt) do   # look for calls to f
```

```
>       if type(cntxt[i],function) and evalb(op(0,cntxt[i])='f')
  then
>         new_operand := new_operand + op(1,cntxt[i]);
>         cntxt[i] := 0; # remove the term from the sum
>         simplified := true
>       fi
>     od;
>     cntxt := convert( cntxt, `+` );  # convert back to a sum
>     if simplified then
>       # first check if the new operand can be simplified
>       new_operand := simplify( new_operand );
>       cntxt := 'f'(new_operand) + cntxt
>     fi;
>   fi;
>   cntxt  # this is always the return value
> end:
```

We need to forget **simplify**'s remember table.

```
> readlib(forget)( simplify );
```

Now try our last example again.

```
> f(f(x))+f(y)+f(f(z));
> simplify( % );
>
```

Let us see if we can add the other part of the definition of linearity to the procedure **`simplify/f`**. We want to teach **simplify** to implement the identity $a\,\mathrm{f}(x) = \mathrm{f}(a\,x)$ when a is a constant, e.g., $3\,\mathrm{f}(x) = \mathrm{f}(3\,x)$. We need to ask ourselves though, just what is a constant in Maple? It turns out that Maple has two data types that represent constant values, **constant** and **complexcons**. An object is of type **complexcons** if it can be evaluated to a complex floating point number by **evalf**. So every number, i.e., **integer**, **fraction**, or **float**, has type **complexcons**. But some other, more symbolic expressions, are also of type **complexcons**.

```
> type( Pi, complexcons );
> type( sin(1), complexcons );
```

But an arbitrary unevaluated function call with a constant parameter need not have type **complexcons**.

```
> type( h(1), complexcons );
```

The type **constant** is a bit more general than **complexcons**. Any object of type **complexcons** is also of type **constant**. In addition, arbitrary unevaluated function calls with constant parameters do have type **constant**.

```
> type( h(1), constant );
```

The type **constant** also includes some symbolic constants that are not very numeric.

```
> type( false, constant );
```

The generality of the type **constant** can make things tricky for us, so we will take the type

`complexcons` as our definition of a constant. After we implement the identity with constants of type `complexcons`, we will look at some of the difficulties that constants of type `constant` can cause.

```
> 
```

Before trying to rewrite `` `simplify/f` ``, let us experiment again and see what kind of contexts `` `simplify/f' `` might be called with. Here is the version of `` `simplify/f` `` that reports the context for us.

```
> `simplify/f` := proc( context )
>   print( `calling simplify/f with parameter` = context );
>   context
> end;
```

Here is a typical expression that we would like to be able to simplify.

```
> simplify( 3*f(x)+f(y) ):
```

Notice that the context is the whole sum. The `simplify` command does not help us out by, say, making `3*f(x)` a context by itself. Here are two more expressions that we would like to be able to simplify.

```
> simplify( 3*x*f(x) ):
> simplify( 3*sin(1)*f(x) ):
```

In these examples the context is a product, and notice that in the second example there are two constants in the product that need to be simplified. So our version of `` `simplify/f` `` will need to look for both product and sum contexts, and in the case of a sum context, `` `simplify/f` `` will need to look for pertinent product sub-contexts. And in a product context, our procedure will need to look for an arbitrary number of constants that need to be moved into the function call.

Since a product context can arise in two different ways, as either the whole context or as a sub-context of a sum context, it is useful to define a helper procedure, `` `simplify/f/scalar` ``, that handles only product contexts, and then have `` `simplify/f` `` call this helper procedure from two places, from where it tests for a product context and from where it test for a product sub-context. Here is the new definition for `` `simplify/f` `` that uses the helper procedure.

```
> 
> `simplify/f` := proc( context )
>   local cntxt, new_operand, simplified, i;
>   cntxt := context;
>   new_operand := 0;
>   simplified := false;
>   if type( cntxt, `*` ) then cntxt := `simplify/f/scalar`(
  cntxt );
>   elif type( cntxt, `+` ) then
>     cntxt := convert( cntxt, list ); # it's easier to work with
  a list
>     for i from 1 to nops(cntxt) do   # look for calls to f
```

```
>       if type(cntxt[i], `*`) then  # found a product
  sub-context
>         cntxt[i] := `simplify/f/scalar`( cntxt[i] );
>       fi;
>       if type(cntxt[i],function) and evalb(op(0,cntxt[i])='f')
  then
>         new_operand := new_operand + op(1,cntxt[i]);
>         cntxt[i] := 0; # remove the term from the sum
>         simplified := true
>       fi
>     od;
>     cntxt := convert( cntxt, `+` );  # convert back to a sum
>     if simplified then
>       # first check if the new operand can be simplified
>       new_operand := simplify( new_operand );
>       cntxt := 'f'(new_operand) + cntxt
>     fi;
>   fi;
>   cntxt  # this is always the return value
> end;
```

And here is the definition of the helper procedure. This procedure assumes that it is called with a product context. After converting the context into a list, the operands of the context are searched to see if one of them is a function call to **f**. If a function call is found, a search is made for constants in the operands of the context. Whenever a constant is found, the constant is multiplied with the local variable **cnstnt** (so **cnstnt** is a running multiple of all the found constants), and then the constant is removed from the context (by setting its operand equal to 1). After all of the constants have been found, the new constant, **cnstnt**, is put inside of the function call. Notice that once a call to **f** is found, the outer for-loop terminates because **cnstnt** will have a non-zero value. Finally, the context is converted back to a product.

