Maple for Math Majors

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

12. Data Structures in Maple

12.1. Introduction

Within Maple's internals, there is a big difference between expressions and functions. Expressions are what computer scientists call data structures, and functions are what they would call procedures. To put it very briefly, a data structure is a collection of information, and a procedure is a collection of instructions. In order to explain this difference better, in this and the next few worksheets we look at some elementary examples of data structures and procedures in Maple. In particular, in this worksheet we look at Maple's basic data structures, at how a data structure can be manipulated, and at how Maple stores an expression as a data structure. In the next worksheet we look at some simple examples of Maple procedures and how mathematical functions can be represented as procedures. In the last worksheet we look at how a procedure can manipulate a data structure, and this will give us an idea of how Maple does its work. Many Maple commands (like **`factor`**, **`simplify`**, **`expand`**) are procedures, written in the Maple programming language, that manipulate data structures. In fact, that is what much of programming and computer science is all about, procedures that manipulate data structures.

```
> 
```

12.2. Basic data structures in Maple

To really understand what makes expressions and functions different in Maple we need to get a sense of what data structures and procedures are. The rest of this worksheet is a very basic introduction to how Maple handles data structures.

Let us try to give a definition of a "data structure". Earlier we said that a data structure is a collection of information but this is an incomplete definition. Since information is the same as "data", we have left off the "structure" part of "data structure". A data structure is a collection of information that has been given a structure or organization. In addition, a data structure often has associated with it a special way of interpreting the data. As we will see below, the structure (and interpretation) aspect of a data structure is as important as the data aspect. Two different data structures can contain the exact same pieces of data but give that data different structures (or interpretations).

Now we will look at a few of the **basic data types** in Maple. Every data structure has a **data type** which describes, or names, the kind of data structure it is. The three most basic kinds of data structures in Maple have the data types **expression sequence**, **list**, and **set**. Maple also has several

numeric data types which are used to represent different kinds of numbers. Each of these kinds of data structures, plus a few other important data types, are described in detail in the following subsections.

```
[ >
```

12.2.1. Expression sequences

First we look at **expression sequences**. We said that a data structure is an organized collection of information (or data). An expression sequence in Maple is the simplest way that some data can be organized; it is a list of things separated by commas. The things (i.e. the pieces of data) in the expression sequence can be anything that Maple can work with, e.g., numbers, expressions, functions, other data structures, etc.

Here is an example of an expression sequence with nine pieces of data in it.

```
[ > 1, 2, abc, x^2+1, `hi world`, Pi, x -> x^2, 1/2, 1;
```

Let us give this expression sequence a name so that it is easier to work with.

```
[ > stuff := %;
[ > stuff;
```

Maple's **whattype** command is used to ask "**what** is the data **type**" of a data structure .

```
[ > whattype( stuff );
```

Maple uses **exprseq** as an abbreviation for "**expr**ession **seq**uence".

Maple's **nops** command is used to ask how many pieces of data are in a data structure. The name **nops** is an abbreviation of " **n**umber of **op**erand**s**" and the word "operand" is often used to mean a piece of a data structure. Notice that in the next command we need to put square brackets around the name **stuff** in order to use **nops**. (Try removing the brackets.) In the next subsection we will see why these brackets need to be there.

```
[ > nops( [stuff] );
```

Maple's **op** command is used to access individual elements of a data structure (the name **op** is of course an abbreviation of the word "**op**erands"). It can be used to access just one element or a range of elements from a data structure. The next command asks for the fifth thing in **stuff**.

```
[ > op( 5, [stuff] );
```

The next command asks for the fifth through seventh things in **stuff**.

```
[ > op( 5..7, [stuff] );
```

Notice that the **op** command used with a range returned an expression sequence which can be thought of as a subsequence of the original expression sequence. Here is another notation that allows us to access an individual piece of data from an expression sequence, a subsequences of data, or the whole sequence.

```
[ > stuff[5];
[ > stuff[5..7];
[ > stuff[];
[ >
```

There is no special interpretation associated to the data in an expression sequence. The only real

organization that Maple gives to an expression sequence is that Maple remembers the order that the data items were given in. So the following two expression sequences are not considered the same data structure, even though they contain the exact same data items.

```
> seq1 := a, b, c, d;
> seq2 := b, a, c, d;
> evalb( seq1=seq2 );
```

Remember, **seq1=seq2** is an equation that could be either true or false. The Maple command **evalb** is used to determine if the equation is true or false. Since the equation is false, the expression sequences are not the same data structure. Another way to put this is that **seq1** and **seq2** have the same data type (**exprseq**), and they contain the same data, but they are not the same data structure.

In the next two subsections we will see what brackets (**[]**) and braces (**{}**) mean to an expression sequence. Parentheses however are ignored by expression sequences. The following commands all define the same expression sequence.

```
> the, very, same, thing;
> (the, very), (same, thing);
> (the, ((very, same), thing));
>
```

Expression sequences can be combined into larger expression sequences. Let us define two expression sequences.

```
> seq1 := a, b, c, d;
> seq2 := u, v, w, x, y, z;
```

Here is one way to combine these two into a third expression sequence.

```
> seq3 := seq1, seq2;
```

There are many ways to combine the data from two expression sequences into one expression sequence. What we just did was to **concatenate** **seq1** and **seq2**. Here is another way to combine these two expression sequences.

```
> seq3 := seq2[1..3], seq1, seq2[4..6];
>
```

Exercise: With **seq1** and **seq2** defined as above,

```
> seq1 := a, b, c, d;
> seq2 := u, v, w, x, y, z;
```

what are the differences between the following four commands

```
> seq3 := seq1, seq2:
> seq4 := seq1, 'seq2':
> seq5 := 'seq1', 'seq2':
> seq6 := 'seq1, seq2':
```

since they all seem to produce the same result.

```
> seq3;
```

```
> seq4;
> seq5;
> seq6;
>
```

Exercise: With **seq1** and **seq2** still defined as above, what does the following command do?

```
> seq1 := seq1, seq2;
>
```

There is one very important expression sequence that has a special name. The empty expression sequence is called **NULL**.

```
> NULL;
```

When Maple evaluates the name **NULL**, the value it gets is the empty expression sequence, and since there is not much to print out for the empty expression sequence, Maple does not print anything.

```
> whattype( 'NULL' );
> whattype( NULL );
> nops( [NULL] );
```

The last three commands verify that **NULL** is a name for an expression sequence and that there are no operands in the expression sequence. The **NULL** expression sequence comes up fairly often, but usually in a pretty invisible way. For example, the command

```
> solve( x^10-sqrt(2)*x^3-Pi=0, x );
```

did not seem to return any result. But it did return a result. It returned **NULL**.

```
> whattype( solve( x^10-sqrt(2)*x^3-Pi=0, x ) );
> evalb( solve( x^10-sqrt(2)*x^3-Pi=0, x )=NULL );
```

Whenever a Maple command seems to do its work "silently", without any output, the command is really returning the **NULL** expression sequence.

```
>
```

Maple has two very important and common uses for expression sequences. First, many Maple commands, when they want to express several outputs, put the output items in an expression sequence. The most common example of this is the **solve** command.

```
> x:='x':
> solve( x^3-4*x^2-5*x=0, x );
```

There are three solutions to the equation in the **solve** command and **solve** puts the three answers in an expression sequence. We can ask Maple to verify this by using the **whattype** command again.

```
> whattype( % );
```

If we modify the **solve** command a little bit and put braces around the last **x** in the command, then we get the solutions in a different form.

```
> solve( x^3-4*x^2-5*x=0, {x} );
```

The output is still an expression sequence, but now each item in the sequence is an equation

with braces around it.

```
> whattype( % );
```

Let us give the expression sequence a name.

```
> solutions := %%;
```

Now we can access individual solutions using the notations we gave above.

```
> op( 1, [solutions] );
> solutions[2];
> solutions[2..3];
> assign( solutions[3] );
> x;
> x:='x';
>
```

We just saw that Maple uses expression sequences to represent multiple outputs from a command. Maple's other common use for expression sequences is to represent multiple inputs to a command (or function). We want to demonstrate that the inputs to a Maple function are always contained in an expression sequence. Here is an example. The next command defines and names an expression sequence.

```
> test_sequence := x^2, x=-2..2, color=blue, axes=box;
```

This sequence has four items in it.

```
> nops( [test_sequence] );
```

Notice that **test_sequence** looks like the inputs to a **plot** command. And we can use it exactly that way.

```
> plot( test_sequence );
```

In the next command, the evaluation of **plot** is delayed but **test_sequence** is evaluated so that we can see how the input to **plot** really is an expression sequence.

```
> 'plot'( test_sequence );
```

Now evaluate **plot**.

```
> %;
```

Here is another example. First, let us define our own multivariate function.

```
> f := (x,y,z,w) -> x+y+z+w;
```

Here is an expression sequence.

```
> es := b, c, d;
```

Now evaluate the function with what looks like (but is not) two inputs.

```
> f(a,es);
```

Let us look at what the function really saw as its inputs.

```
> 'f'(a,es);
```

Suppose we delay the evaluation of **es**. What do you think will be the result of the following function call?

```
> f(a,'es');
>
```

The last several examples have shown that expression sequences are basic for representing both

the input and output of Maple commands and functions.

```
> 
```

Maple has a special command, **seq**, for creating expression sequences. Here are some examples of its use.

```
> seq( x^i, i=-2..5 );
> seq( Pi, i=-2..2 );
> seq( 1/n, n=1..10 );
> seq( evalf(1/n), n=1..10 );
> 
```

```
> 
```

12.2.2. Lists

The next most basic data type in Maple is a **list**. A list is an expression sequence with square brackets around it.

Recall that **stuff** is our expression sequence from the last section. We redefine it here for convenience.

```
> stuff := 1, 2, abc, x^2+1, `hi world`, Pi, x -> x^2, 1/2, 1;
```

An expression sequence with brackets around it gives us a list.

```
> [stuff];
```

Let us give this list a name so that we can work with it.

```
> liststuff := %;
```

Let us check the data type of our data structure, and also how many elements are in it.

```
> whattype( liststuff );
> nops( liststuff );
```

The **op** command with only one argument returns all the elements in the data structure.

```
> op( liststuff );
```

Notice that the **op** command returned an expression sequence. So the **op** command can be used to convert a list into an expression sequence.

```
> 
```

We can access an individual element from a list or a range of elements.

```
> op( 5, liststuff );
> op( 5..7, liststuff );
```

Now compare the previous two commands with the next two.

```
> liststuff[5];
> liststuff[5..7];
```

Notice that **liststuff[5]** is an alternate notation for **op(5, liststuff)**, but **liststuff[5..7]** is not quite the same as **op(5..7, liststuff)**. How are they different? Should **liststuff[5]** really have been the same as **op(5, liststuff)**? To make things even a bit more confusing, look at what the next command does. What is it an

alternative to?

```
> liststuff[];
>
```

Surprisingly enough, we can even use negative numbers to access items from a list. The index `-1` means the last item in a list and as the negative index decreases we access items from right to left through the list.

```
> op( -1, [a, b, c, d, e] );
> op( -2, [a, b, c, d, e] );
> op( -3, [a, b, c, d, e] );
```

But negative indexes do not always work the way you might think they would. Based on the last three commands, what would you expect the next command to output?

```
> op( -1..-3, [a, b, c, d, e] );
```

What about this command?

```
> op( -3..-1, [a, b, c, d, e] );
```

Using negative indexes can lead to very non intuitive results. Consider the next two examples.

```
> op( -4..4, [a, b, c, d, e] );
> op( -3..3, [a, b, c, d, e] );
```

You might think that in any range `m..n` you must have `m` less than `n`. But things are not that simple.

```
> op( 4..-4, [a, b, c, d, e] );
> op( 3..-3, [a, b, c, d, e] );
```

In fact, one way to list all the data items in a list is to use the range `1..-1`, i.e., from first to last.

```
> op( 1..-1, [a, b, c, d, e] );
```

On the other hand, the range `-1..1` (from last to first?) causes an error.

```
> op( -1..1, [a, b, c, d, e] );
```

We can use negative indexes with the bracket selection notation also.

```
> [a, b, c, d, e][-3..-1];
>
```

Exercise: Come up with a simple rule for what is allowed in a range `m..n` when it is used for accessing items in a list and either `m`, `n` or both is negative.

```
>
```

We did not get the `op` command to produce a range of elements from a list in reverse order, but we can do this if we use the `seq` command in combination with `op`.

```
> list1 := [a, b, c, d, e];
> [ seq( op(-i, list1), i=3..5 ) ];
> [ seq( op(-i, list1), i=1..nops(list1) ) ];
```

We can abbreviate the above commands by using index notation.

```
> [ seq( list1[-i], i=3..5 ) ];
```

Notice that the following command does not work.

```
> [ seq( list1[i], i=-3..-5 ) ];
>
```

Exercise: Let `list1` be defined as in the last example.

```
> list1 := [a, b, c, d, e];
```

For each of the following two commands, rewrite the command so that it produces the same result but is easier to read.

```
> [ seq( op(-i, list1), i=-5..-3 ) ];
> [ seq( op(i, list1) , i=-5..-3 ) ];
>
```

A special case of the `op` command is `op(0,` data_structure`)` which, instead of accessing a piece of data from the data structure, (usually) returns the data type of the data structure.

