Maple for Math Majors

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

11. Maple's Evaluation Rules

11.1. Introduction

This worksheet begins our discussions of some of the details of Maple's inner workings. Here we look at Maple's evaluation rules for variables, expressions and functions.

```
> 
```

11.2. Full evaluation

When Maple is presented with an expression of some kind, Maple has to decide what the expression means or, to put it another way, what it's value is. For example, if **p** represents the expression **x^2+2*x+1** and **x** represented the number **3**, it is not obvious how Maple should evaluate the name **p**. If asked the meaning of **p**, Maple could reasonably returned any of **x^2+2*x+1**, or **3^2+2*3+1**, or **16**. How Maple decides what to return is called an **evaluation rule**. Maple has several different evaluation rules that it uses in different situations. In this section we will examine the most common and important of Maple's evaluation rules.

Normally Maple uses an evaluation rule called **full evaluation**. This means that when Maple comes across a variable name, it looks up the name's value, if any. If the value of the variable name contains another variable, its value is also looked up. Maple continues this until it comes up with only numbers and/or unassigned variable names.

Here is an example. First, make **x** unassigned.

```
> x := 'x';
```

Make **y** an expression in **x**.

```
> y := 1+x;
```

Give **x** a numerical value.

```
>  x := 5;
```

Now let us see how Maple evaluates the expression **1+x^2+y**.

```
> 1+x^2+y;
```

How did Maple arrive at this value? The first **x** in the expression represents 5 so **1^x^2+y** becomes **1+5^2+y**. The **y** represents **1+x**, so **1+5^2+y** becomes **1+5^2+1+x**, but the **x** can again be evaluated to 5 so finally Maple has 1+5^2+1+5=32 as the value for **1+x^2+y**.

```
> 
```

Here is another example. First, make both x and z unassigned.

```
> x := 'x';
> z := 'z';
```

Now give y a value in terms of x and z and then give x a numeric value.

```
> y := 1+x+z;
> x := 5;
```

Now let us see what the expression 1+x^2+y evaluates to.

```
> 1+x^2+y;
```

Since z does not have a value, Maple cannot evaluate the expression any further.

```
>
```

One more simple example. First, unassign x, y, and z.

```
> x:='x': y:='y': z:='z':
```

Now make x a name for y, y a name for z, and give z a numeric value, in that order.

```
> x := y;
> y := z;
> z := 3;
```

What does x evaluate to now?

```
> x;
```

Change the value of z.

```
> z := 5;
```

Check the value of x.

```
> x;
```

In this example, x points to y, y points to z, and at first z pointed to 3, so x (and y) evaluated to 3. Then z was pointed at 5, so x (and y) evaluated to 5. This is full evaluation. Maple keeps following the trail of assignments until it gets to a "dead end", either a numeric value or an unassigned name.

```
>
```

So far we have looked at pretty easy examples of full evaluation. The results seem pretty obvious. Now we will look at some examples that are less obvious.

Let us modify our last example a little bit. First unassign x, y, and z.

```
> x:='x'; y:='y'; z:='z';
```

We will reverse the order of the first three assignments from the last example.

```
> z := 3;
> y := z;
> x := y;
```

What does x evaluate to?

```
> x;
```

Now change the value of z.

```
> z := 5;
```

Check the value of x again.

```
> x;
```
In this example, first **z** was assigned the value 3. Then, when **y** was assigned **z**, **z** was evaluated first to its value 3 and so **y** was assigned 3, *not* **z** (this is the full evaluation rule). Similarly, **x** was assigned 3, *not* **y** (why?). So in this example, when the value of **z** is changed, the values of **x** and **y** do not change.
```
>
```

From these last two examples we see, for one thing, that the meaning of three simple commands like **x:=y**, **y:=z** and **z:=3** has a lot to do with the order in which they are executed. Or, to put it another way, the meaning of a Maple command often cannot be determined by just looking at the command. We might need to look at all the commands that were executed before it to determine just what the command means.

Here is another example of full evaluation. Suppose we define **f** as a function.
```
> f := x -> x^2-1;
```
Now let **z** have a value.
```
> z := 3;
```
Now when we enter **f(z)** Maple needs to evaluate both the **f** and the **z**. First Maple evaluates the **f** to a function (the function that squares its input and then subtracts 1), then it evaluates **z** to the number **3**, then it applies the function to the number and calculates the result.
```
> f(z);
```
In general (but not always), Maple uses full evaluation for all the inputs to both functions and Maple commands.
```
>
```

Maple's use of full evaluation can explain some of Maple's more mysterious error messages.
```
> solve( x^2=4, x );
```
Maple first evaluates all of the inputs to the **solve** command so **x^2=4** evaluates to **9=4** (which is not a problem, it is just an equation that does not have any solutions) and the last **x** evaluates to **3** and this is what the **solve** command complains about (the **solve** command can only solve for variables, not for numbers).
```
>
```

Here is another common error message.
```
> plot( x^2, x=0..4 );
```
Maple evaluates all of the inputs to the **plot** command first, so it evaluates the expression **x^2** to be **9** (which is not a problem) and it evaluates the range **x=0..4** to be **3=0..4**, which makes no sense and leads to the error message.
```
>
```

Maple does not always use full evaluation. The next command is not much different from the previous example, but this command does not lead to an error message.

```
> seq( x^2, x=0..4 );
```
Notice that **x** still has a value.
```
> x;
```
So Maple did not use full evaluation for the inputs to the **seq** command. There is no easy way to find out which Maple commands use full evaluation of their inputs and which ones do not. So we can only reiterate a rule of thumb that we stated much earlier in these worksheets. If you get a strange error message, first check that the variable names that you are using as unknowns really are unassigned variables and do not have values.
```
>
```

There are several more evaluation rules in Maple and, not surprisingly, several exceptions to these rules. In the rest of the sections of this worksheet we will go over the most important evaluations rules and some of their important exceptions. In other worksheets we will occasionally have need to look at still other evaluation rules or some other exceptions to these rules.
```
>
```

```
>
```

11.3 Levels of evaluation

Let us go back to our first example of full evaluation.
```
> x:='x':  y:='y': z:='z':
> x:=y;
> y:=z;
> z:=3;
```
Now if we ask Maple to evaluate **x**, Maple uses full evaluation and returns 3.
```
> x;
```
But for Maple to get from **x** to 3 it had to go through the intermediate values of **y** and **z**. There is a name for these intermediate values and there is a way to access them also. The name for these intermediate values is **levels of evaluation** and we can control the level of evaluation using a special form of the **eval** command. For example, here is how we get one level of evaluation of **x**, which returns the name **y**.
```
> eval( x, 1 );
```
Here is how we get two levels of evaluation of **x**, which returns the name **z**.
```
> eval( x, 2 );
```
Here is how we get three levels of evaluation of **x**, which returns the value 3.
```
> eval( x, 3 );
```
We can ask for four or higher levels of evaluation, but 3 always evaluates to 3.
```
> eval( x, 4 )
>
```