```
> 
> `simplify/f/scalar` := proc( context )
>   local cntxt, cnstnt, i, j;
>   cnstnt := 0;
>   cntxt := convert( context, list );
>   # look for a call to f in the operands of the context
>   for i from 1 to nops(cntxt) while evalb( cnstnt=0 ) do
>     if type(cntxt[i], function) and evalb(op(0,cntxt[i])='f')
  then
>       # look for constants in the operands of the context
>       cnstnt := 1; # this will stop the outer loop
>       for j from 1 to nops(cntxt) do
>         if type( cntxt[j], complexcons ) then
>           cnstnt := cnstnt*cntxt[j];
```

```
>           cntxt[j] := 1  # remove the constant from the product
>         fi
>       od;
>       # put the new constant inside the function call
>       cntxt[i] := 'f'( cnstnt*op(1,cntxt[i]) );
>     fi
>   od;
>   convert( cntxt, `*` ); # this is always the return value
> end;
```

Let us try this combination of procedures. First, forget **simplify**'s remember table.

```
> readlib(forget)( simplify );
```

Here are some examples that test these procedures.

```
> simplify( 3*f(x)+f(y) );
> simplify( 3*x*f(x) );
> simplify( 3*sin(1)*f(x)*f(y) );
> simplify( f(2)*3 );
> simplify( h(1)*f(x) );
> simplify( f(2)*3*f(x) );
> simplify( -3*f(2)+f(7) );
> simplify( 5*(f(2*x)+f(y)) );
> simplify( 3*x*f(x)+f(y)+2*f(z) );
>
```

Exercise: Change the type **complexcons** to **constant** in the procedure **`simplify/f/scalar`** and then execute the following **simplify** commands. Try to explain what is going on.

```
> readlib(forget)( simplify );
> simplify( h(1)*f(x) );
> simplify( f(2)*3 );
> simplify( f(2)*f(x) );
> simplify( f(2)*3*f(x) );
> simplify( -3*f(2)+f(7) );
>
```

Exercise: Notice that the following expression is not simplified completely. Modify **`simplify/f/scalar`** and/or **`simplify/f`** to fix this.

```
> simplify( 3*f(f(x)+f(y)) );
```

The expression $3\,f(f(x)+f(y))$ can be simplified to $f(f(3\,x+3\,y))$ in different ways. For example, the steps might be $f(3\,f(x)+3\,f(y))$ then $f(f(3\,x)+f(3\,y))$ then $f(f(3\,x+3\,y))$. Or the steps might be $3\,f(f(x+y))$ then $f(3\,f(x+y))$ then $f(f(3\,x+3\,y))$. Which order does the combination of **simplify**, **`simplify/f`** and **`simplify/f/scalar`** use?

```
>
```

Exercise: Suppose we assume that **a** is a constant.

```
> assume( a, complexcons );
```

It would be nice if the identity $a\ f(x) = f(a\ x)$ worked with this assumption, but so far it does not.

```
> simplify( a*f(x) );
> simplify( 3*a*f(x) );
```

Modify the procedure `` `simplify/f/scalar` `` so that it works with Maple's assume facility. Also try experimenting with the property **constant**, as in

```
> assume( a, constant );
> a := 'a':
>
```

Exercise: Notice that an expression like $3\ f(x)^n$, with n a positive integer, can be transformed to $f(3\ x)\ f(x)^{(n-1)}$ using the linearity of f. It is not even clear that this transformation really is a "simplification", and our current version of `` `simplify/f/scalar` `` does not do this transformation.

```
> simplify( 3*f(x)^2 );
```

Go ahead and implement this transformation in `` `simplify/f/scalar` ``.

```
>
```

Exercise: Suppose that the three functions f, g, and h, satisfy the following identity $f(x) + g(y) = h(x + y)$. Write a procedure `` `simplify/g` `` that will allow **simplify** to implement this identity. Try testing `` `simplify/g` `` both with and without defining the `` `simplify/f` `` procedure that makes f linear. Notice that with `` `simplify/f` `` defined, $f(x) + g(y) + f(z)$ can simplify to any of $h(x + z + y)$ or $h(x + y) + f(z)$ or $f(x) + h(y + z)$. How do we know which result these procedures will produce? What about an expression like $f(x) + g(y) + g(3) + f(z) + f(y)$?

```
>
```

Exercise: Complete the definition of f as a linear function by writing a procedure (or procedures) that interface to the **expand** command and let **expand** implement the identities $f(x + y) = f(x) + f(y)$ and $f(a\ x) = a\ f(x)$, where a is a constant.