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

For most data structures, `op(0,` *data_structure*`)` is the same as `whattype(`*data_structure*`)`. But we will see later that for some data structures, `op(0,` data_structure`)` returns something other than the data type. Also, notice that the following syntax does not do what you think it might.

```
> liststuff[0];
>
```

There is no special interpretation associated to the data in a list. The only real organization that Maple gives to a list is that Maple remembers the order of the data items in the list. So the following two lists are not considered the same, even though they contain the exact same data items.

```
> list1 := [1, 2, 3, 4];
> list2 := [2, 1, 3, 4];
> evalb( list1=list2 );
```

As with the example with expression sequences, we would say that `list1` and `list2` have the same data type (`list`), and they contain the same data items, but they are not the same data structure.

```
>
```

Exercise: Suppose we have two lists.

```
> list1 := [a, b, c, d];
> list2 := [u, v, w, x];
```

Find a command that will combine all the data from the two lists into one list.

```
> list3 := ???
>
```

Exercise: Given the following two expression sequences

```
> es1 := a, b, c, d, e;
> es2 := u, v, w, x, y;
```
use a single **seq** command to create a third expression sequence **es3** that is made by alternating elements from **es1** with elements from **es2**.
```
> es3 := ???
>
```

If **L** is the name of a list, then Maple will use full evaluation when it is asked the find the value of **L**.
```
> L := [a, b, c, d];
> a := 'b'; b := 2; c := 'b'+1; d := 4;
> L;
```
Notice that several levels of evaluation were needed to fully evaluate **L**. The next three commands show the levels of evaluation.
```
> eval( L, 1 );
> eval( L, 2 );
> eval( L, 3 );
```

Recall that in the section on expression sequences we had to put brackets around an expression sequence before we could use the **nops** command to find out the number of operands in the expression sequence.
```
> nops( stuff );
> nops( [stuff] );
```
By now we know that the brackets convert the expression sequence into a list. Why do we need the brackets? Because the **nops** command is defined to accept only one input, the data structure that we are interested in. But when that data structure is an expression sequence, **nops** thinks that it has multiple inputs, which is a mistake, so **nops** returns an error message. By putting square brackets around **stuff** and converting it into a list we do not change the number of operands in the data structure (**stuff** and **[stuff]** have the same number of operands) but now **nops** thinks it only has one input, the list. Here is a way to see this. In the following command, **nops** is prevented from being evaluated by the right quotes, but **stuff** is evaluated.
```
> 'nops'( stuff );
```
So the **nops** command thinks it has nine inputs, but it was designed to only have one input.
```
> %;
```
On the other and, when we put the brackets around **stuff**, it then appears to **nops** as a single input which is a list.
```
> 'nops'( [stuff] );
> %;
>
```

Exercise: Can the **op(0,** *data_structure***)** form of the **op** command be used somehow on expression sequences to get the data type?

```
> 

> 
```

12.2.3. Sets

Our third basic data type in Maple is called a **set**. A set looks like an expression sequence with curly braces around it, but we will see that sets are a little more subtle than that. A set data structure is usually used to represent the idea of a set from mathematics. So set data structures in Maple act a lot like sets in mathematics. Here is an example. Let us make a set out of our original expression sequence `stuff` by putting braces around `stuff` and then giving the resulting set a name. First, let us redefine the expression sequence `stuff`.

```
> stuff := 1, 2, abc, x^2+1, `hi world`, Pi, x -> x^2, 1/2, 1;
```

Now create a set by putting a pair of braces around `stuff`.

```
> {stuff};
```

Now give this set a name.

```
> setstuff := %;
```

Let us check the data type and number of operands for `setstuff`.

```
> whattype( setstuff );
> nops( setstuff );
> nops( [stuff] );
```

Right away we should notice that something is different. First of all, for a set the order of the data items is not important so Maple chooses its own order to put them in. Second, sets cannot have repeated items in them, so there is only one `1` item in `setstuff`, but there were two `1` items in `stuff` (and also `liststuff`, the list version of `stuff`).

The `op` command still works like it did with lists.

```
> op( 5, setstuff );
> op( 5..7, setstuff );
> op( 5..-1, setstuff );
```

We can also use this alternate notation.

```
> setstuff[5];
> setstuff[5..7];
> setstuff[5..-1];
```

Notice that, as with lists, `setstuff[5]` is the same as `op(5, setstuff)` but `setstuff[5..7]` and `op(5..7,setstuff)` are different. And we can use the following alternative to `whattype`.

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

As we stated just above, the elements of a set data structure are interpreted as the elements of a mathematical set, so the following two sets are the same as data structures.

```
> set1 := {1,2,3,4,4};
> set2 := {2,1,3,4};
```

```
> evalb( set1=set2 );
>
```

We can convert a set into an expression sequence by using the **op** command.

```
> op( setstuff );
```

Or we can use index notation.

```
> setstuff[];
```

We can convert a set into a list by using **op** and a pair of brackets.

```
> [op( setstuff )];
```

Notice that this last list is not the same as **liststuff** (why?). The following command uses a lot of brackets to convert a set into a list.

```
> [setstuff[]];
>
```

Exercise: How would you convert a list into a set?

```
>
```

Maple has commands for doing common mathematical operations with sets. Here are two sets.

```
> set1 := {a, b, c, d};
> set2 := {c, d, e, f};
```

We can form their union and intersection, and also find one set minus the other.

```
> set1 union set2;
> set1 intersect set2;
> set1 minus set2;
>
```

Here is a simple example of how the difference between a list and a set can be important in Maple. In the following **plot** command there is a list of functions and a list of colors. Since the lists are ordered, the functions and colors can be associated in an obvious way.

```
> plot( [cos(x), sin(x)], x=-2*Pi..2*Pi, color=[blue, green] );
```

In the next command we have a set of functions and a list of colors. Since the set is not ordered, there is no obvious way to associate a color with a function. In my Maple session, the cosine ends up green and the sine ends up blue, the opposite of what was expected, but this coloring can change in the next Maple session. Orderings for sets are session dependent and can change from one Maple session to another. If your session seems to be coloring the graphs the "correct" way, try switching the order of the functions in the set and re-executing the **plot** command; the coloring of the graphs should end up the "wrong" way.

```
> plot( {cos(x), sin(x)}, x=-2*Pi..2*Pi, color=[blue, green] );
>
```

Exercise: Explain why the empty expression sequence needs a special name, **NULL**, but the empty set and the empty list do not need special names.

```
> 

> 
```

12.2.4. Some numeric data types

Every kind of number that Maple can work with is actually a data structure with some kind of **numeric** data type. Here are some examples.

```
> whattype( 5 );
> whattype( 1/3 );
> whattype( 1.0 );
```

Let us look at these numeric data types a bit more carefully, in particular the `fraction` and `float` data types.

A fractional number is represented in Maple by a data structure of type `fraction`. A `fraction` is stored as two integers, the **numerator** and the **denominator**. Let us look at an example, `22/7`.

```
> whattype( 22/7 );
```

What are the pieces of data in this data structure?

```
> op( 22/7 );
> op( 1, 22/7 );
> op( 2, 22/7 );
```

What are the types of these data items?

```
> whattype( op(1, 22/7) );
> whattype( op(2, 22/7) );
```

To Maple, `22/7` is a data structure with data type `fraction` and it contains the two data items `22` and `7` in that order. On the other hand, `7/22` is also a `fraction` type data structure that contains the two data items `22` and `7`, but in a different order.

```
> op( 7/22 );
> op( 1, 7/22 );
> op( 2, 7/22 );
```

So `22/7` and `7/22` are data structures with the same data type and they contain the same data items, but they are not equal data structures. Notice that this is very analogous to the lists `[22,7]` and `[7,22]`, which are two data structures with the same data type and contain the same data items, but `[22,7]` and `[7,22]` are not equal data structures.

```
> 
```

Exercise: Why are `22/7` and `[22,7]` not equal data structures? They contain the same data items in the same order. What distinguishes them?

```
> op( 22/7 );
> op( [22,7] );
> 
```

If a fraction is negative, Maple puts the minus sign with the numerator data item.

```
> op( -1/2 );
> op( 1/(-2) );
```

So `-1/2` and `1/(-2)` are stored as the same data structure since they have the same data type (`fraction`) and the same data elements, `-1` and `2`, in the same order.

```
>
```

Now we shall turn to decimal numbers. A decimal number is represented in Maple by a data structure of type `float`. Like a `fraction`, a `float` is stored as two integers, the **mantissa** and the **exponent**. A decimal number like `.33` is thought of by Maple as `33*10^(-2)`. The `33` is called the mantissa and is stored as the first item in the `float` data structure, and `-2` is called the exponent and is stored as the second item of the `float` data structure.

```
> whattype( .33 );
> op( .33 );
> whattype( op(1, .33) );
> whattype( op(2, .33) );
>
```

The number `3.3` is thought of by Maple as `33*10^(-1)`, and the number `33.` is considered `33*10^0`.

```
> op( 3.3 );
> op( 33. );
```

This gives you an idea why decimal numbers in Maple are called "floats", or "floating point numbers". All three of the decimal numbers `.33`, `3.3`, and `33.` have the same mantissa and they are distinguished by their different exponents. The exponent causes the decimal point to "float around" the mantissa to give different decimal numbers. The term "floating point numbers" is not unique to Maple. It is used throughout engineering and computer science to mean the way that computers represent decimal numbers.

```
>
```

Do not make the mistake of thinking that Maple floats are just Maple's version of real numbers. Maple floats are more subtle than that. For example the fraction 1/4 is equivalent as a real number to the decimal number .25 which is also equivalent to the decimal numbers .250 and .2500 since the trailing zeros do not change anything as far as real number are concerned. But not in Maple. First of all, in Maple `1/4` is a data structure with type `fraction` and `.25` is a data structure with type `float`, so they are not equal.

```
> evalb( 1/4=.25 );
```

What about comparing `.25` and `.250`?

```
> op( .25 );
> op( .250 );
```

Notice that the two data structures do not contain the same data items, so they are not the same data structure. Not too surprisingly though, the `evalb` command says that they are equal.

```
> evalb( .25=.250 );
```

Here is an interesting way to see how the numbers **1/4**, **.25**, **.250**, and **.2500** are related in Maple. Recall that the **evalf** command is used to approximate an exact number with a decimal number and the **evalf** command can take a second input that specifies the number of decimal digits to use in the approximation. Here are three different decimal approximations of **1/4**.

```
> a := evalf( 1/4, 2 );
> b := evalf( 1/4, 3 );
> c := evalf( 1/4, 4 );
```

The previous three and the next three commands show that **.25**, **.250**, and **.2500** are considered by Maple as three different approximations of **1/4** using different numbers of decimal digits.

```
> op( a );
> op( b );
> op( c );
```

The last example and the next one show that the number of decimal digits in a floating point number is just the number of digits in the integer that is the mantissa part of the float data structure. The next four commands compute four different decimal approximations, using different numbers of decimal digits, of an integer.

```
> a := evalf( 1451, 1 );
> b := evalf( 1451, 2 );
> c := evalf( 1451, 3 );
> d := evalf( 1451, 4 );
```

```
> op( a );
> op( b );
> op( c );
> op( d );
```

Here is another example.

```
> a := 1.;
> b := 1.0;
> c:= 1.00;
```

```
> op( a );
> op( b );
> op( c );
```

These examples show that the numbers **1.**, **1.0**, and **1.00** are three different numbers to Maple, since they are stored with different data in their data structures. Why should we make a distinction between **1.**, **1.0**, and **1.00**? In engineering and physics applications, these numbers have different meanings when they are interpreted as the results of making measurements. The number **1.** means a measurement that was not very precise while the numbers **1.0** and **1.00** represent more precise measurements. An engineer, when told that the result of a measurement was **1.**, would know that there was some error in the measurement and the actual value was somewhere between **.5** and **1.5**. Told that the result of a measurement

was **1.0**, the engineer would know that the actual value was between **0.95** and **1.05**. When told that the result of a measurement was **1.00**, the engineer would know that the actual value was between **0.995** and **1.005**.

```
> 
```

Maple can display floating point numbers in several different ways. If a decimal number is not too big or too small, Maple displays it in the usual way.

```
> evalf( Pi^Pi );
> evalf( Pi^(-Pi) );
```

If the number gets kind of big (or small) then Maple switches to scientific notation.

```
> evalf( Pi^(Pi^Pi) );
> evalf( Pi^(-Pi^Pi) );
```

If the number gets very big (or small) then Maple displays the actual float data structure.

```
> evalf( Pi^(Pi^(Pi^2)) );
> evalf( Pi^(-Pi^(Pi^2)) );
```

If the number gets really, really big (or small) then the float data structure cannot hold a representation for the number and so Maple displays an error message (but it is the same error message for too big and too small).

```
> evalf( Pi^(Pi^(Pi^Pi)) );
> evalf( Pi^(-Pi^(Pi^Pi)) );
> 
```

```
> 
```

12.2.5. Names (or symbols)

The names that we use for assigned and unassigned variables are themselves simple data structures with the data type **symbol**. Here are some examples.

```
> whattype( x );
> whattype( xyz );
```

Here is a name that needs left quotes.

```
> whattype( `hello there` );
> 
```

The only piece of data in a data structure of type **symbol** is the name itself.

```
> op( xyz );
> nops( xyz );
```

Notice that the individual letters in the name are not considered the data items. It is the whole name that is the (single) data item.

```
> op( `hello there` );
> nops( `hello there` );
```

Notice that the words in the name are not themselves data items. Again, it is the whole name which is the data item.