Here is another example using levels of evaluation.
```
> f := a*x+b*y;
```

```
> x := y;
> y := z;
> z := 2;
> b := c;
> c := a;
> a := d;
```

Now evaluate **f** to several different levels. Mentally check each level of evaluation yourself to make sure that you understand what the **eval** command is doing.

```
> eval( f, 1 );
> eval( f, 2 );
> eval( f, 3 );
> eval( f, 4 );
>
```

Exercise: Redo the last example. Can you explain why the results of the example are different the second time that you go through it? How can you get the example to work again like it did the first time?

```
>
```

Levels of evaluation do not work when functions are involved. Consider this example.

```
> g := u*sin(u+v);
> u := w;
> v := z;
> w := z;
> z := Pi/4;
```

Now try using levels of evaluation.

```
> eval( g, 1 );
> eval( g, 2 );
> eval( g, 3 );
> eval( g, 4 );
> eval( g, 5 );
```

Notice that nothing is being evaluated inside the **sin** function. However, if we use the **eval** command without any level parameter, then everything is fully evaluated.

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

```
>
```

11.4. Delayed evaluation

Let us restart Maple so that we do not get confused by any previously assigned names.

```
> restart;
```

Sometimes it is necessary to prevent Maple from evaluating a name or an expression that, because of the rule of full evaluation, would otherwise be evaluated. In this section we will see that the right-quote characters are used in Maple to prevent, or delay, evaluation. Here is a simple example of preventing the evaluation of a name. Let us give **x** a value.

```
> x := 3;
```

In the next command Maple will evaluate **x**, because of the rule of full evaluation, and so the value of **y** becomes 6.

```
> y := x+3;
```

But suppose we had really wanted the value of **y** to be literally **x+3**. We can do this in the next command by putting a pair of right-quotes around **x** which prevent Maple from evaluating the **x**.

```
> y := 'x'+3;
```

Notice from the command's output that **y** is now a name for **x+3** because **x** was not evaluated in the last command.

```
>
```

But the right quotes only prevented Maple from evaluating **x** in that one command. Look carefully at the last command's output. There are no right-quotes there. And so the next command will fully evaluate **y** to get 6, since the evaluation of **x** was only prevented in the command that defined **y**.

```
> y;
```

Using one level of evaluation we can verify that **y** is a name for **x+3**.

```
> eval( y, 1 );
```

Another way to put this is that the definition of **y** is *not* an unevaluated **x** plus 3, the definition of **y** is simply **x** plus 3. For this reason, the right-quotes used in the definition of **y** are more properly referred to as **delayed evaluation** of **x**, and not as preventing the evaluation of **x**. Here is another simple example. The next command is an equation and the first **x** has its evaluation delayed and the second **x** is evaluated right away.

```
> 'x' = x;
```

If we re-evaluate the last output, the **x** in the output will get evaluated.

```
> %;
>
```

Now for a more subtle example. In the next command we have a doubly delayed evaluation of the first **x** and a singly delayed evaluation of the second **x**.

```
> ''x'' = 'x';
```

If we re-evaluate the last output, the second **x** is evaluated but the first **x** still has its evaluation delayed.

```
> %;
```

And if we re-evaluate this last output, then the remaining **x** is finally evaluated.

```
> %;
>
```

Here is another example that uses several levels of delayed evaluation.

```
> y := 3;
> x := '''y''';
```
Notice that the value of `x` is `''y''` and not `'''y'''`. Maple used up one level of evaluation when it evaluated the right hand side of the assignment operator.
```
> eval( x, 1 );
> eval( x, 2 );
> eval( x, 3 );
> eval( x, 4 );
>
```

The last few commands lead us to make the following rule for evaluating expressions containing right-quotes. *Whenever Maple comes across an expression with right-quotes around it, Maple just removes one pair of right-quotes and it does not evaluate whatever was inside the quotes*. This has the effect of "delaying" until later the evaluation of the whatever the quotes surrounded.
```
>
```

Here is an interesting question. What should Maple return if now we ask it to evaluate `x`?
```
> x;
```
Notice that this is the same answer that we got above when we used two levels of evaluation. What if we try to force full evaluation of `x` by using `eval(x)`?
```
> eval( x );
```
This is the same answer that we got above when we used three levels of evaluation, but it is not full evaluation, which above took four levels of evaluation. Let us analyze carefully the steps that lead to these results for evaluating the expressions `x` and `eval(x)`. When Maple is asked to evaluate `x`, it first gets `''y''`. The above rule for evaluating right-quoted expressions tells Maple to remove one pair of quotes and then quit evaluating, and so the final result is `'y'`. When Maple is asked to evaluate the expression `eval(x)`, first Maple needs to evaluate the argument to the `eval` command (see Section 11.7 Evaluating function calls, below), so Maple evaluates `x` to `'y'`. After Maple has the value of the argument to `eval`, Maple calls `eval` with this argument, so Maple calls `eval('y')`. But `eval` applies the above evaluation rule for right-quoted expressions and so it just removes one pair of quotes from `'y'` to get `y` and that is the final result.
```
>
```

A common misunderstanding with delayed evaluation is that it can prevent automatic simplifications. In the next example, the right quotes do not prevent the automatic simplification of `u+u`.
```
> 'u + u';
```
Another way to think about this last example is that it shows that automatic simplification is not the same thing as evaluation; `u+u` does not evaluate to `2*u`, it simplifies to `2*u`. Here is a slightly different version of this example.
```
> u := 5;
> 'u + u';
```

The right quotes prevented the evaluation of **u** but not the simplification of **u+u**. In the next command, **u+u** is both simplified and evaluated, and we know now that the simplification is done before the evaluation.

```
> u + u;
>
```

Exercise: The **rand** function, which is built into Maple, is a function that takes no arguments and returns a random integer. Explain why the following command does not return a random fraction.

```
> rand()/rand();
>
```

Here is one last example of how right-quotes might be used. The following is a very "safe" way to define **y** to be the general quadratic equation.