```
>
```

Exercise: Write a procedure **MakeLinear** that takes one parameter, a name, and makes that name act like a name for a linear function. **MakeLinear** should work by constructing appropriate procedures that interface with the procedures **simplify** and **expand**. So for example if we make the call **MakeLinear(k)**, then **MakeLinear** will define procedures `` `simplify/k` ``, `` `simplify/k/scalar` ``, and `` `expand/k` ``, so that **simplify** and **expand** can implement the identities $k(x + y) = k(x) + k(y)$ and $k(a\ x) = a\ k(x)$ where a is a constant. Notice that the name **k** should remain unassigned after **MakeLinear(k)** returns. The procedure **MakeLinear** should

return NULL. Write another procedure, **UnMakeLinear**, that also takes one parameter, a name, and unassigns the names of the three appropriate interface procedures. This procedure should also return NULL. (Hint: You will need to look up and use the Maple procedure **parse** with the **statement** option.)

```
>
```

```
>
```

15.7. Differentiating functions

Let us go back to our (primitive) differentiation command and see if we can extend it some. Let us try to teach our command to differentiate the trig functions sine and cosine. We will write a new differentiation command and call it **my_diff**. We would like it to do everything that **diff_poly** does plus we should be able to call it this way **my_diff(sin(x))** or this way **my_diff(cos(x))** and get the correct derivative.

```
>
```

First of all, here is our version of **diff_term** (note the colon at the end of the definition).

```
> diff_term := proc( term )
>   if type(term, numeric) then
>     0
>   elif type(term, name) then
>     1
>   elif type(term, {name,numeric}&*name) then
>     op(1, term)
>   elif type(term, `^`) then
>     op(2, term)*op(1,term)^(op(2,term)-1)
>   else
>
  op(1,term)*op(2,op(2,term))*op(1,op(2,term))^(op(2,op(2,term))-
  1)
>   fi
> end:
```

And here is our version of **diff_poly**.

```
> diff_poly := proc( poly )
>   if type(poly, `+`) then
>     map( diff_term, poly )
>   else
>     diff_term( poly )
>   fi
> end:
```

The procedure **my_diff** will have to be able to tell when its input is of the form **sin(x)** or

cos(x). Maple considers **sin(x)** and **cos(x)** to be data structures of data type **function**.

```
> whattype( sin(x) );
> type( cos(z), function );
```

If **cos(x)** is a data structure, what are the pieces of data in the data structure?

```
> op( cos(x) );
> op( sin(y) );
```

In Maple, anything of the form **name(** *expression sequence* **)** is considered a **function** data structure and the expression sequence of actual parameters to the function are the data in the data structure.

```
> whattype( f(x,y,z) );
> op( f(x,y,z) );
> type( colors(green, red, orange), function);
> op( colors(green, red, orange) );
> whattype( plot(sin, -1..1, color=black) );
> op( 'plot(sin, -1..1, color=black)' );
```

So our procedure **my_diff** can test the data type of its input to see if it is a **function** data structure. But if it does find a **function** data structure, how will it tell if the function is **sin** or **cos**? It turns out that a **function** data structure is a bit different from other data structures. A **function** data structure has a 0'th operand and the 0'th operand of a **function** data structure is the name of the function.

```
> op( f(x,y) );
> op( 0, f(x,y) );
> op( sin(w) );
> op( 0, sin(w) );
```

For most other data structures, the 0'th operand is the data type of the data structure.

```
> op( 1/3 ); op( 0, 1/3 );
> op( [a,b,c] ); op( 0, [a,b,c] );
> op( 1..5 ); op( 0, 1..5 );
> op( g(a,b,c) ); op( 0, g(a,b,c) );
>
```

So the procedure **my_diff** can use the **op** command to find out what function it is trying to differentiate. Here is a first attempt at the procedure **my_diff**.

```
> my_diff := proc( f )
>   if type(f, function) then
>     if op(0, f) = 'cos' then -sin( op(f) )
>     elif op(0, f) = 'sin' then cos( op(f) )
>     else 'D'*f
>     fi
>   else
>     diff_poly( f )
>   fi
```

```
> end;
```
Try this out.
```
> my_diff( cos(x) );
> my_diff( sin(y) );
> my_diff( ln(x) );
> my_diff( g(x) );
> my_diff( 3*x^5 );
> my_diff( sin(x^2) );
```
Well we have not taught our command the chain rule yet so we should not have expected anything else. But how hard would it be to get the chain rule to work? All we need to do is multiply the current result with the derivative of what is inside the function. Here is an attempt at this.
```
> my_diff := proc( f )
>   if type(f, function) then
>     if op(0, f) = 'cos' then -sin( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'sin' then cos( op(f) ) * my_diff( op(f) )
>     else 'D'*f * my_diff( op(f) )
>     fi
>   else
>     diff_poly( f )
>   fi
> end;
```
Let us try this version.
```
> my_diff( sin(x^2) );
> my_diff( cos(3*x^6+5*x-2) );
> my_diff( cos(sin(x^2-5*x)) );
> my_diff( cos(h(x)) );
> my_diff( g(h(x)) );
```
It seems to work fairly well (but the last one is a bit messed up because of an automatic simplification). Notice how we have made use of **recursion**. The procedure `my_diff` does the chain rule by calling itself. A procedure that calls itself is said to be recursive.
```
>
```

Let us teach `my_diff` a bunch of new derivative rules.
```
> my_diff := proc( f )
>   if type(f, function) then
>     if   op(0, f) = 'cos' then
>       -sin( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'sin' then
>       cos( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'tan' then
>       sec( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'sec' then
>       sec( op(f) )*tan( op(f) ) * my_diff( op(f) )
```