Here is an example that shows how evaluation rules can help explain what might otherwise be pretty confusing results. First, define a list and a function.

```
> f := [1,2,3];   # A list.
> g := x -> x^2;  # A function.
```

In the next command, **whattype** uses full evaluation of **f**.

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

In the next command **g** points to a function so **whattype** uses last name evaluation.

```
> whattype( g );
```

Here is the way to see what it is that **whattype** is checking.

```
> 'whattype'( f );
> 'whattype'( g );
```

Last name evaluation can sometimes make a name act as if it were unassigned. Here is how we find out the type of what **g** points to. We need to force the full evaluation of the name **g**.

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

As we will see in a later worksheet, functions are data structures with the data type **procedure**.

```
>
```

Exercise: Explain the results of the following four **whattype** commands.

```
> abc := 123;
> whattype( abc );
> whattype( 'abc' );
> whattype( `abc` );
> whattype( `123` );
>
```

Exercise: Explain the result of the second **whattype** command.

```
> whattype( 0 );
> whattype( % );
```

Hint: What does **%** refer to in the last command?

```
>
```

```
>
```

12.2.6. Strings

Strings are a data structure made up of characters, just like the symbol data structure described in the last subsection. However, strings are defined in Maple using the double quote key rather than the left quote key used for symbols. Here are some examples of strings.

```
> "xyz";
> "a string can contain any characters, like these; &, @ ! ***
  =?";
> "0123456789";
> "There is not a lot that you can do with strings.";
```

Strings and symbols may seem more similar than they really are. The symbol `hello world` and the string "hello world" are not considered equal by Maple.

```
> evalb( `hello world`="hello world" );
```

They are two data structures with different data types.

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

Both data structures contain only one piece of data.

```
> nops( `hello world` );
> nops( "hello world" );
```

But notice in the next two commands that Maple does not quite consider them as containing the same thing. Also, notice the subtle difference in the fonts used in the next two outputs.

```
> op( `hello world` );
> op( "hello world" );
```

The convert command can be used to convert a name into a string and a string into a name.

```
> convert( `hello world`, string );
> convert( "hello world", name );
```

We just mentioned that a string data structure contains only one piece of data, and gave as evidence the nops command. But in fact, it is not really clear just what Maple considers as the data in a string data structure. For example, let s be a name for the string "hello world".

```
> s := "hello world";
> whattype( s );
```

The nops command implies that there is only one piece of data in s.

```
> nops( s );
```

But consider the following commands.

```
> s[1];
> s[2];
> s[3];
> s[11];
> s[-1];
> s[-2];
```

These commands would seem to imply, by analogy with the index notation for the list and set data types, that s contains 11 pieces of data, i.e., each of the letters in the string. The length command also backs up this claim.

```
> length( s );
```

Whether or not a string has one or many data items in it turns out not to be important. What is more important is that we can use the index notation to access individual letters from a string, which is something that we cannot do with the letters in a name.

```
> "hello world"[3..-3];
> `hello world`[3..-3];
```

The most important difference between strings and symbols is that strings cannot be used as variable names. Trying to use a string as a variable is an error.

```
> "hello world" := 0;
```

Of course the following is allowed.

```
> `hello world` := 0;
> 
```

Exercise: Why do you think that Maple does not allow index notation with names to refer to the individual letters of the name as it does with strings?

```
> 
```

So what are strings used for? Mostly as labels for graphs and as messages to be printed out by the **print** command (which we will make use of in later worksheets). Also, Maple's error messages are strings.

```
> plot( x^2, x=-10..10, title="A graph of x squared.",
>      titlefont=[COURIER, BOLDOBLIQUE, 14] );
> print( "This is a message from Maple." );
> plot();  # Cause an error message.
> 
```

It is worth noting that in versions of Maple before Maple V Release 5 there was no distinction made between symbols and strings. In fact, the string data type did not exist and there was only the symbol data type. So symbols were used for two kinds of purposes, as variable names and as "strings", i.e., labels for graphs and messages to be printed out (e.g., error messages). It was finally decided that it was not very elegant to blur the distinction between these two uses of strings of characters, so the string data type was defined. The double quote key was chosen to define strings since it is commonly used for this purpose in other programming languages. But the double quote key had been the last result variable! So in Maple V Release 5 the last result variable had to be redefined to be the percent key. If you should ever use an older version of Maple, it is useful to remember this change in the Maple language.

```
> 
```

```
> 
```

12.2.7. Equations and inequalities

Let us restart Maple, to avoid any confusion from previous assignments.

```
> restart;
```

Earlier we defined an equation as an equals sign with an expression on either side of it. It turns out that Maple considers an equation to be a data structure with data type **equation**. An equation data structure always has two operands, the left and right hand sides of the equation. Here are some examples.

```
> whattype( y = x^2 );
> op( y = x^2 );
> op( 1, y = x^2 );
> op( 2, y = x^2 );
```
Maple has special commands for accessing the operands of an equation data structure, **lhs** and **rhs**.
```
> lhs( y = x^2 );
> rhs( y = x^2 );
```

Now we can be very specific about the meanings of the two kinds of "equals signs" in a Maple command like the following.
```
> eqn := y = x^2;
```
This command gave a name to a data structure. The name given to the data structure is **eqn**, and the data structure being named is the equation data structure **y=x^2**. As far as Maple is concerned, the following two commands are doing the same kind of thing, they are giving names to data structures.
```
> ds1 := 1 = x^2 + y^2;
> ds2 := [1, `=` , x, `^`, 2, `+`,  y, `^`, 2];
> whattype( ds1 );
> whattype( ds2 );
```
(What are all the left-quotes in **ds2** for? Describe in detail the contents of **ds2**.)
```
>
```

There are three other data structures in Maple that are closely related to the equation data structure. These are the **inequality data structures**, **<>**, **<=**, **<**. Below are some examples of inequality data structures.
```
> x <> y;
> whattype( x<>y );
> op( x<>y );

> u+v <= 5;
> whattype( u+v<=5 );
> op( u+v<=5 );

> 5 < 0;
> whattype( 5<0 );
> op( 5<0 );
```

Should there be a **>=** and a **>** data structure? The answer is no, because they are not needed. Maple automatically converts any "greater than" inequality into a "less than" data structure.
```
> 5 > 0;
> whattype( 5>0 );
```

```
> op( 5>0 );
```
Notice how the order of the data items in the data structure are not what you would expect.
```
> op( 1, 5>0 );
> op( 2, 5>0 );
```
Unfortunately, this can get to be a bit confusing.
```
> lhs( 5>0 );
> rhs( 5>0 );
```
Here is another example.
```
> x-y >= 0;
> whattype( x-y>=0 );
> op( x-y>=0 );
>
```

Equations and inequalities in Maple are not entirely equivalent to equations and inequalities in standard mathematical notation. For example, the mathematical equation $x = y = z$ means that x, y, and z all have the same value. But if we translate this directly into Maple, we get an error message.
```
> x = y = z;
```
Equations and inequalities in Maple are not transitive. So we need to put parentheses into the last equation in order for Maple to accept it. But neither
```
> (x = y) = z;
```
nor
```
> x = (y = z);
```
means the same thing in Maple that $x = y = z$ means in mathematics (we will see why shortly). Similarly, we would want the following inequality to mean that x is less than y *and* y is less than z. But instead it gives us an error message.
```
> x <= y <= z;
```
The following two inequalities do not cause an error, but neither one has the meaning that we want either.
```
> x <= (y <= z);
> (x <= y) <= z;
```
We will see later what is the proper way to fix `x<=y<=z`.
```
>
```

We have said before that equations (and also inequalities) are used to ask questions. An equation or inequality data structure can be interpreted as being either true or false. The Maple command **`evalb`** allows us to evaluate an equation or inequality data structure to see if it is true of false. Here are some simple examples.
```
> evalb( 0>=5 );
```
Here are two true inequalities.
```
> evalb( 0<1 );
> evalb( 0<>1 );
```

Here is an equation.

```
> x := 5:
> y := 1:
> evalb( x=5*y );
```

What do you think should be the result of the next command?

```
> evalb( (x=5*y)=(2*x-5=5*y) );
```

Let us go over this result in detail. Each of `x=5*y` and `2*x-5=5*y` is an equation data structure, and so `(x=5*y)=(2*x-5=5*y)` is an equation data structure where each of the left and right hand sides are themselves equations. It might seem that we are asking if these are the same two equations. But they are definitely not the same equation, so that is not why the last command returned true. The last equation is true because Maple uses full evaluation before the `evalb` command is evaluated. Here is what the `evalb` command sees as its input just before it is evaluated.

```
> 'evalb'( (x=5*y)=(2*x-5=5*y) );
```

If we delay the evaluation of the inner equations, then we can verify that they are not the same equation.

```
> evalb( ('x=5*y')=('2*x-5=5*y') );
>
```

As we mentioned earlier, the following equation is syntactically incorrect.

```
> evalb( 1=1=1 );
```

Putting in parentheses corrects the syntax but does not give us what we really intended. Why do you thing the following two equations are false?

```
> evalb( (1=1)=1 );
> evalb( 1=(1=1) );
```

Hint: Which equal sign is `evalb` evaluating? What is on either side of it?

Mathematically, the inequality 0<1<2 is true. But `0<1<2` is not correct Maple syntax so we cannot even ask if it is true.

```
> evalb( 0<1<2 );
```

If we put parentheses in 0<1<2, then we can correct the syntax.

```
> (0<1) < 2;
> 0 < (1<2);
```

But if we put one of these in the `evalb` command to see if it might be true, we get one of Maple's most inscrutable error messages.

```
> evalb( (0<1)<2 );
```

Just what sum is it that this message is referring to? Here is a hint.

```
> a:='a': b:='b':
> evalb( a<b );
```

It seems that in the process of trying to determine if an inequality is true or not, Maple at some point converts the inequality into one involving zero. In the case of `a<b`, since `a` and `b` are unassigned, Maple cannot determine if this inequality is true or false so it just returns the

inequality, but in the form `a-b<0`. (Maple could have chosen to use alphabetical ordering of names to evaluate this inequality, but it did not.) Now back to `(0<1)<2`. Maple probably converted this to `(0<1)-2<0`.

```
> (0<1)-2<0;
```

Now we see what sum Maple means and where it came from.

```
> (0<1)-2;
```

```
>
```

So how does one express in Maple the fact that 0<1<2. The answer comes from reading this aloud, "zero is less than 1 and 1 is less than 2". In Maple we translate this as `0<1 and 1<2`.

```
> evalb( 0<1 and 1<2 );
```

Similarly, we would express the mathematical notation $x = y = z$ as `x=y and y=z`.

```
> x := 3: y := x: z:= y:
> evalb( x=y and y=z );
```

When we evaluate an equation (or inequality) data structure as either true or false, we are evaluating the equation (or inequality) as a **boolean expression**. The command `evalb` is an abbreviation for "`eval`uate `b`oolean". In general, a boolean expression is any expression whose value is either true or false (as opposed to, say, a number). Any two boolean expressions can be combined using `and` or `or`. For example $1 < |x|$ is equivalent to `x<-1 or x>1`. We will examine boolean expressions in more detail later in this worksheet and also in the worksheet about Maple's control statements.

```
>
```

While we are discussing inequalities and the `evalb` command, consider the following command.

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

Maple has no problem with the next two commands.

```
> evalb( 3.14<4 );
> evalb( evalf(Pi)<4 );
```

I have no idea why Maple cannot make the comparison in the inequality `Pi<4`.

```
>
```

Finally, notice that the equation data structure is a lot like an ordered pair. In fact, Maple often uses equation data structures as a way to hold ordered pairs. One example of this that we will see later is the way Maple stores entries in a table data structure (in particular, the remember table stored in a procedure data structure).

```
>
```

```
>
```

12.2.8. Ranges

We use ranges in many different situations in Maple. Here are few examples.

```
> x.(0..4);
> seq( x^2, x=0..4 );
> plot( x->x^2, 0..4 ):
```
In this subsection we show that the "dot dot" operator (`..`) is really used by Maple to define a data structure, with the data type `range`, that holds exactly two pieces of data (somewhat like the equation and inequality data structures). As we will see, a `range` data structure is often interpreted in Maple to mean some kind of range of numbers, but this interpretation is not universal, and the two pieces of data in a range data structure can actually be used for almost any purpose.
```
>
```

Here is a simple example of a range data structure.
```
> p..p^2;
```
We can give it a name.
```
> r := %;
```
We can ask what its data type is and what are its pieces of data.
```
> whattype( r );
> op( r );
```
We can try to use it.
```
> seq( i, i=r );
> w.(r);
```
Since the data items in our range do not yet evaluate to numbers, the range has limited use right now. We can give `p` a value, and then try using the range again.
```
> p := 2;
> w.(r);
> seq( i, i=r );
```
Here is what the `seq` command saw as its input before it was evaluated but after `r` and its two data items were evaluated.
```
> 'seq'( i, i=r );
```
A range data structure can hold any two pieces of data, but if the data items of the range do not evaluate to numbers, then the range will not work in most Maple commands.
```
>
```

Notice that two levels of evaluation are needed before `r` evaluates to a "real" range. The next two commands try to show these levels of evaluation.
```
> eval( r, 1 );
> eval( r, 2 );
> eval( r );
```
Notice that the `eval` command did not evaluate the names inside the range when we asked for levels of evaluation. When we used `eval` with a list data structure, `eval` was able to show the levels of evaluation of the names inside the list. For some kinds of data structures the `eval` command can show the levels of evaluation inside the data structure, but for some other data

structures **eval** cannot (we saw another example of this in the last worksheet with **eval** and function calls).

```
> 
```