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

In this definition of **y**, we did not have to worry about any preassigned values of **a**, **b**, **c**, and **x**, and we did not need to unassign those variables either, which may, for some reason, be an undesirable thing to do. There are times when it is necessary to be this safe about a definition.

```
>
```

Exercise: Is there a simpler, yet just as "safe", way to define the same **y** as from the last example?

```
>
```

Exercise: You should be aware by now that the following two commands will produce an error message and you should be able to explain why.

```
> x := 5;
> plot(x^2, x=-5..5);
```

Explain the results of the following three commands.

```
> plot(x^2, 'x'=-5..5);
> plot('x'^2, 'x'=-5..5);
> plot('x'^2, x=-5..5);
>
```

One level of delayed evaluation is an often used trick in Maple. We will see several uses for it in this and later worksheets. Even two or three levels of delayed evaluation is needed in some odd circumstances, as we will see.

```
>
```

```
>
```

11.5. A no evaluation rule

We have made the claim that Maple uses full evaluation most of the time. You may have noticed already one place where Maple does not use full evaluation. Suppose we let **x** represent 5.

```
> x := 5;
```
Now let **x** represent 6.
```
> x := 6;
```
Maple did not use full evaluation here. If it had, Maple would have tried to assign the value 6 to 5. Maple did not evaluate the **x** on the left hand side of the assignment operator.
```
>
```

So we can state another of Maple's evaluation rules. The name on the left hand side of an assignment operator is not evaluated. Here is another example. Let us unassign **y** and then let **x** be a name for **y**.
```
> y := 'y';
> x := y;
```
Now consider the following assignment.
```
> x := 5;
```
If Maple had evaluated the left hand side of the assignment operator, it would have assigned 5 to **y**. But it did not. The variable **y** is still unassigned.
```
> y;
>
```

Consider this example.
```
> x:='x': y:='y': z:='z':
> x := y;
> y := z;
> z := -1;
```
What should the next command do?
```
> x := x;
```
Maple used full evaluation on the right hand side of the assignment operator and no evaluation on the left hand side. Compare the last Maple command with the next one.
```
> x := 'x';
```
Now we can explain how "unassigning a variable" works. In the Maple command **x:='x'** the first **x** is not evaluated, since it is on the left hand side of the assignment operator. The second **x** is not evaluated either since the right-quotes delay evaluation. So **x** gets assigned to it an unevaluated **x**, i.e., **x** is given itself as its value.
```
>
```

In an earlier worksheet we pointed out that the **assign** command can be used to make the equal sign in Maple act like an assignment operator. So for example the following commands makes 2 the value of **x**.
```
> assign( x = 2 );
> x;
```
Here is an example of a typical use of **assign**.
```
> x := 'x':
> solve( 3*x+17*a=1, {x} );
```

```
> assign( % );
> x;
```
But it turns out that the **assign** command does not make an equals sign act exactly like an assignment operator. There is one important difference. The **assign** command uses full evaluation on *both* sides of the equal sign. Consider the following example.
```
> x := 'x';
> c := 'c';
> x := c;
> assign( x = 23 );
```
Now let us investigate what the **assign** command did. First check the value of **x**.
```
> x;
```
It appears that **assign** did what we wanted. But if **assign** used full evaluation on both sides of the equals sign, then the **x** in **x=23** should have evaluated to **c** and so the **assign** command should have assigned 23 to **c** and *not* **x**. Let us check the value of **c**.
```
> c;
```
Let us evaluate **x** to only one level.
```
> eval( x, 1 );
```
Now we see that the **assign** command did not assign anything to **x**. It did assign 23 to **c**. (So why does **x** evaluate to 23?) So the **assign** command was not equivalent to the assignment statement **x:=23**.
```
>
```

Here is one way to think about this last example. The assignment operator **:=** was purposely chosen to be a non symmetric symbol to remind you that **x:=y** is not the same as **y:=x** (and **y=:x** is syntactically incorrect) and also to remind you that the assignment operator does not use the same evaluation rules on its left and right hand sides. On the other hand, the equals sign **=** is a symmetric symbol and, when interpreted as an equation, **x=y** is the same as **y=x**. Now **assign(x=y)** is not the same as **assign(y=x)**, but the **assign** command does use the same evaluation rule on both sides of the equals sign (i.e., full evaluation) so in that sense the **=** in an **assign** command is more symmetric than the **:=** in an assignment statement.
```
>
```

Let us look at one last example of evaluation and the assignment operator. Give **x** the value 5.
```
> x := 5;
```
What should the next statement mean?
```
> x := x+2;
```
If Maple had evaluated all of the **x**'s, it would have ended up trying to assign 7 to 5. But Maple only evaluated the **x** on the right of the assignment operator, so it assigned 7 to **x**.
```
> x;
>
```

Let us stop for a moment and compare the two Maple statements **x:=5** and **x:=x+2** with standard

mathematical notation. The Maple command **x:=5** translated into standard mathematical notation would become $x = 5$. What about **x:=x+2**? Should we translate it into $x = x + 2$? But $x = x + 2$ would be interpreted in standard mathematical notation as an equation that simplifies to $0 = 2$ which is not at all what we mean. Notice how the dual nature of the equals sign in standard mathematics is popping up again. The equals sign in $x = 5$ is naturally taken as an assignment, but the equals sign in $x = x + 2$ is naturally taken as part of an equation. So how should we express **x:=x+2** in mathematical notation? In fact, there is no standard mathematical notation that would mean let x have the value that is 2 more than what x currently has (notice that just saying $x = 7$ is not really the same thing). Without an explicit and unambiguous assignment statement, it is hard to express this idea.

```
> 
```

But Maple can have its own problems with **x:=x+2**. Even to Maple there is something a bit strange about this command. For example, suppose we unassign **x**.

```
> x := 'x';
```

Now tell Maple that **x:=x+2**. What should this mean? Stop and think about it before executing the command.

```
> x := x+2;
```

Maple gave us a warning. Here is an explanation of what is wrong. When Maple executes the command **x:=x+2**, Maple needs to evaluate the right hand side (but not the left hand side). When Maple evaluates **x+2**, it sees that, at this point, **x** does not have a value, so it stops evaluating **x+2** and it assigns **x+2** to **x**. Now suppose that in a separate command we ask Maple to evaluate **x**. This is where there is a problem. When Maple evaluates **x** is sees that the value of **x** is **x+2** so it replaces **x** with **x+2**. But then, because of the rule of full evaluation, Maple needs to evaluate **x** again. When Maple evaluates **x** again it gets **x+2** for **x**, so it plugs this into **x+2** to get **(x+2)+2**. But now full evaluation requires that **x** be evaluated once again. A little bit of thought shows that this will go on for ever. So the reason for Maple's warning from the last command is that the following command can cause real problems.

WARNING: The following command will crash some versions of Maple.

```
> x;
```

Well, if you are still here, what happened is that, in the wink of an eye, Maple evaluated **x** and got **x+2** thousands (maybe millions) of times until Maple ran out of memory and did not have any more room left to keep on evaluating **x**. When Maple ran out of room it (hopefully) just stopped, printed an error message, and presented another prompt. Let us use levels of evaluation to check on this explanation.

```
> eval(x, 1);
> eval(x, 2);
> eval(x, 3);
```

Each level of evaluation allows one more level of depth of **x** referring to itself. Let us try quite a few levels.

```
> eval(x, 1000);
```

You can try higher values of evaluation, but when I tried it, Maple crashed (and I lost quite a bit of my work!).

```
> 
```

At this point, we cannot use the variable `x` in any expressions (unless we control the level of evaluation). To get rid of this problem we just unassign `x`.

```
> x := 'x';
> x;
> 
```

So if `x` has a value, then `x:=x+2` is OK. But if `x` does not have a value, then `x:=x+2` is a recursive name definition, which is not good. There are actually many ways to get a recursive name definition. Here is another.

```
> y := x;
> x := y + 2;
```

Notice that it was not obvious from the command `x:=y+2` by itself that this was a recursive name definition. If you should ever get the recursive name definition warning, just unassign the name you were using and then choose a different name.

```
> x := 'x';
> 
```

Exercise: Consider the following assignment.

```
> x := [apple, pear, x[3]];
```

Let us ignore this warning and execute the following commands.