```
>     elif op(0, f) = 'cot' then
>       -csc( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'csc' then
>       -csc( op(f) )*cot( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'ln' then
>       (1/op(f)) * my_diff( op(f) )
>     elif op(0, f) = 'exp' then
>       f * my_diff( op(f) )
>     else 'D'*f * my_diff( op(f) )
>     fi
>   else
>     diff_poly( f )
>   fi
> end;
```

Here are several of examples of its use.

```
> my_diff( ln(x^2-2) );
> my_diff( csc(exp(5*x-x^(-2))) );
> my_diff( x^2+cos(x^2) );
```

We should be able to differentiate functions like this last one. But fixing this problem, given the way that we have **my_diff**, **diff_poly**, and **diff_term** organized, would be too difficult. So let us step back a little and reorganize what we have written so far. We want **my_diff** to be able to do the sum rule for differentiation (i.e., the derivative of a sum is the sum of derivatives). Right now, we have that rule done in **diff_poly** for sums of monomials (implemented using the **map** command and **diff_term**). Let us pull the test for the **`+`** data type out of **diff_poly** and put it in **my_diff**. Then **my_diff** can handle sums of functions and monomials, and we will no longer even need **diff_poly** (we just need to change the call to **diff_poly** at the end of **my_diff** into a call to **diff_term**). And we do not need to make any change in **diff_term**. Here is the new version of **my_diff**.

```
> 
> my_diff := proc( f )
>   if type(f, `+`) then  # do the sum rule
>     map( my_diff, f )
>   elif type(f, function) then
>     if   op(0, f) = 'cos' then
>       -sin( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'sin' then
>       cos( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'tan' then
>       sec( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'sec' then
>       sec( op(f) )*tan( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'cot' then
```

```
>       -csc( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'csc' then
>       -csc( op(f) )*cot( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'ln' then
>       (1/op(f)) * my_diff( op(f) )
>     elif op(0, f) = 'exp' then
>       f * my_diff( op(f) )
>     else 'D'*f * my_diff( op(f) )
>     fi
>   else
>     diff_term( f )
>   fi
> end;
```

Let us see how well this works.

```
> my_diff( x^2 + cos(x^2) );
> my_diff( tan(x) + x^2 + 3/x );
> my_diff( ln(x^2+exp(x)) );
> my_diff( cos( x+exp(x^2+sin(x))) );
```

Very nice.

```
>
```

Exercise: Explain in detail how `my_diff` manages to compute the derivative of `ln(x^2+exp(x))` which has the sum rule *inside* of the chain rule.

```
>
```

Now how about the product rule? Can we do that also? The product rule should probably go in `my_diff` where the sum rule is also taken care of.

```
>
> my_diff := proc( f )
>   local rest;            # needed for the product rule
>   if type(f, `+`) then   # do the sum rule
>     map( my_diff, f )
>   elif type(f, `*`) then # do the product rule
>     # remove the first operand from the product; put the rest
  in rest
>     rest := subsop( 1=1, f );
>     my_diff( op(1, f) )*rest + op(1, f)*my_diff( rest )
>   elif type(f, function) then
>     if   op(0, f) = 'cos' then
>       -sin( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'sin' then
>       cos( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'tan' then
```

```
>       sec( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'sec' then
>       sec( op(f) )*tan( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'cot' then
>       -csc( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'csc' then
>       -csc( op(f) )*cot( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'ln' then
>       (1/op(f)) * my_diff( op(f) )
>     elif op(0, f) = 'exp' then
>       f * my_diff( op(f) )
>     else 'D'*f * my_diff( op(f) )
>     fi
>   else
>     diff_term( f )
>   fi
> end;
```

Will it work?

```
> my_diff( x^2*tan(x) );
> my_diff( f(x)*sin(x) );
> my_diff( x^2*exp(3*x)*cos(x) );
> expand( % );
> my_diff( f(x)*g(x) );
```

Not quite right (can you see what happened?). We need a slight change in the definition of **my_diff** in the case for unknown functions.