Here is another example of using a range that contains expressions instead of numbers. The following **seq** command causes an error since the data items in the range do not evaluate to numbers yet.

```
> x:='x':
> seq( i, i=x..x^2 );
```

Here is an interesting example of how we can give the expressions in the range a value. We can take this **seq** command and nest it inside of another **seq** command. The outer **seq** command gives values to the range in the inner **seq**.

```
> seq( seq(i, i=x..x^2), x=1..5 );
```

The last command created an expression sequence of expression sequences. It can be a bit hard to see where the sub expression sequences begin and end in the output. Here is a slight modification of the last command that creates an expression sequence of lists.

```
> seq( [seq(i, i=x..x^2)], x=1..5 );
```

Exercise: Write a single command that will generate an expression sequence of lists where each list contains the first *n* odd integers.

```
> 
```

Exercise: Write a single command that will generate a list of lists where each list contains the first *n* integers each raised to the *n*th power.

```
> 
```

Now let us look at the interpretation that Maple gives to a range data structure. The meaning of a range **a..b** need *not* be "all the numbers from **a** to **b**". The meaning of a range depends a great deal on the context in which it is used. For example in the following command

```
> seq( i, i=-2..2 );
```

the range **-2..2** means all *integers* from -2 to 2. But in the command

```
> plot( i, i=-2..2 );
```

the range **-2..2** means all *real numbers* from -2 to 2. And in the command

```
> op( -2..2, [a, b, c, d, e, f] );
```

the range **-2..2** does not have a meaning, it caused an error. On the other hand the range **2..-2** works fine in the next **op** command.

```
> op( 2..-2, [a, b, c, d, e, f] );
```

The range **2..-2** does not cause an error in the following **seq** command, but it results in no output (because the range is interpreted here as an empty range).

```
> seq( i, i=2..-2 );
```

And in the following **plot** command, **2..-2** seems to be interpreted just like **-2..2**.

```
> plot( i, i=2..-2 );
```

So the meaning of a range data structure is *very* command dependent!

```
> 
```

```
> 
```

12.2.9. Function calls

The last of Maple's basic data structures that we describe in this section is a **function call** (also referred to as an **unevaluated function call**). This is anything of the form

`name(exprseq)`

that is, a name followed by a left parenthesis followed by an expression sequence followed by a right parenthesis, and where **`name`** may be either an assigned or an unassigned variable. Here is an example of a function call.

```
> f(x, y, z);
> whattype( % );
> op( %% );
```

Notice that Maple calls the data type **`function`**, which is confusing since **`f`** (the name part of this function call) need not be the name of any Maple function. The operands of the function call data structure are the operands of the expression sequence between the parentheses. Here is another example.

```
> a_funny_name( z^2, [a,b,c], x, 2/3 );
> whattype( % );
> op( %% );
> 
```

The unevaluated function call data structure has two main interpretations in Maple depending on whether the name in the function call is an assigned or unassigned name. The first is, not surprisingly, to represent function calls in the usual sense when the name in the function call is assigned to a function definition. Here are a few examples.

```
> whattype( exp(x) );
> whattype( 'sin(Pi)' );
```

(Why were right-quotes needed in the last example? What would be the type if they were removed?)

```
> whattype( '(z -> z^2)(w)' );
> f := z -> z;
> whattype( 'f(u)' );
```

Notice the difference between the last **`whattype`** command and the next three.

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

Since **`f`** points to a function definition, by last name evaluation **`f`** evaluates to **`f`**, so **`whattype(f)`** returns **`symbol`**. If we force full evaluation of **`f`**, then **`whattype(eval(f))`** returns **`procedure`**, which is the data type of a Maple function definition that uses the arrow notation. And by the full evaluation rule and the definition of **`f`**,

f(u) evaluates to **u**, which is a name, so **whattype(f(u))** returns **symbol**. This can get a bit confusing. Remember that a function definition is stored in a data structure with data type **procedure** (that we will look at in a different worksheet) and that **function** is the data type of a function call data structure that represents just the *form* of a function call.

```
> 
```

Maple's other interpretation of a function call data structure is as a kind of tagged, or labeled, expression sequence. This is the way Maple interprets a function call in the case where the name at the front of the call is an unassigned name. The name at the front of the function call is the tag (or label) for the expression sequence inside the parentheses.

```
> my_so_far_undefined_function(12, -2, 102);
> whattype( % );
> op( %% );
```

Maple makes use of a lot of these kinds of tagged expression sequences. Here is an example of Maple returning an unevaluated function call data structure as the result of a Maple command.

```
> x := 'x':
> solve( x^5-x+1=0 );
> whattype( % );
> op( %% );
```

The information that Maple wanted to pass on as the result of the **solve** command was placed in an unevaluated function call data structure.

```
> 
```

Here is another example of how Maple can store information in an unevaluated function call. Let us make an assumption about the variable **x**.

```
> assume( x>0, x<=5 );
```

Now we ask Maple about **x**.

```
> about( x );
```

We see that Maple stores the information about the assumptions on **x** in a couple of nested unevaluated function calls, **RealRange(Open(0),5)**. Try changing the assumption on **x** and seeing how Maple stores the new assumption.

```
> 
```

Here is another example of Maple returning an unevaluated function call as the result of a command.

```
> evalf( Pi^(Pi^(Pi^2)) );
```

The number $\pi^{\left(\pi^{(\pi^2)}\right)}$ is too large for Maple to display the usual way, so it displays it as a float data structure, and it uses an unevaluated function call to display the data structure. Notice though that the type of this result is **float**, not **function** like in the last example.

```
> whattype( % );
> op( %% );
```

```
> 
```

Notice that the following two data structures, called **fc1** and **fc2**, have the same data type and contain the same data items in the same order, but they are not the same data structure. So what distinguishes them?

```
> fc1 := h(a, b, c);
> fc2 := k(a, b, c);
> whattype( fc1 );
> whattype( fc2 );
> op( fc1 );
> op( fc2 );
> evalb( fc1 = fc2 );
```

A **function** data structure has a feature that most other data structures do not have, a zero'th data item. The zero'th data item in a **function** data structure is the **name** that appears in front of the parentheses as part of the function call.

```
> op( 0, fc1 );
> op( 0, fc2 );
```

So this is how Maple distinguishes **fc1** from **fc2**. If it were not for the **function** data structure's zero'th data item, **fc1** and **fc2** would be indistinguishable. (Recall that for most data types, the " zero'th data item" as returned by the **op** command is the data type of the data structure.)

```
> 
```

Here is another example of looking at the zero'th data item in an unevaluated function call.

```
> solve( x^5-x+1=0 );
> whattype( % );
> op( 0, %% );
> op( %%% );
> 
```

Maple makes extensive use of unevaluated function call data structures. Many of Maple's predefined internal data structures are unevaluated function call data structures that often contain other unevaluated function call data structures as operands. A good example of this, that we will examine in detail later, is the PLOT data structure used to describe Maple's graphs.

```
> 
```

The official definition of an unevaluated function call data structure and the official definition of the unevaluated function call data type in the Maple help pages both say it is of the form **name**(**exprseq**). But in reality, Maple has a much broader idea of what an unevaluated function call is. We have already sneaked into this section one example of this broader definition.

```
> whattype( '(z -> z^2)(w)' );
> op( 0, '(z->z^2)(w)' );
```

```
> whattype( % );
```
So we just saw a **function** data structure of the form **procedure**(**exprseq**) and not **name**(**exprseq**). Here is another example.
```
> whattype( '5(-2)' );
```
The next two commands also verify that Maple interprets **5(-2)** as a function call data structure.
```
> type( '5(-2)', function );
> op( 0, '5(-2)' );
```
5 is not a name.
```
> whattype( % );
```
Notice what **5(-2)** evaluates to.
```
> 5(-2);
>
```

Exercise: What function does the function call **5(-2)** refer to?
```
>
```

Here is another example of a function call that is not of the form **name**(**exprseq**).
```
> f(x)(y,z);
```
Let us check that this really is an unevaluated function call.
```
> whattype( f(x)(y,z) );
```
What are the operands?
```
> op( f(x)(y,z) );
> op( 0, f(x)(y,z) );
```
So **f(x)(y,z)** is a function call of the form **function**(**exprseq**). In particular, the name of the function being called is **f(x)**. Notice that in **f(x)(y,z)**, the **f(x)** part represents both the value of one function call and the function of another function call. To put it another way, for the expression **f(x)(y,z)** to make sense, it must be that **f** represents a function whose value is another function (in particular, a function of two variables). In an optional section of the worksheet about functions we looked at examples of functions that return functions. Here is an example that works with the expression **f(x)(y,z)**.
```
> f := x -> (u,v)->u^x+v^x;
```
Notice that **f** is *not* a function of three variables. It is a function of one variable that returns a function of two variables. Here is what **f(x)** evaluates to.
```
> f(x);
```
Here is what **f(4)** looks like.
```
> f(4);
```
Notice how the value of **f(4)** is a function of two variables. Here is what **f(x)(y,z)** evaluates to.
```
> f(x)(y,z);
>
```

Exercise: What do you think Maple is doing when it evaluates the following expression?

```
> (a,b)(c,d);
>
```

Exercise: Let us define an expression named **f**.

```
> x:='x':
> f := x^2+x+3;
```

Recall that we evaluate an expression at a point using the **subs** (or the **eval**) command.

```
> subs( x=1, f );
```

Recall that we pointed out earlier that it is a mistake to use functional notation to try and evaluate at a point a function represented as an expression.

```
> f(1);
```

Try to make sense out this last output. What did Maple do to get it? In particular, in the next output, why are the x's displayed using two different typefaces?

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

(Here is a hint.)

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

Exercise: Suppose that you mean to enter the expression $(5+w)(3-w)$ into Maple and so you type the following.

```
> w:='w':
> (5+w)(3-w);
```

You forgot the multiplication symbol *****. Explain, with as much detail as you can, how Maple parsed the input **(5+w)(3-w)**.

```
>
```

Exercise: Suppose you want to represent the mathematical function $g(x)=(x^2-3)(x^2-2)$ as an expression and then evaluate it at $x=2$. So you make the following two mistakes.

```
> x:='x':
> g := (x^2-3)(x^2-2); # Mistake 1.
> g(2);                # Mistake 2.
```

Explain the last output as best you can.

```
>
```

Exercise: Suppose you want to represent the mathematical function $g(x)=(x^2-3)(x^2-2)$ as a Maple function and then evaluate it at $x=2$. So you make the following mistake.

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

Now evaluate **g** at 2.

```
> g(2);
```

Notice how *subtle* the mistake is! Explain the last output as best you can.

```
>
```

If you start to use Maple a lot, believe me, you will make the mistakes that the last four exercises demonstrate. And these can be *hard* mistakes to track down in long, complicated calculations, especially the last one! After years of doing mathematics, we are so used to seeing `(x^2-3)(x^2-2)` as correct that it is easy to pass over it when double checking a large expression.

```
>
```

```
>
```

12.3. Data vs. data structure vs. data type

Here is an example that shows that a data structure is more than just the data contained in it, but it also includes the way the data is organized or interpreted (which is expressed as the data structure's data type). Below are seven data structures.

```
> data_structure_1 := [1,2];          # a list
> data_structure_2 := {1,2};          # a set
> data_structure_3 := 1=2;            # an equation
> data_structure_4 := 1/2;            # a fraction
> data_structure_5 := evalf(100,1); # a float
> data_structure_6 := 1..2;           # a range
> data_structure_7 := h(1,2);         # a function call
```

All seven of these data structures contain exactly the same data.

```
> op( data_structure_1 );
> op( data_structure_2 );
> op( data_structure_3 );
> op( data_structure_4 );
> op( data_structure_5 );
> op( data_structure_6 );
> op( data_structure_7 );
```

But they all have distinct data types.

```
> whattype( data_structure_1);
> whattype( data_structure_2 );
> whattype( data_structure_3 );
> whattype( data_structure_4 );
> whattype( data_structure_5 );
> whattype( data_structure_6 );
> whattype( data_structure_7 );
```

In every one of these data structures, the data in the data structure is the same and in the same order, the two integers `1` and `2`. But the fact that this data is interpreted differently is very apparent in the last four examples where `1` and `2` represent the fraction 1/2, the decimal number100 (to one decimal place), the range from 1 to 2, and the inputs in the function call h(1,2), respectively. The way to think about this is that a data structure includes not just the data, but it also includes a way of

interpreting the data, a way of giving meaning, or structure, to the data.

```
> 
```

```
> 
```

12.4. Data types in mathematics

In mathematics, the notion of a "data type" exits, but it is rarely ever mentioned. For example, in a mathematics book, the notation (1, 2) can have (at least) two distinct meanings: the point in the plane with coordinates 1 and 2, and the open interval in the real line between 1 and 2. Each of these meanings of (1,2) has a different "data type", but when you see the notation (a,b) in a math book, you cannot ask the book "what is the data type of (a,b)?" (that is, "what does (a,b) mean?"). Instead, you are supposed to realize from the context what the notation means. On the other hand, Maple is not smart enough to understand an implied context; Maple needs to always be told explicitly what we are talking about. So if we want to tell Maple about the point (1,2) in the plane, we would use a list data type `[1,2]` and if we want to tell Maple about the interval (1,2) we would use a range data type `1..2`. Notice that Maple uses syntax to distinguish different kinds of data types. Some mathematics books will also try to use syntax to distinguish between different data types. For example, some books use the notation <1,2> to mean an ordered pair, to distinguish it from the open interval (1,2). But these different syntaxes are not universally used, and even when used, the idea that one is trying to distinguish between data types is never mentioned.

```
> 
```

```
> 
```

12.5. Nested data structures

Let us restart Maple to avoid any confusion from previous assignments.

```
> restart;
```

Here is an example of a data structure where each piece of data in the data structure is itself a data structure. We will refer to data structures like these as **nested data structures**.

```
> complex_data_structure := [ [a,b,c,d..abcd], {10,20,30.0}, 1/2
  ];
> whattype( complex_data_structure );
> nops( complex_data_structure );
> op( complex_data_structure );
```

The data structure has three pieces to it. What is the data type of each individual piece?

```
> whattype( op(1, complex_data_structure) );
> whattype( op(2, complex_data_structure) );
> whattype( op(3, complex_data_structure) );
```

Each piece of the data structure is itself a data structure so the pieces of the data structure have their own pieces.