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

Is the above warning a false alarm?

```
> 
```

We will return to the idea of "recursion" when we get to the worksheets on Maple programing. We will see that the idea of something referring to itself (i.e., recursion) is an important part of how Maple works, and it is important to computer science in general.

```
> 
```

```
> 
```

11.6. Last name evaluation

Maple uses the full evaluation rule most of the time, but as we have already seen, there are exceptions to this rule. Three exceptions that we have seen so far are delayed evaluation (i.e. using right-quotes), forcing certain levels of evaluation with the `eval` command, and not evaluating names on the left hand side of an assignment operator. Another exception to the full evaluation rule is the **last name evaluation rule** that is used for the names of tables, arrays, procedures, and Maple

functions. We have not discussed tables, arrays, and procedures yet, so for now we will only use Maple functions as examples for the last name evaluation rule. Here are some examples that demonstrate this rule. Let us define a Maple function named **f**.

```
> f := x -> x^2-1;
```

If we ask Maple to evaluate the name **f**, it just returns **f**.

```
> f;
```

To find out the definition of **f** we have to use the **eval** command to force full evaluation of the name **f**.

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

Now let **h** be a name for **g** and let **g** be a name for **f**.

```
> h := g;
> g := f;
```

Now ask Maple to evaluate the names **g** and **h**.

```
> g;
> h;
```

Notice that the evaluation of both **g** and **h** stopped with the name **f**. On the other hand, if we force full evaluation of either **g** or **h** then we get the definition of **f**.

```
> eval( g );
> eval( h );
```

Now we can state the last name evaluation rule. If the full evaluation of a name would return the definition of a Maple function (i.e., an expression that uses the arrow operator), then Maple will stop the evaluation at the last name it reaches just before the function definition and Maple will return that name. Hence the term "last name evaluation" for the evaluation rule of names that point to a function definition.

```
>
```

But Maple does not always use last name evaluation for names that evaluate fully to a function definition. Here is a counter example.

```
> f:='f': g:='g': h:='h': w:='w':
> f := g;
> g := h;
> h := k;
> k := x -> x^2;
```

If we now ask Maple to evaluate just the name **f**, then it will use last name evaluation.

```
> f;
```

But if we ask Maple to evaluate the functional notation **f(w)**, then Maple will not use last name evaluation.

```
> f(w);
```

If Maple had used last name evaluation in this last command, then the result would have been k(w). Maple instead used full evaluation on the name **f** which returned the definition of **k** which Maple then applied to the input of the function, **w**, and returned w^2.

```
>
```

Maple uses last name evaluation when it evaluates a name of a function in isolation. But if the function name is part of a functional notation, then Maple uses full evaluation of the function name so that it can get to the definition of the function (if there is one). We will say more about this in the next section.

```
> 
```

We end this section with an example where the **eval** command uses last name evaluation when it is supposed to be using full evaluation. Make the following two assignments.

```
> f := 'g'+4; g := x -> x^2;
```

Now notice that the **eval** command, which is supposed to do full evaluation, uses last name evaluation.

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

In this example, last name evaluation is the same as one level of evaluation.

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

We can get **eval** to do full evaluation by forcing two levels of evaluation.

```
> eval( f, 2 );
> 

> 
```

11.7. Evaluating function calls

If **f** is the name of a Maple function (so **f** was defined using the arrow operator), then an expression of the form

f(any-maple-expression**)**

is called a function call. Here is an example of a function and several function calls.

```
> f := x -> x^2-1;
> f(2);
> f(x);
> f(z);
> f(w+u);
> f( (1+cos(Pi/4))/12 );
> f( f(x) );
```

In this section we will see how Maple evaluates a function call. To get a sense of what it is we want to know, consider the following example of a function call.

```
> z := 4;
> f(z);
```

What did Maple do exactly? Did it plug **z** into **f** and get **z^2-1** and then evaluate the **z** to get 4 so it had 4^2-1=15. Or did Maple evaluate **z** first to get 4, then plug 4 into **f** to get 4^2-1=15? As we will see shortly, the difference between these two orders of evaluation is important and we should know which one Maple uses. (In short, Maple uses the later of the two orders.)

```
> 
```

There are two parts to a function call, `f(w)`. There is the name of the function, in this case `f`, and there are the operands of the function, in this case `w`. There are three steps to evaluating a function call. First Maples evaluates the operand (or operands if it is a multivariate function). Then Maple evaluates the name of the function to get the function's definition. Then Maple plugs the results from evaluating the operands into the function's definition and computes the value of the function. (Note: When Maple evaluates the operands it can use either full evaluation or last name evaluation, depending on the operands.)

```
> 
```

Here are some more examples.

```
> f := g;
> g := x -> x^3;
> z := 2;
> f(z);
```

In the last function call, first the `z` was evaluated to 2, then the `f` was evaluated to `g` and `g` was evaluated to the cubing function, then the cubing function was applied to 2 to get 8.

```
> 
```

Now let us wipe out the definition of `g` and try the function call `f(z)` again.

```
> g := 'g';
> f(z);
```

Once again `z` was evaluated to 2 and `f` was evaluated to `g`, but `g` does not evaluate to anything now. So the result for `f(z)` was `g(2)`. We call `g(2)` an **unevaluated function call**. We get unevaluated function calls whenever an unassigned variable is used as a function name in a function call. One of the most common ways to get un unevaluated function call as a result is to misspell the name of a Maple command.

```
> factir( x^2+2*x+1 );
> 
```

Here is a quirk in Maple's function call evaluation. If we call a function with more operands than its definition specifies, then Maple just ignores the extra operands. (In a later worksheet we will see why Maple does this.)

```
> f := (x,y) -> x+y;
> f(2,3,4);
```

But if we call a function with fewer operands than its definition specifies, then we get an error message.