```
>
> my_diff := proc( f )
>   local rest;            # needed for the product rule
>   if type(f, `+`) then   # do the sum rule
>     map( my_diff, f )
>   elif type(f, `*`) then # do the product rule
>     # remove the first operand from the product; put the rest
  in rest
>     rest := subsop( 1=1, f );
>     my_diff( op(1, f) )*rest + op(1, f)*my_diff( rest )
>   elif type(f, function) then
>     if   op(0, f) = 'cos' then
>       -sin( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'sin' then
>       cos( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'tan' then
>       sec( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'sec' then
```

```
>       sec( op(f) )*tan( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'cot' then
>       -csc( op(f) )^2 * my_diff( op(f) )
>     elif op(0, f) = 'csc' then
>       -csc( op(f) )*cot( op(f) ) * my_diff( op(f) )
>     elif op(0, f) = 'ln' then
>       (1/op(f)) * my_diff( op(f) )
>     elif op(0, f) = 'exp' then
>       f * my_diff( op(f) )
>     else 'D'(f) * my_diff( op(f) ) # slight change here
>     fi
>   else
>     diff_term( f )
>   fi
> end;
```

Now check this version.

```
> my_diff( f(x)*g(x) );
> my_diff( f(x)*g(x)*h(x) );
> expand( % );
```

Much better. Let us try some more examples.

Exercise: For each of the following examples, explain the order of the recursive calls to **my_diff** that do the calculation. For each recursive call, give the parameter for the call and the part of the procedure that handles the call. In each example, exactly how many times are **my_diff** and **diff_term** called?

Here is a sum rule inside of a product rule.

```
> my_diff( (x^2+2*x)*ln(x) );
```

Here is a product rule inside of a sum rule.

```
> my_diff( x+x*exp(x)+sin(x) );
```

Here is a product rule inside of a chain rule.

```
> my_diff( tan(sin(x)*cos(x)) );
```

Here is a chain rule inside of a product rule.

```
> my_diff( exp(sec(x))*ln(x) );
>
```

But all is not well now. Here is a simple example (that used to work).

```
> my_diff( a*x );  # Opps!
```

It turns out that this mistake is related to an earlier bug that we ignored. Recall that our differentiation procedures do not compute the following derivative correctly either.

```
> my_diff( x^2+y^2 );
```

The last result does not make sense because we are not specifying the variable that we are differentiating with respect to. In the next section, we will present a more versatile differentiation procedure that will correct this problem.

```
>
```

Exercise: Explain exactly how `my_diff` arrives at the following, incorrect, result.

```
> my_diff( a*x );
>
```

```
>
```

15.8. Differentiating almost anything (optional)

Here is a differentiation procedure that can differentiate almost anything. In particular, it can do all the basic differentiation rules from freshman calculus. This differentiation procedure, like the one built into Maple, takes two parameters, the expression to be differentiated and the variable to differentiate with respect to.

```
> my_diff := proc(f, x)
>   local i, rest;
>   if not has( f, x ) then 0
>   elif f = x then 1
>   elif type( f, list ) then map( my_diff, f, x )
>   elif type( f, set  ) then map( my_diff, f, x )
>   elif type( f, `=`  ) then map( my_diff, f, x )
>   elif type( f, `+`  ) then map( my_diff, f, x )
>   elif type( f, `*`  ) then
>     rest := mul(op(i,f), i=2..nops(f));
>     my_diff(op(1,f), x)*rest + op(1,f)*my_diff(rest, x);
>   elif type( f, `^` ) then
>     if has(op(1,f), x) and not has(op(2,f), x) then
>       op(2,f)*op(1,f)^(op(2,f)-1)*my_diff(op(1,f), x)
>     elif not has(op(1,f), x) and has(op(2,f), x) then
>       f*ln(op(1,f))*my_diff(op(2,f), x)
>     else
>       f*my_diff( op(2,f)*ln(op(1,f)), x )
>     fi
>   elif type( f, function ) then
>     if op(0,f) = 'sin' then
>       cos(op(f))*my_diff(op(f), x)
>     elif op(0,f) = 'cos' then
>       -sin(op(f))*my_diff(op(f), x)
>     elif op(0,f) = 'tan' then
>       sec(op(f))^2*my_diff(op(f), x)
>     elif op(0,f) = 'sec' then
>       sec(op(f))*tan(op(f))*my_diff(op(f), x)
>     elif op(0,f) = 'csc' then
```

```
>       -csc(op(f))*cot(op(f))*my_diff(op(f), x)
>     elif op(0,f) = 'cot' then
>       -csc(op(f))^2*my_diff(op(f), x)
>     elif op(0,f) = 'ln' then
>       (1/op(f))*my_diff(op(f), x)
>     elif op(0,f) = 'exp' then
>       f*my_diff(op(f), x)
>     else
>       if nops(f) = 1 then
>         'D'(op(0,f))(op(f))*my_diff(op(1,f), x)
>       else
>        add( 'D'[i](op(0,f))(op(f))*my_diff(op(i,f), x),
  i=1..nops(f) )
>       fi
>     fi
>   else
>     RETURN('procname(args)')
>   fi
> end:
```

The rest of this section consists of a series of exercises that ask you to analyze how this procedure works and examples comparing this differentiation procedure with Maple's built in **diff** command.

```
>
```

Exercise: Find where this procedure distinguishes between the following differentiation rules

```
> my_diff( x^n, x );
> my_diff( x^n, n );
> my_diff( n^x, x );
> my_diff( x^x, x );
>
```

Exercise: This differentiation procedure is structured differently from the one in the previous section. Explain why there is no longer a need for **diff_term**. In particular, explain how this procedure differentiates each of the (five) different kinds of monomials. Explain exactly how this procedure arrives at the correct result when differentiating **a*x** with respect to **x**.