Since each piece of the original data structure is a data structure, we can apply the `op` command to

each of the individual pieces.

```
> op( op(1, complex_data_structure) );
> op( op(2, complex_data_structure) );
> op( op(3, complex_data_structure) );
```

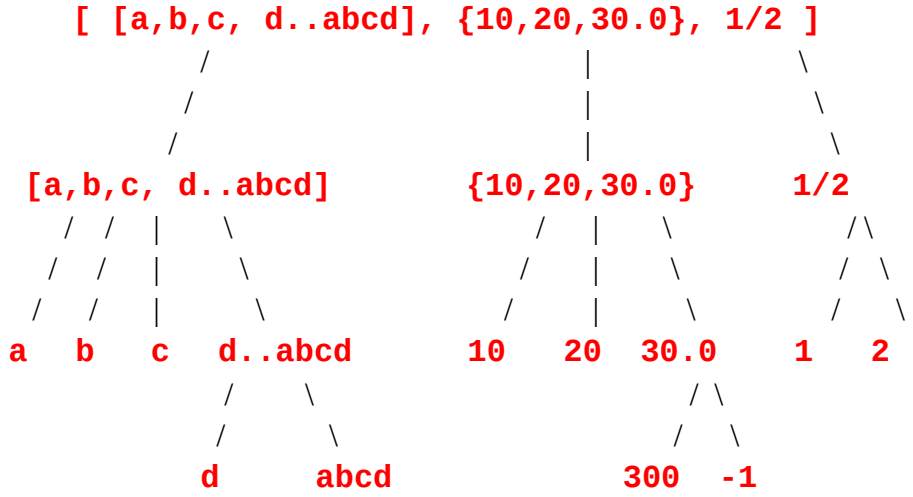
Now notice that two of the data elements listed above are themselves data structures. We can once again use the **op** command to see what the pieces are from these data structures.

```
> op(4, op(1, complex_data_structure) );
> op( op(4, op(1, complex_data_structure) ) );
> op(3, op(2, complex_data_structure) );
> op( op(3, op(2, complex_data_structure) ) );
```

Notice how we used nested **op** commands to "take apart" a nested data structure and look at all of its most elementary data elements. Nested commands like these are not easy to read, but we will be using them quite a bit as we dig deeper into how Maple works. Most of the data structures that Maple uses to do symbolic mathematics are in fact nested data structures.

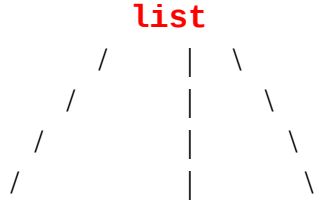
```
>
```

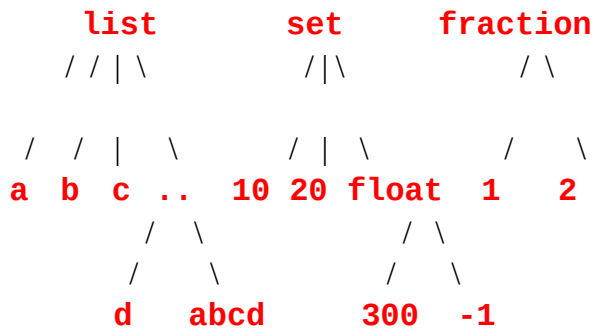
There is a geometric way to visualize how these data structures are nested within data structures. The two graphs below are called **data structure trees**. Each shows how data structures are nested within **complex_data_structure**. The first graph emphasizes the sub data structures contained in **complex_data_structure**.




```
>
```

The second graph emphasizes the data types of the sub data structures contained in each data structure. Notice how in the second graph, by working your way up from the "bottom" nodes of the graph to the top node you can reconstruct the original data structure, even though it is not explicitly shown anywhere in the graph (like it is in the first graph).





Notice how all of the "bottom" nodes of these graphs are either integers or names. These data elements are data structures that cannot be taken further apart. We sometimes refer to the data types of these data elements as **primitive data types**.

```
> 
```

Based on this geometric way of visualizing nested data structures we can define what we will call the **levels** of a nested data structure. We will say that a piece of data or a data structure is at the **n'th level of a nested data structure** if it takes *n* nested `op` commands to access the it. For example, the set `{10, 20, 30.0}` is at the first level of `complex_data_structure`, the float `30.0` is at the second level of `complex_data_structure`, and the name `abcd` is at the third level.

```
> op(2, complex_data_structure);                    # One op
  command.
> op(3, op(2, complex_data_structure) );            # Two op
  commands.
> op(2, op(4, op(1, complex_data_structure) ) );    # Three op
  commands.
```

Notice how the levels of a nested data structure are exactly the "levels" of its data structure tree. The nested data structure itself is at level zero, since you do not need an `op` command to access it, and it is the top node of the data structure tree. The pieces that are drawn just below the top node are the pieces that are at level one and you need one `op` command to access them. As you go down the tree from one node to a node below it, you need one more `op` command to access whatever is at the lower node. (Do not confuse "levels of a data structure" with "levels of evaluation". They are different ideas.)

```
> 
```

We will define the **height** of a nested data structure to be the number of levels that are in the data structure. So our example `complex_data_structure` has height 3.

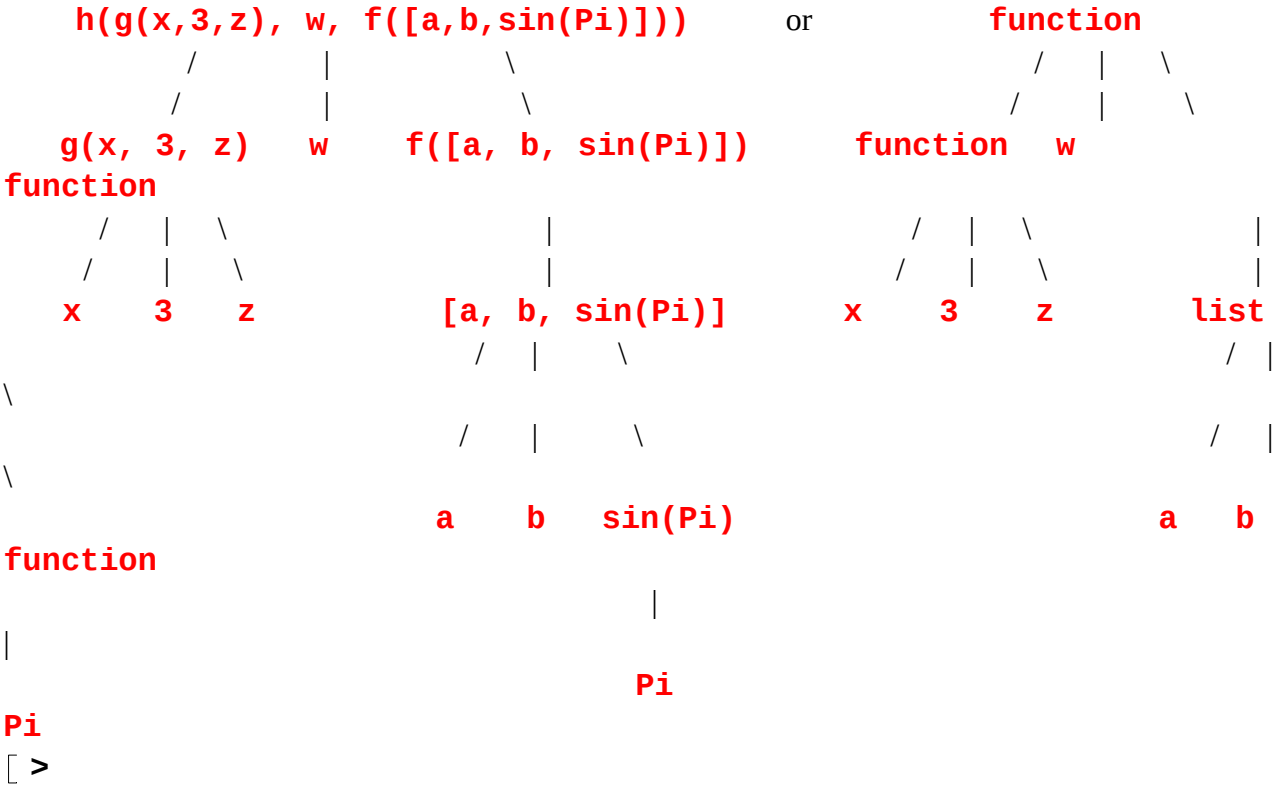
One common thing to do with complicated, nested data structures is to ask if a particular data structure or piece of data is contained somewhere in the data structure. Maple has two special commands for asking questions like this, the `member` and `has` commands. The command `has` asks if something is in a data structure at any level. The command `member` asks if something is at the first level of a data structure.

```
> has( complex_data_structure, abcd );
> member( abcd, complex_data_structure );
> member( [a,b,c, d..abcd], complex_data_structure );
```

Notice how, even though these two commands serve very similar purposes, one of them has the data structure as its first parameter and the other has the data structure as the second parameter. This makes sense if you think of the commands **has** and **member** as being used in an "infix" instead of "prefix" position. So we could read the command **has(data_structure, data_item)** as the question "**data_stucture has data_item**?". And we would read the command **member(data_item, data_structure)** as the question "is **data_item** a **member** of **data_structure**?".

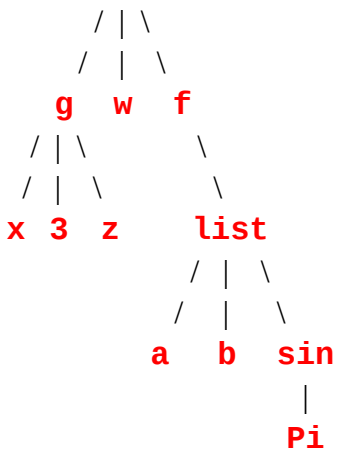
```
>
```

Here is an example of a nested data structure that contains several unevaluated function calls **h(g(x,3,z),w,f([a,b,sin(Pi)]))**. Here are data structure trees for this data structure.




```
>
```

Notice that in the second expression tree, the one that emphasizes the data types rather than the sub expressions, there is something missing. The names of the functions that are being called cannot be recovered from the tree diagram. So we will sometimes use the following version of a data structure tree. For unevaluated function call data structures, we will use the name of the function in place of the name of the data type, **function**. Recall that the name of the function is the zero'th data element in a **function** data structure, so using this notation in the data structure tree allows us to specify all of a function call's data.

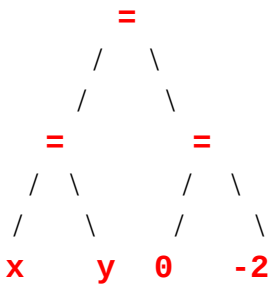



```
[ >
```

Exercise: Draw a data structure tree, similar to the one we just drew, for the data structure `f(h(23,-2),15,g(h(0,1)))`. Notice how, if you were given the definitions of the functions `f`, `g` and `h`, you would evaluate this function call by working your way up the data structure tree from the bottom to the top.

```
[ >
```

Another example of a nested data structure is `(x=y)=(0=-2)`. This is an equation data structure that contains two more equation data structures. Here is a data structure tree for this equation.




```
[ >
```

Exercise: What are the data structure trees for `x<=(y<=z)` and `(x<=y)<=z` and `(x<=y)>=z`? (Be careful about the third one.)

```
[ >
```

We mentioned earlier that levels of a data structure are not the same thing as levels of evaluation. Let us look at an example that compares these two ideas. Here is the data structure we will use `[a, [b,c], {d,e,f}]` and we will use the following assignments.

```
[ > data_structure := '[a, [b,c], {d,e,f}]';
[ > a:='b'; b:='c'; c:=-1; d:='f'; e:='f'; f:='g';
```

Now we will evaluate the data structure using different levels of evaluation.

```
[ > eval( data_structure, 1 );
[ > eval( data_structure, 2 );
```

```
> eval( data_structure, 3 );
> eval( data_structure, 4 );
```
For each level of evaluation we get a new data structure. We could draw a data structure tree for each of these data structures. Notice how a level of evaluation may or may not lead to a change in the structure of a data structure tree.
```
>
```

Exercise: Draw the data structure trees for each data structure in the last example.
```
>
```

Exercise: Start with the same expression as in the last example `[a, [b,c], {d,e,f}]`. Use the following assignments
```
> a:='b'; b:={'c','f'}; c:=-1; d:='a'; e:='f'; f:='g';
```
and draw the expression tree for each data structure that results from evaluating the original data structure to a different level.
```
> eval( [a, [b,c], {d,e,f}], 1 );
> eval( [a, [b,c], {d,e,f}], 2 );
> eval( [a, [b,c], {d,e,f}], 3 );
> eval( [a, [b,c], {d,e,f}], 4 );
>
```
Notice how a data structure tree can "grow" in some places and "shrink" in other places as the original data structure is evaluated.
```
>
```

In the next few sections we will see that a Maple expression is an example of a nested data structure. We will use data structure trees, which we will call expression trees when dealing with expressions, as a way of understanding the "structure" of an expression.
```
>
```

```
>
```

12.6. Expressions as data structures

Now that we have looked at some elementary data types and data structures, let us return to the idea of an expression. We have said several times that an expression is a data structure. So if it is a data structure, what is its data type? Let us look at an example. Define an expression in the variable `x` and give it a name (but first make sure that `x` is unassigned).
```
> x := 'x':
> f := x^2+2*x-1;
```
If it is a data structure, as we are claiming, what is its data type?
```
> whattype( f );
```
The expression is a `` `+` `` data type because it is a sum of terms. What are the pieces of data stored in this data structure?

```
> op( f );
```
The pieces of data are the terms of the expression. But each term of the expression is itself an expression, so each piece of data in the expression **f** must itself be a data structure. What are their data types?
```
> whattype( x^2 );
> whattype( 2*x );
```
The first term of **f** is a **`^`** data type because it is an exponential term, and the second term is a **`*`** data type because it is a product. What is the data type of the third term?
```
>
```

The last four commands show that an expression like **x^2+2*x-1** is a data structure (of type **`+`**) made up of three pieces of data each of which is itself a data structure (of types **`*`**, **`^`**, and **integer**). So an expression is an example of a nested data structure.