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

Here is an example where we can really see that Maple evaluates a function's operands before evaluating the function. Here is a multivariate function `f`.

```
> f := (x,y) -> x-y;
```
Now we shall define a function that is a bit unusual. The **rand** function, which is built into Maple, is a function that takes no arguments and returns a random integer. Let us define a function **g** that makes use of **rand** like this.
```
> g := x -> rand();
```
So **g** is a function that takes in one number and outputs a randomly chosen integer (that does not depend in any way on the input number!). Now consider the function call **f(g(0),g(0))**. If Maple took the arguments and plugged them directly into the definition of **f**, Maple would get **g(0)-g(0)**, which is zero (see the next command), so the value of the function call would be zero.
```
> g(0)-g(0);
```
But if Maple evaluates the operands to **f** first, then Maple will evaluate **f** with two randomly chosen integers, so the value of the function call will be non zero (with probability *very* close to 1).
```
> f(g(0),g(0));
```
Every time you execute this function call you will get a different value. (Try it.) This example shows that the order that Maple uses for evaluating a function call is important since, in some circumstances, the order of evaluation can determine the value of the function call.
```
>
```

You may at this point have a question about why **g(0)-g(0)** should be zero. Algebraically that seems reasonable. But if Maple used full evaluation and evaluated both of the **g(0)** terms before doing the subtraction, then **g(0)-g(0)** would not be zero. What Maple did with **g(0)-g(0)** was apply one of its **automatic simplification rules**. Whenever Maple has the same expression on either side of a minus sign, it automatically simplifies this to zero (without evaluating the expressions). Maple has many other automatic simplification rules. These are not the same as evaluation rules but as we have just seen, an automatic simplification rule can influence how Maple finds the value of an expression. Here is another example.
```
> h := x -> x^2;
> h(g(0)/g(0));
```
If it were not for an automatic simplification rule (anything divided by itself is 1), Maple would have evaluated this last expression differently (the automatic simplification rule in this case was applied before Maple evaluated the operand in the function call). Most of Maple's automatic simplification rules are pretty obvious, but unfortunately there does not seem to be any documentation about them in Maple's online help system.
```
>
```

Here is another way to see that Maple evaluates the operands of a function call before applying the function. We can use delayed evaluation to prevent Maple from making the function call.
```
> 'f'(g(0),g(0));
```
The right-quotes around **f** prevent Maple from evaluating the name **f** to the function **(x,y)->x-y** so Maple is not able to actually complete the function call. But nothing prevented Maple from evaluating the operands of the function call, and so we see that Maple evaluates the operands before it tries to apply the function.

This last example is actually a very useful trick for figuring out what is sometimes going on with Maple. When we get an error message from Maple or a command does not do what we expect it to do, it can sometimes be useful to delay the evaluation of a Maple command to see what were the exact operands that the command was working with. This may help explain the cause of the error message or why the command did what it did. Here is an example. Give **x** a value.

```
> x := 5;
```

We know that the following command causes an error message.

```
> plot( x^2, x=-2..2 );
```

By delaying evaluation of the name **plot**, we can get more information about what went wrong in the above command.

```
> 'plot'( x^2, x=-2..2 );
```

And of course we see that 5 running from -2 to 2 is what does not make sense.

```
>
```

Exercise: Explain the following sequence of commands.

```
> f := x -> x^2;
> z := 5;
> f('z');
> 'f'(z);
> 'f(z)';
>
```

```
>
```

11.8. Evaluating function definitions

In the last section we emphasized how Maple evaluates function calls. Here we will look at how Maple evaluates a function's definition. We will see that this will also give us some more information on the evaluation of function calls.

```
>
```

Let us give **x** a value.

```
> x := -2;
```

If we define **f** as an expression in **x**, then full evaluation will evaluate **x** right away and so **x** will not really be part of the value of **f**.

```
> f := x^2-1;
```

If we now change **x**, this does not affect **f**.

```
> x := -5;
> f;
```

Now define **g** as a Maple function and use **x** as the independent variable in the definition of **g**.

```
> g := x -> x^2-1;
```

Maple did not use full evaluation here. None of the **x**'s on the right hand side of the assignment

operator were evaluated. If we evaluate a function call using **g**, then the **x** in the definition of **g** will get the operand of the function call, not the current value of **x**.

```
> g(6);
```

Notice that the current value of **x** (which is –5) has nothing to do with the evaluation of **g(6)**.

```
>
```

Here is another example. Suppose we give the name **c** a value and then use it in the definition of **g** along with **x**.

```
> c := -2;
> g := x -> c*x^2-1;
```

Notice that neither **c** nor **x** was evaluated in the definition of **g**.

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

If we evaluate a function call using **g**, then Maple will use the current value of **c** in evaluating the function call but not the current value of **x**. This is because **c** is a parameter in the definition of **g** but **x** is the independent variable (and independent variables get their values from the function call's operands).

```
> g(3);
```

If we now change the value of **c**, then the definition of **g** is unchanged, but the evaluation of **g** in a function call will change.

```
> c := 2;
> eval( g );
> g(3);
```

We can even leave **c** unassigned.

```
> c := 'c';
> g(3);
>
```

So when Maple evaluates a function definition using the arrow operator, it does not evaluate any of the names in the definition, neither the independent variables nor the parameters. When Maple evaluates a function call, then the independent variables get their values from the operands of the function call, and the parameters in the function definition are fully evaluated to whatever values they may currently have at the time of the function call.

```
>
```

So far we have looked at how Maple evaluates a function definition that uses the arrow notation. But we can also define Maple functions using the **unapply** command. Let us see how Maple evaluates a function definition that uses the **unapply** command.

```
> g := unapply( c*x^2-1, x );
```

We got an error message. What happened is that Maple used full evaluation for the **unapply** command. So this last command looked to Maple like the following

```
> g := unapply( 25*c-1, -5 );
```

because **x** has a value (and **c** does not).

```
> x, c;
```
Here is a way to verify this.
```
> 'unapply'( c*x^2-1, x );
```
So the **unapply** command uses full evaluation. Let us see how that affects the function we are defining. First of all, we must have the independent variable (in this case **x**) as an unassigned variable.
```
> x := 'x';
```
Now we can use **unapply**.
```
> g := unapply( c*x^2-1, x );
```
Notice that since **c** is currently unassigned, **c** is a parameter in the definition of **g**.
```
> g(3);
```
We can give **c** different values and get different evaluations of function calls to **g**.
```
> c := -1;
> g(3); g(w);
> c := 2;
> g(3); g(w);
```
What happens if we now use **unapply** again to redefine **g** using the same expression?
```
> g := unapply( c*x^2-1, x );
```
Now, since **c** is an assigned variable and Maple uses full evaluation in the **unapply** command, we no longer have **c** as a parameter in the definition of **g**. The value 2 is now a permanent part of the definition of **g**.
```
> eval( g );
> c := 5;
> eval( g );
```
What if we want to use **unapply** and force **c** to be a parameter in the definition of **g**? Then we would need to delay the evaluation of **c** in the **unapply** command.
```
> g := unapply( 'c'*x^2-1, x );
> eval( g );
> g(x);
```
Why is there a 5 in the last expression?
```
> c := -2;
> g(x);
>
```