```
> my_diff( a*x, x );
>
```

Exercise: Explain exactly how this procedure computes the following result for a derivative that in freshman calculus would be calculated using the quotient rule.

```
> sin(x)/x^2;
> my_diff( %, x );
```

Compare this result with the result from Maple's **diff** procedure.

```
>
```

Exercise: Explain how this procedure arrives at the following results.

```
> my_diff( [t,t^2,t^3], t );
> my_diff( {x^2, x^2+c, ln(5*x), ln(x)}, x );
```

Compare these results with the results from Maple's `diff` procedure.

```
>
```

Exercise: Explain why this procedure returns the following result.

```
> my_diff( vector(3, i->t^i), t );
```

Compare this result with the result from Maple's `diff` procedure.

```
>
```

Exercise: Explain how this procedure computes the following results.

```
> my_diff( f(x), x );
> my_diff( f(x,x), x );
> my_diff( f(x^2,sin(x)), x );
```

Compare the last three results with the results from Maple's `diff` command.

```
>
```

Exercise: Explain exactly how this procedure arrives at the correct derivative for the following expression.

```
> my_diff( x^2+y^2, x );
>
```

Exercise: The notation `5(x)` represents a call to the constant function with value 5. The derivative of this with respect to `x` should be zero.

```
> my_diff( 5(x), x );
> diff( 5(x) , x );
```

Is there anything in the code for `my_diff` that explicitly handles an expression like `5(x)`? How do you think that the above results were arrived at by Maple?

```
>
```

Exercise: Fix the `my_diff` procedure so that it returns an error message when its second operand is not a name. In other words, the following command should return an error message similar to the one returned by Maple's `diff` procedure.

```
> my_diff( 3*x^2, x^2 );
> diff( 3*x^2, x^2 );
>
```

Here are a number of examples that demonstrate minor differences between the `my_diff` procedure and Maple's built in `diff` procedure. For each of these differences, try to determine where in the procedure `my_diff` the difference might come from.

```
> f(g(x));
> my_diff( %, x );
> diff( %%, x );
> f(x)*g(x);
> my_diff( %, x );
> diff( %%, x );
> h(f(x),g(x));
> my_diff( %, x );
> diff( %%, x );
> f(x);
> 'my_diff( %, x )'; # unevaluated call to my_diff
> 'diff( %%, x )';   # unevaluated call to diff
> a*x*exp(x);
> my_diff( %, x );
> diff( %%, x );
> x^n;
> my_diff( %, x);
> diff( %%, x);
> tan( x );
> my_diff( %, x );
> diff( %%, x );
> cot( x );
> my_diff( %, x );
> diff( %%, x );
> exp(f(x));
> my_diff( %, x );
> diff( %%, x );
> vector(3, i->x^i);
> my_diff( %, x );
> diff( %%, x );
>
```

One major difference between Maple's **diff** procedure and our procedure **my_diff** is that **diff** allows the computation of higher order derivatives. If a sequence of names follows the expression to be differentiated, then **diff** will successively differentiate the expression with respect to each name in the sequence. So, for example, the following command computes a second order (ordinary) derivative.

```
> diff( x^3, x,x );
```

And the next three commands compute mixed second order partial derivatives.

```
> diff( x^3*y^3, x,y );
> diff( h(u,v), u,v );
> diff( h(u,v), v,u );
```

Calling **diff** with a sequence of names after the expression amounts to calling **diff** recursively.

That is `diff(f,x,x)` is the same as `diff(diff(f,x),x)`, and `diff(f,x,y)` is the same as `diff(diff(f,x),y)`. Here is how we can implement this feature with our procedure `my_diff`. First, let us rename `my_diff` to `` `my_diff/first` `` to indicate that it only computes first order derivatives.

```
> `my_diff/first` := eval( my_diff ):
```

Now let us redefine `my_diff` as a "front end" to `` `my_diff/first` `` so that `my_diff` can do higher order derivatives.

```
> my_diff := proc(f, x)
>   if nargs > 2 then
>     my_diff( `my_diff/first`(f, x), args[3..-1] )
>   else
>     `my_diff/first`(f, x)
>   fi
> end;
```

Now we can try out this new version of `my_diff`. Compare these results with the above results from `diff`.

```
> my_diff( x^3, x,x );
> my_diff( x^3*y^3, x,y );
> my_diff( h(u,v), u,v );
> my_diff( h(u,v), v,u );
```

Notice that the last result is correct, but it is not as easy to read as the result from `diff`. On the other hand, notice that the next two results are exactly the same for both `my_diff` and `diff`.

```
> my_diff( h(u,u,v), u,v,u );
>    diff( h(u,u,v), u,v,u );
>
```

Exercise: Having two procedures, `my_diff` and `` `my_diff/first` ``, is not really very optimal. We did it that way to make it easier to understand the change that was needed in order to implement higher order derivatives. Combine these two procedures back into a single procedure named `my_diff` that can calculate higher order derivatives.