Notice we say that **x^2+2*x-1** is a data structure of type **`+`**, not **+**.
```
> type( x^2+2*x-1, `+` );
> type( x^2+2*x-1, + );
```
We need the left-quotes around the plus sign because **`+`** is the *name* for the data type and in Maple **+** cannot be a name. Similarly, **2*x** is a data structure of type **`*`**, not *****.
```
> type( 2*x, `*` );
>
```

The expression **a*x^2+b*x+c** is also a data structure to Maple. Let us see how it can be manipulated. First let us define it and give it a name.
```
> quad := a*x^2+b*x+c;
```
Check what its data type is and see how many operands it has.
```
> whattype( quad );
> nops( quad );
```
Here are the operands of the data structure.
```
> op( quad );
```
Let us look more closely at the first operand. What is its data type and what are its operands?
```
> op(1, quad);
> op( op(1, quad) );
```
Let us look at the second operand in this last output, that is, the second operand of the first operand of **quad**.
```
> whattype( op(2,op(1,quad)) );
> op( op(2,op(1,quad)) );
```
What we are doing is taking a data structure (i.e. **quad**) and tearing it apart, or breaking it down into its basic components using the **op** command.
```
>
```

We can also built up a data structure. First, let us use the **seq** command to create an expression

sequence data structure

```
> seq( x^i, i=0..7 );
```

Convert the expression sequence into a list.

```
> [%];
```

Now use the **add** command to convert the list into a `+` data structure. Examine the following command carefully and make sure you understand the meaning and purpose of every character in it.

```
> add( op(i,%), i=1..nops(%) );
```

Here is a much more compact and terse version of the last command. This version makes **i** successively represent each term from the list data structure, and adds up these terms.

```
> add( i, i=%% );
```

We have built up a data structure that represents a polynomial. Now we can use the **factor** command to further manipulate our data structure and factor it.

```
> factor( % );
```

Of course, this produced a new data structure. What is its data type and what are its operands?

```
> whattype( % );
> op( %% );
```

It is important to realize that most of Maple's commands for doing symbolic algebra are really commands for manipulating data structures.

```
>
```

Here is another example of the idea that a data structure is more than just the data contained in it. First let us make sure that **x** and **y** are unassigned.

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

Here are three different data structures (i.e., expressions), **x+y**, **x*y**, and **x^y**.

```
> whattype( x+y );
> op( x+y );
> whattype( x*y );
> op( x*y );
> whattype( x^y );
> op( x^y );
```

So **x+y**, **x*y**, and **x^y** are three different data structures, but all three contain the same data (and in the same order).

```
>
```

Let us look at another example of taking apart an expression. This time the expression is a bit more complicated and the data structure that represents the expression is not at all obvious. Here is the expression, a rational function.

```
> r := (x+1)/(x^2-1);
```

What is its data type?

```
> whattype( r );
```

It would have been reasonable to expect a `/` data type for division, but it turns out that Maple does not have such a data type. Here are the operands of **r**. This helps explain why **r** has a `*`

data type

```
> op( r );
```

Now let us examine each of the operands of `r`.

```
> whattype( op(1,r) );
> whattype( op(2,r) );
```

Notice that the second operand of `r` is of type `` `^` `` (for exponentiation). The following command shows why.

```
> op( op(2,r) );
```

Maple represents the second operand of `r` as `x^2-1` raised to the `-1` power. The following expression, `r2`, shows how Maple really thinks of the expression `r`.

```
> r2:=(x+1)*(x^2-1)^(-1);  # This is how Maple thinks of r,
```

The next command shows that Maple really does consider `r` and `r2` as the same expression.

```
> evalb( r=r2 );
```

Here is another way to write the expression `r`. It can be simplified.

```
> r3 := simplify( r );
```

However, Maple does not consider the expression `r` and its simplified version `r3` to be the same expression. They are very different as data structures.

```
> evalb( r=r3 );
```

In the next section we will draw data structure trees for `r` and `r3` and see why they are so different.

```
>
```

Exercise: Draw the data structure tree for the expression `2*x+3/x`.

```
> 2*x+3/x;
>
```

Exercise: Draw the data structure tree for the expression `2*x^2+3/x^2`.

```
> 2*x^2+3/x^2;
>
```

```
>
```

12.7. Expression trees

As an exercise, let us take the expression `r` from the last section and break it down into its most basic parts and see how they are organized together to make `r`.

```
> r := (x+1)/(x^2-1);
```

Here is what is at the top (zero) level.

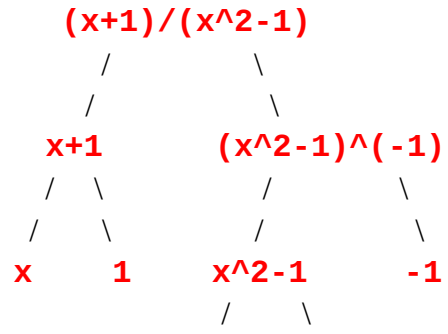
```
> r;
> whattype( r );
```

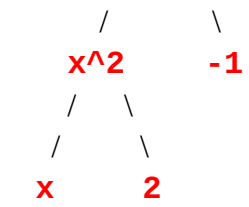
Here is what is at the first level of `r`.

```
> op( r );
> whattype( op(1, r) ), whattype( op(2, r) );
```

Here is part of the second level, the part under `op(1,r)`.

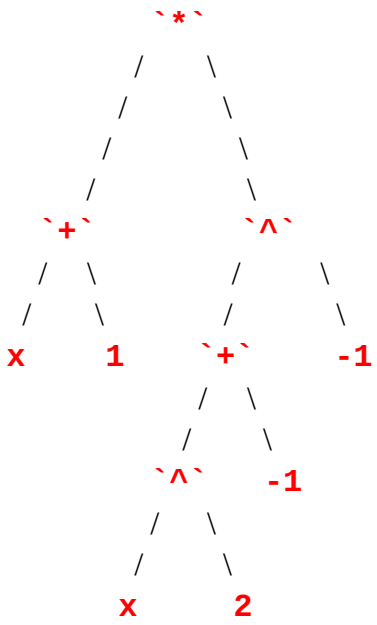
```
> op( op(1,r) );
```
Let us look at these two parts of the second level.
```
> op(1, op(1,r));
> whattype( op(1, op(1,r)) );
> op(2, op(1,r));
> whattype( op(2, op(1,r)) );
```
Now we can go back and look at the other part of the first level.
```
> op(2,r);
> whattype( op(2,r) );
```
Here is the rest of the second level, the part under **op(2,r)**.
```
> op( op(2,r) );
> whattype( op(1, op(2,r)) ), whattype( op(2, op(2,r)) );
```
Here is one part of the second level that is under **op(2,r)**.
```
> op(2, op(2,r));
> whattype( op(2, op(2,r)) );
```
Here is the other part of the second level that is under **op(2,r)**.
```
> op(1, op(2,r));
> whattype( op(1, op(2,r)) );
```
Now we can look at the third level.
```
> op( op(1, op(2,r)));
```
The third level has two pieces. Here is one piece.
```
> op(2, op(1, op(2,r)));
> whattype( op(2, op(1, op(2,r))) );
```
Here is the other piece of the third level.
```
> op(1, op(1, op(2,r)));
> whattype( op(1, op(1, op(2,r))) );
```
Finally, we can look at the two pieces of the fourth level of **r**.
```
>  op( op(1, op(1, op(2,r))) );
>
```
Let us notice a few things. First, to access something from level *n* of **r**, we needed to use *n* **op** commands. Recall that this is what we mean by the "levels of a nested data structure" and we will also use this to define the "levels of an expression". Second, what we have figured out above can be given a geometric representation as a tree diagram, and the levels of **r** correspond directly to levels in the tree diagram.





This tree diagram is called an expression tree for **r**. The levels of **r** are exactly the levels in the tree diagram. Here is another way to draw an expression tree for **r**.

```
>
```



The first form of the expression tree emphasizes the sub expressions contained in **r**. Each node of the tree is one of **r**'s sub expressions. The second form of the expression tree emphasizes the data type that each sub expression has. Each node is either the name of a data type or a "primitive" data structure.

```
>
```

Exercise: Draw the expression tree for the expression **r3** from the last section (that is, the simplified expression for **r**).

```
> 'r' = r;
> r3 := simplify( r );
>
```

In the above expression trees, notice how the type of each sub expression is exactly the arithmetic operator that is evaluated at that level. So the organization of the data structure for **r** directly reflects the rules for the precedence of arithmetic operations. The higher an arithmetic operation's precedence is, the lower it is in the expression tree. Maple would evaluate this expression by working it way up the tree, from bottom to top. Here is another example that shows this. First let us define and name another expression.

```
> q := 3*x^2+y;
```

Here is the top level of the expression.

```
> q;
> whattype( q );
```

Here are the sub expressions at the first level.

```
> op( q );
```

Here is one of the first level sub expressions.

```
> op( 1, q );
> whattype( op(1,q) );
```

Here are the sub expressions at the second level of **q**.

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

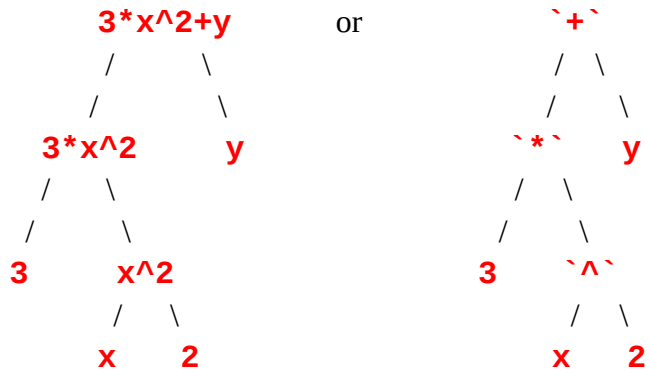
Here is one of the second level sub expressions.

```
> op( 2, op(1,q) );
> whattype( op(2, op(1,q)) );
```

Finally, here are the pieces of the third level of **q**.

```
> op( op(2, op(1,q)) );
```

The expression trees for **q** look like the following.