In summary, the arrow operator and the **unassign** command use very different evaluation rules when used to define a Maple function. The arrow operator does not evaluate any of the names used in the definition of the function, and the **unapply** command uses full evaluation (except in cases where last name evaluation should be used).
```
>
```

Exercise: Explain each the following three sequences of commands.
```
> h := x -> x^2;
```

```
> g := x -> c*h(x) -1;
> g(3);
> h := x -> x^3;
> g(3);
>

> h := x -> x^2;
> g := unapply( c*h(x)-1, x );
> g(3);
> h := x -> x^3;
> g(3);
>

> h := x -> x^2;
> g := unapply( c*'h'(x)-1, x );
> g(3);
> h := x -> x^3;
> g(3);
> eval( g );
>
```

Exercise: Explain the following sequence of commands.

```
> h := x -> x^3;
> g := h -> h(5);
> g(h);
> h := x -> x^2;
> g(h);
> g := unapply( h(5), h );
> g := unapply( 'h'(5), h );
```

Since **h** has a value, why was there no error message from the last two commands?

```
> eval( h );
```

For example, here we get an error message.

```
> x := 5;
> unapply( h(x), x );
```

Which we get rid of this way.

```
> unapply( h(x), 'x' );
>
```

Exercise: Explain the following sequence of commands.

```
> h := x -> x^3;
> g := h -> h(x);
> g(h);
> x := y;
```

```
> g(h);
> h := x -> x^2;
> g(h);
> g := unapply('h'(x), h);
> x := z;
> g(h);
> g := unapply('h'('x'), h);
> g(h);
>
```

Exercise: Explain with as much detail as you can how Maple evaluates the following commands. What rules of evaluation are used at each step.

```
> x := 0;
> h := x -> x^2;
> g := (h, x) -> h(x);
> g(h, y);
> g := (h, x) -> 'h'(x);
> g(h, y);
> g(sin, Pi);
> g(eval(h), y);
> %;
> eval(g(h, y));
> g(eval(sin), Pi);
```

How would we use **unapply** to define the exact same function **g** that we have right now?

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

The following is not correct. Try to fix it.

```
> g := unapply( 'h'('x'), h, 'x');
```

Explain what happens if you remove any one of the pairs of right-quotes from the **unapply** command. And why does the last **h** in the **unapply** command not even need right-quotes?

```
>
```

```
>
```

11.9. Evaluating dotted names (optional)

In a previous section we looked at what Maple does with the left hand side of an assignment operator and we said that Maple does not evaluate any names it finds there. But this is not entirely true. If there is a dotted or indexed name on the left hand side of an assignment operator, then Maple must do some kind of evaluation. In this section and the next one we look at Maple's rules for evaluating dotted and indexed names.

```
>
```

In the following command, Maple evaluates the **x.0** on the left hand side of the assignment

operator to the name **x0**. Notice that **x.0** is not a name, it evaluates to a name.

```
> x.0 := 1;
```

In the next command, Maple once again evaluates **x.0** to the name **x0**, but it does *not* evaluate the name **x0**, which has a value now.

```
> x.0 := 2;
```

If Maple had evaluated the name **x0**, the last command would have become **1:=2**, which does not make sense.

```
>
```

Now consider the next two commands.

```
> i := 0;
> x.i := x.i + 1;
```

In the last command Maple had to evaluate both the name **i** to the value 0 and then evaluate the dotted notation **x.0** to the name **x0**. In this example Maple did two kinds of evaluations on the left hand side of the assignment operator. But once Maple got to the name **x0**, it stopped evaluating (on the left hand side of the assignment operator, but not on the right hand side). This is a special kind of evaluation, called **evaluate to a name**, and Maple has a special command for performing this kind of evaluation, **evaln** (which is an obvious abbreviation of "**eval**uate to a **n**ame").

```
> evaln( x.i );
```

Evaluation to a name is not the same as full evaluation which is what Maple did with the **x.i** on the right hand side of the last assignment statement and which is what Maple does with the next expression.

```
> x.i;
```

The steps involved with fully evaluating **x.i** are first evaluate **i** to 0 to get **x.0**, then evaluate **x.0** to the name **x0**, then evaluate the name **x0** to the value 3.

It is important to realize that evaluation to a name is not the same as last name evaluation. Here is an example that distinguishes between these two evaluation rules. First make the following assignments.

```
> a, b, c := 'a', 'b', 'c':
> a, b, c := b, c, x->x^2;
```

Here is last name evaluation of **a** (which is the default way for Maple to evaluate **a**).

```
> a;
```

And here is **a** evaluated to a name.

```
> evaln( a );
```

Since **a** is a name, **evaln** evaluates **a** to itself. So last name evaluation and evaluation to a name are two different ways to evaluate **a**. Here is a slightly more elaborate example that distinguishes between last name evaluation and evaluation to a name.

```
> i := 0; x0 := 'w'; w := z -> z^2;
```

Here **x.i** evaluated using last name evaluation (why?).

```
> x.i;
```

And here is **x.i** evaluated to a name.

```
> evaln( x.i );
```
For the sake of completeness, here is full evaluation of `x.i`.
```
> eval( x.i );
>
```

Exercise: With `i`, `x0`, and `w` assigned as in the last example,
```
> i := 0; x0 := 'w'; w := z -> z^2;
```
explain in detail the evaluation rules used to evaluate each part of the following assignment statement.
```
> x.i := x.i + x.i(4);
```
What can you say about the value of `x0` after the last assignment?
```
>
```

When Maple evaluates a dotted notation to a name, it uses full evaluation on the right hand side of the dot operator (unless it should use last name evaluation). Here is an example. First make the following assignments.
```
> i, j, k := 'i', 'j', 'k':
> i, j, k := j, k, 0;
```
Here is `w.i` evaluated to a name, using full evaluation of the `i`.
```
> w.i;
```
The next few commands control the level of evaluation of the `i` in `w.i`.
```
> w.(evaln(i));
> w.(eval(i,1));
> w.(eval(i,2));
> w.(eval(i,3));
>
```

Here is an example that uses last name evaluation on the right hand side of the dot operator.
```
> i, j, k := 'j', 'k', x-> x^2;
```
Here is `w.i` evaluated to a name, using last name evaluation of the `i` (the last name that `i` evaluates to is `k`).
```
> w.i;
```
If we change the value of `k` from a function to an expression, then `x.i` will be evaluated using full evaluation on the right hand side of the dot (but Maple will not be able to evaluate the dot, since the right hand side of the dot will not evaluate to a name).
```
> k := x^2;
> w.i;
>
```

Exercise: First make the following two assignments.
```
> i := 'j'; j := 0;
```
Explain why the following two expressions are different. Explain in detail the steps that Maple uses

to evaluate each of the two expressions. (Hint: You need to know about the precedence of operations.)

```
> w.eval(i,1);
> w.(eval(i,1));
>
```