```
>
```

Another difference between `my_diff` and `diff` is that the `diff` command knows a lot more derivatives.

```
> diff( erf(x), x );
> diff( BesselJ(2,x), x );
> diff( FresnelS(x), x );
> diff( LambertW(x), x );
> diff( x!, x );
> 'diff'( Int(f(x^2),x), x );
> %;
>
```

Exercise: Add one of the above differentiation rules to the `my_diff` procedure.

```
[ >
```

The last exercise shows that to add a new function to `my_diff` we need to edit and then reenter the complete definition of `my_diff`. On the other hand, the Maple procedure `diff` handles functions in a completely different way. For each function `f` that `diff` knows how to differentiate, there is a function `` `diff/f` `` that `diff` calls to actually do the differentiation. For example, here is the function that `diff` uses to compute derivatives involving the function `tan`.

```
[ > interface( verboseproc=2 );
[ > readlib(`diff/tan`):
[ > print( `diff/tan` );
```

Notice that we now know why `diff` computes the derivative of `tan` differently from `my_diff`.

```
[ > diff( tan(x), x );
[ > my_diff( tan(x), x );
```

One advantage of the way that `diff` handles functions is that we can modify the way `diff` does a differentiation by modifying one of `diff`'s helper functions, as the next example demonstrates.

Suppose that we really do not like the way `diff` differentiates the `tan` function. Let us change this behavior by redefining the function `` `diff/tan` ``.

```
[ > `diff/tan` := proc(a,x) (sec(a)^2)*diff(a,x) end;
[ > diff( tan(x), x );
```

Our redefinition of `` `diff/tan` `` did not have any effect because `diff` has a remember table. So we need to clear this remember table.

```
[ > readlib(forget):
[ > forget( diff, reinitialize=false );
[ > diff( tan(x), x );
```

Notice the form of `forget` that we used. The help page for `forget` mentions two important facts. First, when we call `forget(diff)`, Maple automatically calls `forget` for every function whose name begins with `diff/`. So, in particular, calling `forget(diff)` means that Maple also calls `forget(`diff/tan`)`. Second, the help page mentions that calling `forget(`diff/tan`)` has the effect of not just resetting the remember table for `` `diff/tan` `` (which doesn't even have a remember table), it also reinitializes `` `diff/tan` `` back to its original definition. So the call `forget(diff)` wipes out our redefinition of `` `diff/tan` ``. The above way of calling `forget` prevents this wiping out of our redefinition of `` `diff/tan` ``.

```
[ > forget( diff );
[ > diff( tan(x), x );
[ >
```

Exercise: In this exercise you are to analyze the source code for the `` `diff/Int` `` procedure.

```
[ > interface( verboseproc=2 );
[ > readlib(`diff/Int`):
```

```
> print( `diff/Int` );
```
There are two kinds of integrals that we can differentiate, indefinite and definite integrals. Determine the exact part of the code from `diff/Int` that implements the following three deriviatives of indefinite integrals.
```
> 'diff'( Int(f(x), x), x );
> %;
> 'diff'( Int(f(t), t), x );
> %;
> 'diff'( Int(f(x), t), x );
> %;
```
Now notice that in the general form of the derivative of a definite integral, there are three terms in the derivative, one term from the integrand and one term from each of the limits of integration.
```
> Diff( Int(f(t,x), t=a(x)..b(x)), x )
>    = diff( Int(f(t,x), t=a(x)..b(x)), x );
```
Determine the exact steps that `diff/Int` goes through to compute each of the following derivatives. In particular, look for where each of the three terms are calculated (each of the three terms is always calculated, but a term may end up being zero).
```
> 'diff'( Int(f(t), t=a..b), x );
> %;
> 'diff'( Int(f(x), x=a..b), x );
> %;
> 'diff'( Int(f(t), t=a..x), x );
> %;
> 'diff'( Int(f(t), t=x..b), x );
> %;
> 'diff'( Int(f(t,x), t=a..b), x );
> %;
> 'diff'( Int(f(t), t=a(x)..b(x)), x );
> %;
```
Explain in detail the different steps that `diff/Int` uses in the calculation of each of the following derivatives.
```
> 'diff'( Int(f(t,x), t=a(x)..b(x)), x );
> %;
> 'diff'( Int(f(t,x), x=a(x)..b(x)), x );
> %;
>
```

Exercise: Look at the source code for differentiating the absolute value function.
```
> interface( verboseproc=2 );
> readlib(`diff/abs`):
> print( `diff/abs` );
```
Now look at the following result.
```
> diff( abs(x^2), x );
```

It is not clear from the source code for `diff/abs` where the $|x|$ in the output comes from. Do the same calculation but with some delayed evaluation thrown in.

```
> diff( 'abs'(x^2), x );
```

Do the calculation again with a different kind of delayed evaluation.

```
> diff( abs(''x^2''), x );
> %;
```

Use the help page on `abs` and the source code for `diff/abs` to explain exactly how Maple arrived at each of these results. In particular, explain exactly how the chain rule was applied. (Notice how this example shows that delayed evaluation is not always benign. It can sometimes change the form of a result.)