The exponential, which has the highest order of precedence in this expression and is computed first, is almost at the bottom. The multiplication, which is computed next, is at the middle level. And the addition, which has the lowest order of precedence in this expression, and so it is evaluated last, is at the top level. Given values for the variables **x** and **y**, Maple would evaluate this expression by working its way up the expression tree.

```
>
```

Exercise: You can change the order of the operations in an expression by inserting parentheses into the expression. Draw the expression trees for the expressions **(3*x)^2+y**, **3*x^(2+y)** and **(3*x)^(2+y)**, each of which is **q** with parentheses inserted in it to change the order of the operations. Notice how, even though the parentheses do not show up in the expression tree, the order of operations is clear from the structure of the expression tree.

```
>
```

Exercise: Draw the expression tree for **sin(x^2+x/y)-exp(3*z*f(x,Pi*y))**.

```
>
```

Exercise: This exercise compares the ideas of levels of evaluation and levels of an expression. First make the following assignments.

```
> unassign('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h');
> a := 'c': b := 'a': c := 2: d := 'f': e := 1: f := 'b':
```

Now consider the following expression `(3*a+b+h/d)^e+f`. Draw an expression tree for this expression. Then draw an expression tree for the expression that results when we evaluate `(3*a+b+h/d)^e+f` to one level (given by the next command).

```
> eval( (3*a+b+h/d)^e+f, 1 );
```

Now draw an expression tree for `(3*a+b+h/d)^e+f` evaluated to two levels.

```
> eval( (3*a+b+h/d)^e+f, 2 );
```

Draw an expression tree for `(3*a+b+h/d)^e+f` evaluated to three levels.

```
> eval( (3*a+b+h/d)^e+f, 3 );
```

Draw an expression tree for `(3*a+b+h/d)^e+f` evaluated to four levels.

```
> eval( (3*a+b+h/d)^e+f, 4 );
```

Draw an expression tree for `(3*a+b+h/d)^e+f` evaluated to five levels.

```
> eval( (3*a+b+h/d)^e+f, 5 );
```

The expression `(3*a+b+h/d)^e+f` evaluated to six levels is fully evaluated.

```
> eval( (3*a+b+h/d)^e+f, 6 );
```

Notice how, with each level of evaluation, Maple tries to compute as much of the expression tree as it can. As the expression is evaluated to more levels, more branches of the expression tree can be "pruned off" starting from the leaf nodes of the tree and working up the tree.

```
>
```

Exercise: Is it possible that a level of evaluation can cause an expression tree to "grow" a branch? To put it another way, can an expression tree become more complicated as we increase the levels of evaluation?

```
>
```

```
>
```

12.8. Why are expression trees important?

Here is an example that seems puzzling at first. Let us define an expression.

```
> w := x+2+sin(x+2);
```

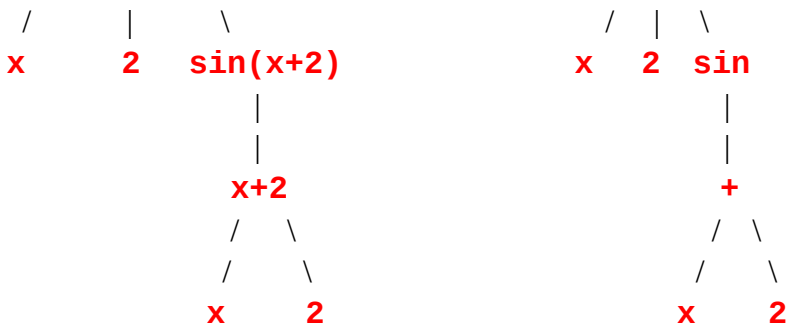
Now let us try and do a simple substitution, something that we might do in calculus. We will substitute `u` for `x+2` in the expression `w`.

```
> subs( x+2=u, w );
```

Why did that not work? Only one of the `x+2` terms was replaced in the substitution. The answer comes from looking at the expression trees for `w`.

```
>
```

```
      x+2+sin(x+2)          or          +
     /     |     \                    / | \
```



The explanation of why this is the correct expression tree comes from the following commands. Notice that at the top level, **w** is a sum of three terms, and the third term is a function call that contains only one operand, the sub expression **x+2**.

```
> op( w );
> op(3,w);
> op( op(3,w) );
```

Now we can use the expression tree to explain why the **subs** command did not do what we expected it to do. The **subs** command can only replace sub expressions when it does a substitution. The expression **x+2** only appears once in **w** as a sub expression (see the first expression tree). So the **subs** command only did one substitution! In the online help for the **subs** command, this kind of substitution is called **syntactic substitution**.

You might think that adding a pair of parentheses to the expression **w** would help with the substitution but it does not. Let **w1** be **w** with an extra pair of parentheses.

```
> w1 := (x+2)+sin(x+2);
```

Notice that Maple just dropped the parentheses from **w1**. And the following command shows that **w1** is exactly the same expression (i.e., has the same expression tree) as **w**.

```
> op( w1 );
```

How can we get around this kind of limitation in the **subs** command? Here is one way, and then we will look at another way which uses a different substitution command. The first way is to use a trick. Instead of substituting **u** for **x+2**, we shall substitute **u-2** for **x**.

```
> subs( x=u-2, w );
```

That worked.

Here is another substitution command that can also be used.

```
> algsubs( x+2=u, w );
```

The **algsubs** command worked the way we originally thought the **subs** command would. This substitution command does what the online help calls **algebraic substitution** (and hence the name of the command). Algebraic substitutions are more general than the kind of substitutions that are used in calculus. Consider the following example. Suppose we want to replace x^2 in the expression x^3 with u, but in an "algebraic way" rather than in the usual calculus way.

```
> algsubs( x^2=u, x^3 );
```

Notice that the following **subs** command does not do anything. (What is the expression tree for **x^3**?)

```
> subs( x^2=u, x^3 );
>
```

Exercise: In calculus, what would be the result of letting u be x^2 in the expression x^3? Try to find a way of doing this substitution with Maple. (Hint: Use the **subs** or the **algsubs** command and use a trick similar to the one we used above with **subs**.)

```
>
```

Exercise: Find two ways to get Maple to substitute **w** for **x^2/2** in the expression **3*x^2/2*exp(5*x^2/2)**.

```
>
```

Exercise: The following substitution is perfectly legitimate in Maple even though at first glance the **2=3** may not seem to make sense.

```
> subs( 2=3, x^2+sin(2*Pi*x) );
```

Explain this last substitution in terms of the expression tree for **x^2+sin(2*Pi*x)**. Now consider the following substitution.

```
> subs( -1=1, 1/x-x );
```

Explain this last substitution using the expression tree for **1/x-x**.

```
>
```

In this section we saw the need for expression trees to help us understand the **subs** command when it does its form of syntactic substitution. This is just one of many ways that understanding expression trees is important for working with Maple.

Exercise: Read the help page about the **subsop** command. Compare and contrast the **subs** and **subsop** commands.

```
>
```

Note: If we try to explain the following substitution in terms of an expression tree, we run into problems.

```
> subs( 1=2, x+y+2 );
```

In the way that we have drawn expression trees, there would be no term 1 anywhere in the expression tree for **x+y+2**, so it is not clear how Maple made the substitution. The answer is that the way we have drawn expression trees is not exactly how Maple represents expressions internally. Maple's internal representation for an expression is really quite a bit more complicated than an expression tree. Maple uses something called a **directed acyclic graph**. If you want to know more about this, see *Introduction to Maple*, 2nd Ed., by Andre Heck, Chapter 6.

```
>
```

```
> 
```

12.9. Some other basic data types (optional)

According to the online help for the `whattype` command, Maple has 29 **basic data types**. So far we have discussed 17 of these. They are `` `*` ``, `` `+` ``, `` `..` ``, `` `<` ``, `` `<=` ``, `` `<>` ``, `` `=` ``, `` `^` ``, `expseq`, `float`, `fraction`, `function`, `integer`, `list`, `set`, `string`, and `symbol`. In this section we discuss eight of the remaining basic data types. They are the three logical data types `` `and` ``, `` `or` ``, and `` `not` ``, unevaluated dotted and indexed names, `series`, `uneval`, and `` `::` ``. In the next section we discuss two more of the basic data types, `table` and `array`. Of the remaining two basic data types, we will discuss `procedure` in the worksheets about procedures, and we will not discuss the `hfarray` data type.

```
> 
```

12.9.1. Logical data types

Maple has three **logical data types**, `` `and` ``, `` `or` ``, and `` `not` ``. These data types and their data structures are used to build up boolean expressions. A boolean expression is analogous to an algebraic expression but a boolean expression evaluates to either of the logical values `true` or `false` (instead of an algebraic value, like a number). Just as the `` `+` `` and `` `*` `` data structures are used to represent the operations of addition and multiplication in an algebraic expression, the `` `and` ``, `` `or` ``, and `` `not` `` data structures are used to represent the logical operations of **conjunction**, **disjunction**, and **negation** in a boolean expression. Here is an example of a boolean expression.

```
> not( a or b or c) and d;
```

Let us analyze this expression much as we would analyze an algebraic expression. First, let us give our anonymous boolean expression a name.

```
> bexp := %;
```

What type of data structure is it?

```
> whattype( bexp );
```

How many operands are there?

```
> nops( bexp );
```

What are the operands?

```
> op( bexp );
```

Check the data type of each operand.

```
> whattype( op(2, bexp) );
> whattype( op(1, bexp) );
```

Let us analyze the first operand of `bexp`.

```
> nops( op(1, bexp) );
> op( op(1, bexp) );
> whattype( op(1, op(1,bexp) ) );
> nops( op(1, op(1,bexp) ) );
```

The last result may be a bit surprising. The next command explains why the result was 2 (and not 3).

```
> op( op(1, op(1,bexp) ) );
```
Notice, interestingly, how the boolean expression `a or b or c` is stored in a very different way from an algebraic expression like `a + b + c`.
```
> nops( a or b or c );
> nops( a + b + c );
>
```

Exercise: Use the information from the previous commands to draw an expression tree for the boolean expression `bexp`.
```
>
```

We should briefly mention the difference between `and`, `or`, `not` and `` `and` ``, `` `or` ``, `` `not` ``. Each of `and`, `or`, and `not` is a Maple keyword. These three keywords represent in Maple the three logical operations of conjunction, disjunction, and negation. On the other hand, `` `and` ``, `` `or` ``, and `` `not` `` are Maple names. These are the names given in Maple to the three logical data types. Here is an example of the distinction between `and` and `` `and` ``. The following command asks if the expression `a and b` is an `` `and` `` data structure.
```
> type( a and b, `and`);
```
Notice that each of the following commands is a syntax error.
```
> type( a and b, and);
> type( a `and` b, `and`);
```
Notice how, in the first of the above two error messages, the placing of left-quotes around the keyword `and` is a bit misleading (since a keyword is not a name). Notice also that when Maple returns the name of a data type, it does not use left-quotes.
```
> whattype( a and b );
>
```

We will make quite a bit of use of boolean expression when we get to the worksheets about Maple programming, in particular when we discuss conditional statements and while-loops. We will explain a more about boolean expressions in those worksheets.
```
>
```

```
>
```

12.9.2. Dotted names

An unevaluated dotted name is another data structure that holds an ordered pair of data items, much like equations and ranges. For example, here is an unevaluated dotted name data structure.
```
> whattype( 'x.i' );
> nops( 'x.i' );
> op( 'x.i' );
```
The data structure holds the two items that are on either side of the dot.

If we let Maple evaluate the dotted name, then it becomes a name and is no longer a data

structure of type `.`. Here is an example of an evaluated dotted name.

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

Unlike the range and equation data structures, an unevaluated dotted name data structure cannot hold two arbitrary pieces of data. The piece of data on the left side of the dot must be a name. If we try to put anything other than a name on the left side of the dot, we get a syntax error.

```
> 27.i;
> (x+y).i;
> (x).i;
```

On the right hand side of the dot, we can put pretty much any data item. But Maple can evaluate a dotted name only if the item on the right side of the dot is a name, an integer, or a range. For example, here is an unevaluated dotted name data structure that cannot be evaluated to a name (so we do not need left quotes to delay its evaluation).

```
> whattype( x.(i^j) );
> nops( x.(i^j) );
> op( x.(i^j) );
```

Here is an example of an evaluated dotted name with a range to the right of the dot.

```
> x.(-3..3);
```

Notice that this evaluated to an expression sequence of names. Here is the unevaluated dotted name data structure that Maple just evaluated..

```
> whattype( 'x.(-3..3)' );
> nops( 'x.(-3..3)' );
> op( 'x.(-3..3)' );
>
```

Since an unevaluated dotted name data structure can evaluate to a name, we can put one on the right of the dot of another dotted name. In other words, we can nest unevaluated dotted name data structures. Here is an example of a nested, unevaluated dotted name data structure.

```
> whattype( 'x.i.j' );
> nops( 'x.i.j' );
> op( 'x.i.j' );
> op(1, 'x.i.j' );
> whattype( op(1, 'x.i.j' ) );
> op(2, 'x.i.j' );
> whattype( op(2, 'x.i.j' ) );
```

Notice how the above commands show that the dot operator is left associative. If we remove the left quotes, Maple has no trouble evaluating `x.i.j`, since there is a name on both sides of both dots.

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

Here is a slight variation on the previous example. In this example we use a pair of parentheses to change the order of evaluation of the dot operators. Notice how the parentheses in the expression change the structure of the underlying data structure.

```
> whattype( 'x.(i.j)' );
> nops( 'x.(i.j)' );
> op( 'x.(i.j)' );
> op(1, 'x.(i.j)' );
> whattype( op(1, 'x.(i.j)' ) );
> op(2, 'x.(i.j)' );
> whattype( op(2, 'x.(i.j)' ) );
>
```

As a final note, notice that even though parentheses are allowed in the expression **x.(i.j)** to change the order of the evaluation of the dots, for some reason parentheses are not allowed in an expression like **(x.i).j**, which, since the dot operator is left associative, should be equivalent to **x.i.j**.

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

```
>
```

12.9.3. Indexed names

An unevaluated indexed name data structure is very much like an unevaluated function call data structure. Here is an example of an unevaluated indexed name data structure.

```
> f:='f':
> whattype( f[x,y] );
> nops( f[x,y] );
> op( f[x,y] );
```

Recall that Maple has a special way of typesetting (or pretty printing) an indexed name as a subscripted name. Remember that typesetting an indexed name is *not* the same as evaluating it. The following indexed name is unevaluated since it is still an unassigned name.

```
> f[x,y];
```

Like an unevaluated function call data structure, an unevaluated indexed name data structure has a zero'th data item, which is the **header** for the indexed name.

```
> op( 0, f[x] );
```

Like an unevaluated function call data structure, an unevaluated indexed name can hold an arbitrary number of arbitrary data items. Here is a more complicated example of an unevaluated indexed name data structure.

```
> whattype( f[x^2, 12, a-b, u=v^2] );
> nops( f[x^2, 12, a-b, u=v^2] );
> op( f[x^2, 12, a-b, u=v^2] );
```

If we let Maple typeset this indexed name, then we get a very strange looking name. Again, remember that typesetting an indexed name is *not* the same thing as evaluating it. The following indexed name is still unevaluated.

```
> f[x^2, 12, a-b, u=v^2];
>
```

Even the header for an indexed name can be pretty complicated. If we want the header to be something other than a name (or a list or a set), we need to enclose it on a pair of parentheses.

```
> (r^t=z^3)[x^2, 12, a-b, u=v^2];
> whattype( (r^t=z^3)[x^2, 12, a-b, u=v^2] );
> nops( (r^t=z^3)[x^2, 12, a-b, u=v^2] );
> op( (r^t=z^3)[x^2, 12, a-b, u=v^2] );
> op( 0, (r^t=z^3)[x^2, 12, a-b, u=v^2] );
>
```

Exercise: Say as much as you can about the following expression.