Here is an example of a typical and important use for the **evaln** command. Suppose we automatically create a bunch of new variables and assign them values.

```
> seq( random.i=rand(), i=1..10 );
> assign( % );
```

Now suppose that we want to do a simple calculation with each of the variables and display the results in equations of the following form.

```
> 'random3'-'random2' = random3-random2;
```

We could just list nine commands in a row just like the last one, but it would be better if we could automate this. Here is an attempt to do this.

```
> seq( 'random.(i+1)'-'random.i' = random.(i+1)-random.i, i=1..9
  );
```

That did not work. The right-quotes delayed evaluation of the whole dotted name. What we want on the left hand side of every equal sign is to evaluate the expressions **i** and **i+1** and then evaluate the dot operators, but do not evaluate the resulting names. And that is exactly what we mean by "evaluate to a name" so let us use the **evaln** command.

```
> seq( evaln(random.(i+1))-evaln(random.i)
>      = random.(i+1)-random.i, i=1..9 );
>
```

The dot operator has two sides to it. So far we have only looked at what Maple does on the right hand side of the dot operator. What about the left hand side? First make the following assignments.

```
> i := 0;
> x := 2;
```

Now evaluate **x.i**.

```
> x.i;
```

In the last command Maple evaluated the **i** but not the **x** in the dot notation. Maple will not evaluate the name that is on the left side of the dot operator. But here is an example that seems to contradict this rule.

```
> j := 1;
> x.i.j;
```

Maple evaluated the **i** and the **j** but not the **x**. The **i** is on the left side of a dot operator, so did Maple evaluate a name on the left side of a dot operator? The answer is no, because of the order that Maple did the evaluations in. The online help states that the dot operator is left associative. So in the expression **x.i.j**, the dot between the **x** and the **i** is evaluated first. This puts **i** to the right of the dot, so it is evaluated to get the name **x0**. Then the dot on the right side of the name **x0** is evaluated, so **j** is evaluated to 1 and the dot returns the name **x01**.

We just mentioned that the dot operator is left associative. However, the following example shows that Maple does not allow the use of parentheses for grouping around the left dot.

```
> (x.i).j;
```

The next exercise shows that Maple does allow the use of parentheses for grouping around the right dot.

```
>
```

Exercise: First make the following assignments.

```
> i, j := 0, 1;
```

Explain in detail how Maple evaluates each of the following expressions.

```
> x.i.('j');
> x.('i').j;
> x.('i').('j');
> x.(i.j);
> x.(i.('j'));
> x.(('i').j);
>
```

Let us look at an example of unassigning a dotted name.

```
> x := 'x';
> i := 235;
> x.i := 0;
```

Now suppose you wish to unassign `x.i` but you do not happen to know the value if `i` (knowing the value of `i` would make the problem too easy). Situations like this occur often while programming Maple. The following does not work.

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

This last command did change the value of `x.i` but it did not unassign it.

```
> x.i;
```

In fact it created a recursive variable name. Here is how we can unassign `x.i` without having to know the value of `i`.

```
> x.i := evaln( x.i );
```

Now `x.i` is unassigned (but not `i`).

```
> x.i;
>
```

Maple will not evaluate dotted names as the independent variables in a function definition.

```
> f := x.2 -> (x.2)^2;
```

And there can be problems from using the independent variables of a function in dotted names.

```
> f := (x,y) -> x.y;
> f('i',3);
> f(i,3);
```

```
> 
```

Exercise: What prevents `4.3` from being interpreted as the concatenation of `4` and `3` together to make `43`?

```
> 
```

```
> 
```

11.10. Evaluating indexed names (optional)

Now let us consider the evaluation of indexed names. Before going into the details of the evaluation rules, let us look at an example. Consider the following two commands.

```
> x.1 := 0;
> x[1] := 0;
```

In the first command, Maple *evaluates* `x.1` to the name `x1`. In the second command, `x[1]` *is* a name. Maple does not need to evaluate it. (The subscripted version of `x[1]` in the output is just a "prettyprinted" version of this name. This is analogous to the variable name `theta` and the way Maple prettyprints it as a Greek letter.) There is even a way to ask Maple to verify this distinction for us.

```
> type( 'x.1', name );   # Is x.1 a name?
> type( 'x[1]', name );  # Is x[1] a name?
```

Similarly, in the next two commands `x[i]` *is* a name and Maple does not need to evaluate anything on the left hand side of the second assignment operator.

```
> i := 'i';
> x[i] := -1;
```

We can even check with Maple.

```
> type( 'x[i]', name );
```

Now, to make things interesting, in the next two commands, Maple *does* evaluate `x[i]` on the left hand side of the assignment operator to get the name `x[2]`.

```
> i := 2;
> x[i] := -2;
```

Look at the difference in the following two evaluations.

```
> x[i];
> x['i'];
```

Recall that we have said that assigning values to an indexed name creates a table. Here is what is in the table named `x`.

```
> op( x );
```

Notice that the table contains entries for both the names `x[i]` and `x[2]`. The following `evaln` command can reasonably return two different names. It is not clear which one it will return.

```
> evaln( x[i] );
```

(The help page for `evaln` does not explain why Maple choose this name over the other one.) Now let us turn to the explicit rules that Maple uses for evaluating indexed names.

```
> 
```

An indexed name has two parts to it, the part in front of the brackets and the part inside the brackets. We will call the part in front of the brackets the **header**, and the part inside the brackets the **index**. Let us look at how Maple evaluates each of these two parts.

The index of an indexed name is always fully evaluated (unless the situation calls for last name evaluation). We saw this in the examples just above. Here are a few more examples.

```
> a := 'b+c': b := 3: c := 'u': u := 'u':
> x[a];
> x[u.(1..3)] := x[i];
```

Here is an example of last name evaluation of an index.

```
> g := 'f';
> f := x->x^2;
> x[g] := eval( g );
```

If we change the definition of `f` to be an expression, then the `g` in `x[g]` will be fully evaluated.

```
> f := x^2;
> x[g] := g;
```

At this point, it is interesting to look at the table named `x` that we have been building up.

```
> op( x );
>
```

Now let us turn to the evaluation rules for the header part of an indexed name. The header of an indexed name need not be evaluated the same way on both sides of an assignment operator. Consider this example.

```
> a := 'a': b := 'b':
> w := a; a := b;
> w[w] := w[w];
```

Notice that the first `w` in `w[w]` was fully evaluated on the right hand side of the assignment operator but it was not evaluated on the left hand side of the assignment operator. By contrast, the dot operator acts the same on both sides of the assignment operator.

```
> a := 'a': b := 'b':
> w := a; a := b;
> w.w := w.w;
>
```

Exercise: Suppose that we combine the last two examples together in one execution group. Explain why the assignment statement `w.w := w.w` has a result that is different below from what it was just above.

```
> a := 'a': b := 'b':
> w := a; a := b;
> w[w] := w[w];
> w.w := w.w;
>
```

The previous examples may leave the impression that the header of an indexed name is not evaluated when it is on the left hand side of an assignment operator. But that is not quite the case. Here are two counter examples.

```
> w := 'a'; a := 2;
> w.w[w] := w[w];
> w[w[w]] := w[w];
```

Notice the following observation. In the last two assignments, the indexed name `w[w]` on the left side of the assignment is not the name being assigned to (it is just part of the name), so the header name `w` is fully evaluated in each case.