```
>
```

One really big advantage of the way that `diff` handles functions is that it makes the `diff` procedure extensible. This means that we can use this interface to the `diff` command to teach `diff` new differentiation rules. For example, suppose we have defined a pair of functions `fob` and `gob`, and the derivative of `fob` is `gob` and the derivative of `gob` is `fob` squared. We can teach the `diff` command these new differentiation rules by defining two new procedures `diff/fob` and `diff/gob`. Here is the definition of `diff/fob`.

```
> `diff/fob` := proc(u, x)
>   gob(u) * diff(u, x)  # be sure to implement the chain rule
> end;
```

And here is the definition of `diff/gob`.

```
> `diff/gob` := proc(u, x)
>   fob(u)^2 * diff(u, x)
> end;
```

Here are some derivatives using `fob` and `gob`.

```
> diff( fob(x), x );
> diff( gob(x), x );
> diff( fob(x^2), x );
> diff( fob(x), x,x );
> diff( fob(x^2), x,x );
>
```

In general, if we have a function `f` that we want to teach `diff` how to differentiate, we need to define a procedure called `diff/f`. If `f` is a function of one variable, then a call such as `diff(f(u),x)` causes the call `diff/f`(u,x)`. If `f` is a function of two variables, then a call such as `diff(f(u,v),x)` causes the call `diff/f`(u,v,x)`. If `f` is a function of three variables, then a call such as `diff(f(u,v,w),x)` causes the call `diff/f`(u,v,w,x)`. So `diff/f` is a procedure with one more parameter than `f`, and the last parameter for `diff/f` is the variable name that we are to differentiate with respect to. The `diff/f` procedure is responsible for doing all of the differentiation of the expression `f(u)`, so in particular `diff/f` needs to make sure to implement the chain rule and compute `diff(u,x)`.

```
>
```

Let us change the definition of `diff/fob` and drop the chain rule from the definition.

```
> `diff/fob` := proc(u, x)
>   gob(u)
> end;
```

Let us try out this (incorrect) version of `diff/fob`.

```
> diff( fob(x^2), x );
```

The chain rule is still there. The reason is because diff has a remember table, and diff is pulling that old result from its remember table. Let us clear diff's remember table and then try again.

```
> readlib(forget):
> forget( diff );
> diff( fob(x^2), x );
```

And now we see that if `diff/fob` does not implement the chain rule, then the chain rule does not get done.

```
>
```

Exercise: A function hob has the property that its derivative is the negative of the square root of itself times the ln function. That is

$$\frac{\partial}{\partial x}\,\mathrm{hob}(x) = -\sqrt{\mathrm{hob}(x)\,\ln(x)}\,.$$

Write a procedure `diff/hob` that implements this differentiation rule.

```
>
```

Exercise: Suppose that we have three functions, fg of two variables, f of one variable, and g also of one variable, such that the partial derivatives of fg are given by the following formulas.

$$\frac{\partial}{\partial x}\,\mathrm{fg}(x,y) = \mathrm{f}(x)\,\mathrm{g}(y)$$

$$\frac{\partial}{\partial y}\,\mathrm{fg}(x,y) = \mathrm{f}(y)\,\mathrm{g}(x)$$

Write a procedure `diff/fg` that implements these differentiation rules. Find concrete examples of three functions fg, f, and g that have these properties.

```
>
```

```
>
```

15.9. Online help for Maple programming

There is not a lot of online information about Maple programming. Here are some help pages related to the above examples.

The following help page gives an example of an extension `diff/f` for diff.

```
> ?diff
```
The following help page mentions how extensions `expand/f` for **expand** are called, but it does not give any examples.
```
> ?expand
```
The following help page mentions a direct way of calling an extension `simplify/f` for **simplify**, but it gives no information about the automatic calling of `simplify/f` that we have been making use of, and it does not give any examples.
```
> ?simplify
```

In the examples we gave above of extending Maple commands, we informed Maple about the properties of some functions. But in those examples the functions were unassigned names, so there was no sense of evaluating the functions. The following commands give some information on how to extend Maple's evaluators so that we can tell Maple how we want a certain function to be evaluated.
```
> ?eval
> ?evalf
> ?evalapply
```

It is possible to define new data types by extending Maple's **type** command. This is briefly mentioned in the following help page.
```
> ?type
```

The next example worksheet works out a fairly detailed example of a collection of procedures that extend Maple.
```
> ?examples,binarytree
```

We used the example of defining procedures that differentiate and integrate symbolically. The following two example worksheets describe a very different way to implement symbolic differentiation and integration.
```
> ?examples,define
> ?examples,patmatch
```

There is a lot more to Maple programming than just the use of the Maple language and Maple's data structures. Another key ingredient to understanding how Maple works is to know the algorithms that Maple uses. Here is a help page that lists several references for the algorithms that Maple uses for polynomial manipulations.
```
> ?polyrefs
```
And here are some references for integration algorithms.
```
> ?intrefs
```

```
>
```