Here is a rule for evaluating the header of an indexed name. If a name is the header of an indexed name *that is being assigned to*, then the name is not evaluated, otherwise, the name is fully evaluated (unless last name evaluation is appropriate).

This rule for the header of an indexed names is actually a special case of the "no evaluation" rule that we stated in a previous section, that is, when a name is being assigned to, the name is not evaluated. When an indexed name is being assigned to, the header of the indexed name is not evaluated.

```
>
```

Here are some more examples of evaluating indexed names. This first example uses both last name and full evaluation.

```
> a := 'a'; f := 'g'; g := 'h'; h := x->a*x^2;
```

Here is last name evaluation in an indexed name.

```
> f[f];
```

Here is full evaluation.

```
> f(u)[f(y)];
```

Here is last name, full, and no evaluation all at the same time.

```
> f[f(3)] := f[f];
```

This last assignment created a table named `f`. Here is the table.

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

Exercise: Explain in detail how Maple arrived at the output for the second of the following two commands.

```
> g[x] := x^2+1;
> g[g(0)];
>
```

Here are two examples that will cause us to re-evaluate our evaluation rules for indexed names.

```
> unassign('a', 'b', 'c'):
> w := a+b+c;
> w[w];
```

```
> w[w] := 0;
```
In the second to last command, both parts of the indexed name were fully evaluated since the indexed name was not being assigned to. In the last command, the same indexed name was evaluated differently since it was being assigned to. And the assignment created a table named **w**.
```
> op( w );
```
Now here is a bit more complicated example which shows that our rule for evaluating an indexed name is not quite correct.. Let **w** be a name for a list.
```
> w := [a,b,c];
> w[w];
> w[w] := 0;
```
To understand what happened in the last command, consider this next assignment statement.
```
> w[2] := 0;
```
In each of the last two commands, the header of the indexed name was evaluated so that Maple could know that the indexed name referred to a list. In the second to last command, the index was not appropriate for the list, hence the error message. If Maple had not evaluated the header name, then the assignment would have created a table named **w**. But **w** is still a list.
```
> w;
```
So in some circumstances, Maple will evaluate the header of an indexed name that is being assigned to. So we should try to restate our rule for evaluating an indexed name. Here is another attempt at a rule.

In an indexed name, the index is always fully evaluated (unless last name evaluation is appropriate). If an indexed name is not being assigned to, then the header is also fully evaluated (unless last name evaluation is appropriate). If an indexed name is being assigned to, then the header must be a name. If an indexed name is being assigned to, and the header name does not evaluate to an expression sequence, set, list, or table, then the header name is not evaluated (and the assignment creates a new table with the header name as its name). If an indexed name is being assigned to, and the header name evaluates to an expression sequence or set, then there is an error message. If an indexed name is being assigned to, and the header name is a name for a list or a table, then the header name is evaluated and an element of the list or table is assigned to (assuming that the index is appropriate). If an indexed name is being assigned to, and the header name evaluates indirectly to list or table, then things get complicated. Here are a couple of examples that demonstrate the complications.
```
>
```

Let **w** be indirectly a name for a table.
```
> x := 'x'; w := x;
> x[apple] := pear;
```
Now assign something to an indexed name with **w** as the header.
```
> w[2] := Pi;
```
It seems like the last command created a table named **w** and put an entry in the table. Let us check.
```
> eval( w );
```
But this looks like the table named **x**. The following command shows that there is in fact no table

named **w**, **w** is still an indirect name for **x**.

```
> eval(w, 1);
```

So in the assignment statement **w[0]:=Pi**, even though the output from the command made it look like **w** was not evaluated, **w** was evaluated to be **x** and the assignment was really made to **x[0]**. So if an indexed name is being assigned to, and the header name evaluates indirectly to a table, then the header name is evaluated and an element of the table is assigned to.

```
>
```

Now let **w** be indirectly a name for a list.

```
> x := 'x'; w := x;
> x := [apple, pear, orange];
```

Now assign something to an indexed name with **w** as the header.

```
> w[2] := Pi;
```

The output once again makes it look like a table was created with the name **w**. Let us check.

```
> eval( w );
```

It looks like maybe **w** is still pointing indirectly to the list **x**. Let us check.

```
> eval(w, 1);
```

So **w** is not pointing to **x**. Let us check **x**.

```
> x;
```

So the assignment statement **w[2]:=Pi** made a copy of the list **x**, made **w** a name for the copy, and then made the assignment in the copy. So if an indexed name is being assigned to, and the header name evaluates indirectly to a list, then the header name is not evaluated, a copy of the list is made and given the name of the header, and an element of the new list is assigned to (assuming that the index is appropriate for the list).

```
>
```

Exercise: First make the following assignments.

```
> l := '[a,b,c]';
> s := 'a+b+c';
> i := 2;
```

Explain how Maple arrived at each of the following outputs.

```
> l[i];
> s[i];
> 'l[i]';
> 's[i]';
> 'l'[i];
> 's'[i];
> l['i'];
> s['i'];
> l[0];
> s[0];
> l[abc];
```

```
> s[abc];
> l[abc] := 0;
> s[abc] := 0;
>
```

Exercise: Is there any difference between the following names, `w.(w[w])`, `(w.w)[w]`, and `w.w[w]`? Investigate this with `w` both unassigned and assigned.

```
>
```

```
>
```

11.11. Online help for evaluation rules

There does not seem to be any one online help page that describes, or even summarizes, Maple's evaluation rules. Little bits and pieces of information are scattered in a few help pages.

A definition of full evaluation and examples of levels of evaluation can be found in the following help page.

```
> ?eval
```

Examples of delayed evaluation and using right quotes to convert an assigned name into an unassigned name are described in the next two help pages.

```
> ?quotes
> ?uneval
```

The last help page also describes the special case of unassigning an indexed or dotted name where, instead of using right quotes, you need to use the `evaln` command.

```
> ?evaln
```

The evaluation rule used on the left hand side of an assignment operator is described in the next help page.

```
> ?assignment
```

A little bit about evaluating dotted names can be found in the following page.

```
> ?dot
```

And some information about the evaluation of indexed names is in the next help page.

```
> ?selection
```

```
>
```