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

Here is an example of an evaluated indexed name. The header is the list `[a,b,c,d]`.

```
> [a,b,c,d][2..3];
```

Here is the unevaluated indexed name that Maple just evaluated.

```
> '[a,b,c,d][2..3]';
> whattype( '[a,b,c,d][2..3]' );
> op( 0, '[a,b,c,d][2..3]' );
> op( '[a,b,c,d][2..3]' );
>
```

We can nest unevaluated indexed name data structures inside of each other. The next three indexed names show three different ways of nesting unevaluated indexed name data structures. The first has an indexed name as a regular operand of an indexed name data structure. The second has an indexed name as the zero'th operand of an indexed name data structure. And the third has indexed names as both a regular and the zero'th operand of an indexed name data structure.

```
> f[g[0]];
> f[g][0];
> f[g][g[0]];
```

Notice how much the first two look alike when they are typeset. It is very difficult to tell them apart, though if you look very closely you can see a slight difference in the spacing of the letters. The following pairs of commands show how each of these three data structures is put together, which makes it easy to see that they really are different.

```
> op( 0, f[g[0]] );
> op( f[g[0]] );
> op( 0, f[g][0] );
> op( f[g][0] );
> op( 0, f[g][g[0]] );
> op( f[g][g[0]] );
>
```

Exercise: Say as much as you can about the following expression.

```
> [a,b,c,d][2..3][-1];
>
```

Exercise: Describe the structure of the following expressions.

```
> f[x](y,z);
> f(x)[y,z];
> f[x][y,z];
> f[x]([y,z]);
> (f[x])[y,z];
>
```

```
>
```

12.9.4. Series

An infinite series would obviously be a difficult object to store in a computer. But infinite series are very common mathematical objects, so Maple needs a way to represent them. Maple represents infinite series with a **series** data structure.

```
>
```

Infinite series are hard to write down completely on paper also, so mathematics has various ways of representing them abstractly. In order to distinguish between a sum of a few terms, like this

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$$

and an infinite series

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots$$

we often use an ellipsis, that is the three periods in a row (...), to mean "go on adding for ever". Maple uses a similar kind of notation when it displays a **series** data structure, but instead of using the ellipsis notation, Maple uses a "big-oh" kind of notation. Here is a command that creates a **series** data structure.

```
> series( exp(x), x=0, 5 );
```

The last term in the output, $O(x^5)$, denotes that this output represents a **series** data structure. The x^5 in the last term indicates that the series is in the variable x and that the next term of the series will have order 5. Here is another **series** data structure that represents the same infinite series but it displays more terms explicitly.

```
> series( exp(x), x=0, 10 );
```

Let us give our first (anonymous) **series** data structure a name so that we can analyze it more conveniently.

```
> exp_series := %%;
```

Let us check the data type of this data structure.

```
> whattype( exp_series );
```

Let us see how many operands the data structure contains and what the operands are.

```
> nops( exp_series );
> op( exp_series );
```

This **series** data structure represents five explicit terms from the infinite series plus the last, big-oh, term that tells us that this is a **series** data structure. Each explicit term from the series is represented in the **series** data structure by two numbers, the coefficient of the term and the exponent of the term (in that order). The last, big-oh, term is also represented by two items, the special item O(1) (which is an unevaluated function call data structure) and an integer that denotes the order of the next term in the series. The next few commands analyze the term O(1) to show that it really is just an unevaluated function call.

```
> op(11, exp_series);
> whattype( op(11, exp_series) );
> op(0, op(11, exp_series));
> op(1, op(11, exp_series));
>
```

Here is another example of a **series** data structure.

```
> series( cos(x), x=0, 4 );
```

This data structure should have six operands.

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

Notice that the **series** data structure is efficient in the sense that it does not waste any space storing terms from the series that have zero coefficient. In the last example, the first and third degree terms from the series have zero coefficient, and they are not represented in the data structure.

```
>
```

Notice that the **series** data structure seems to be missing some information. From just the data items listed by the **op** command, we cannot determine the series that the data structure represents. For example, compare the following **series** data structure and its contents with the previous example.

```
> series( sin(w), w=Pi/2, 4 );
> nops( % );
> op( %% );
```

The last two **series** data structures contain the exact same data items, though they represent different series. The information that is missing from a **series** data structure is the term that the series is in. The first of the last two examples is a series in x, and the second is a series in $w - \frac{\pi}{2}$. This information is in fact stored in the zero'th data item of the **series** data structure (recall that this is where Maple usually stores the data type of a data structure).

```
> series( cos(x), x=0, 4 );
> op(0, %);
> series( sin(w), w=Pi/2, 4 );
> op(0, %);
>
```

Maple has three commands that can create **series** data structures, **series**, **taylor**, and **numapprox[laurent]** (from the **numapprox** package). Here are some examples of series created by these commands.

```
> exp(x^2*sin(x));
> taylor( %, x=Pi, 4 );
```

The **series** data structure can represent series that are more general than the power series from calculus. The following command computes a Laurent series, which allows for negative exponents in the series.

```
> exp(x)/sin(x)^2;
> numapprox[laurent]( %, x=0, 8 );
> whattype( % );
> nops( %% );
> op( %%% );
```

The **series** command can produce some pretty unusual series. The next example is a series in the variable *x* that has coefficients that are themselves functions in *x*.

```
> series( x^x, x=0 );
> whattype( % );
> nops( %% );
> op( %%% );
>
```

Surprisingly, the result of the **series** command need not be a **series** data structure.

```
> series( 1/(1+sqrt(x)), x=0 );
> whattype( % );
> nops( %% );
> op( %%% );
```

The last **series** command produced a sum that looked like a series. The following **series** command produces a series that looks like a sum.

```
> series( 3*x^2-2*x+5, x=0 );
> whattype( % );
> nops( %% );
> op( %%% );
>
```

It is worth mentioning that Maple has a command for converting series into sums, **convert/polynom**.

```
> series( ln(x+1), x=0 );
> convert( %, polynom );
> whattype( % );
```

The result of the **convert/polynom** command need not be a polynomial however, since a series can be more general than a power series.

```
> series( ln(x+1)/x^3, x=0);
> convert( %, polynom );
> whattype( % );
> type( %%, polynom );
>
```

Note: In mathematics it is also common to study infinite products along with infinite sums. For example, here is a very famous infinite product.

$$\sin(z) = z\left(\prod_{n=1}^{\infty}\left(1 - \frac{z^2}{(n\,\pi)^2}\right)\right)$$

However, Maple does not have a special data structure for representing infinite products like it has for representing infinite sums.

```
>
```

```
>
```

12.9.5. Unevaluated expressions

Maple has a highly specialized data structure of type **uneval** for holding unevaluated expressions. Consider the following simple example. First give **x** a value.

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

In the next command, Maple fully evaluates **x** before calling **whattype**, so 5 is passed to **whattype**, which returns **integer**.

```
> whattype( x );
```

In the next command, Maple strips off one layer of delayed evaluation from **'x'** before calling **whattype**, so **x** is passed to **whattype**, which returns **symbol**.

```
> whattype( 'x' );
```

In the next command, Maple strips off one layer of delayed evaluation from **''x''** before calling **whattype**, so **'x'** is passed to **whattype**, which returns the special data type **uneval**.

```
> whattype( ''x'' );
```

In the last command, **whattype** was passed a data structure that holds an unevaluated expression. Here is a slightly more interesting example. First give **y** a value.

```
> y := 3;
```

In the next command, Maple fully evaluates **x+y** before calling **whattype** and so passes an integer to **whattype**.

```
> whattype( x+y );
```

In the next command, Maple removes one layer of delayed evaluation from **'x+y'** before

calling **whattype** and so passes to **whattype** a `+` data structure.

```
> whattype( 'x+y' );
```

In the next command, Maple removes one layer of delayed evaluation from **''x+y''** before calling **whattype** and so passes to **whattype** the unevaluated expression **'x+y'**.

```
> whattype( ''x+y'' );
```

Maple places the unevaluated expression **'x+y'** in a data structure of type **uneval**. A data structure of type **uneval** always holds exactly one data item, the expression that is unevaluated. Let us confirm this for the unevaluated expression **'x+y'**.

```
> nops( ''x+y'' );
> op( ''x+y'' );
```

Here is another way to verify that **'x+y'** is an **uneval** data structure and that it contains only one operand.

```
> op(0, ''x+y'' );
> op(1, ''x+y'' );
> op(2, ''x+y'' );
>
```

Exercise: Explain how Maple evaluates the following command and why it returns what it does.

```
> op( ''x+y'+y' );
>
```

Note: When we get to Maple procedures we will discuss the idea of procedure parameter type declarations and the "uneval" procedure parameter type. Unfortunately, the "uneval procedure parameter type" uses the term "uneval" in a way that is related to, but not exactly the same as, the "uneval data type".

```
>
```

```
>
```

12.9.6. `::`

The `::` data structure is, like the equation and range data structures, a data structure that holds exactly two operands that can be thought of as a left and a right hand side. Here is an example of a `::` data structure.

```
> w::integer;
> whattype( % );
> op( %% );
> type( %%%, `::` );
```

Unlike the equation and range data structures, the `::` data structure is not widely used. In fact it has only one very specific use. It is used to hold type declarations in the parameter lists for procedure definitions. A typical use of a `::` data structure would look something like this.

```
> proc(w::list, n::posint) op(n,w) end;
> op( 1, % );
```

```
> op( 1, %% )[1];
> whattype( % );
> op( %% );
```
We will say much more about type declarations for procedure parameters in the worksheets about Maple programming (although we will never need to make any reference to `` `::` `` data structures or the `` `::` `` data type).
```
>
```

Note: The `::` notation does have some other uses in Maple. We can use `::` as a shorthand for the **type** command. Here are a couple of examples.
```
> type( 5, integer );
> evalb( 5::integer );
> type( x+y, `+` );
> evalb( (x+y)::`+` );
> 5::integer and (x+y)::`+`;
```
The second bullet item of the help page for the **type** command mentions that **x::t** can be used as a synonym for **type(x,t)**, though we have just seen that it is not a direct synonym unless it is used in a boolean expression. The **patmatch** and **typematch** commands also use the `::` notation as part of their syntax for pattern matching.
```
>
```

```
>
```

12.10. Tables and arrays (optional)

Tables and arrays are two very important and closely related data structures. Both data structures can be though of as a way to generalize the list data structure. Arrays allow for more than just one integer index and tables allow for more general indices than just integers. Both data structures permit the use of index functions for computing, rather than looking up, the value associated with an index. Arrays are also the basis for Maple's vector and matrix data types, which are used to model mathematical vectors and arrays from calculus and linear algebra.
```
>
```

12.10.1. Tables

A table data structure can be thought of as a generalization of a list data structure. A list holds items, and the name of each item is the name of the list indexed with an integer. A table is a data structure that holds items much like a list, but the name of each item is the name of the data structure indexed by an arbitrary expression.

When we create a list, Maple "indexes" the items in the list automatically, using the order of the items in the definition of the list. For example define the following list (of symbols).
```
> l := [first, second, third];
```
The items in this list are automatically indexed in the obvious way.

```
> 'l[1]' = l[1],
> 'l[2]' = l[2],
> 'l[3]' = l[3];
>
```

Now let us see how we create a table and put indexed items into it. Here is a command that creates a table data structure.

```
> table();
```

The **table** command created an empty table. This table is also anonymous, so let us give it a name.

```
> t := %;
```

Now let us put some items into this table. Since an item in a table can have an arbitrary index, Maple cannot automatically index a table for us. So when we put an item in a table, we need to explicitly tell Maple what the index should be for the item. Here is how we put two items in the table and inform Maple what we want the index for each item to be.

```
> t[apple] := red;
> t[pear] := green;
```

By using the name of the table along with an index we can recall an item from the table.

```
> t[pear], t[apple];
```

Here is how we can see what is inside the whole table data structure.

```
> eval( t );
>
```

It is also possible to simultaneously create a table and put indexed items into it. The following command creates and initializes a table and then names it **t**, all in just one command.

```
> t := table([apple=red, pear=green, orange=orange, math=fun]);
```

We initialize the table by giving the **table** command a list of equations. In each equation, the left hand side is the index for the item on the right hand side. Each pair of an index and an item is often called a **key-value pair**. The index is considered a key and the item is the value for the key. Sometimes it is useful to find out what all the keys or values are for a table. The **indices** command returns a sequence of all of the keys in a table.

```
> indices( t );
```

The **entries** command can be used to get a sequence of all of the values in a table.

```
> entries( t );
>
```

Exercise: Explain why the list returned by the **indices** command cannot have any duplicates in it. Explain why the list returned by the **entries** command may contain duplicates. We say that a table is 1-1 if there are no duplicates in the list returned by **entries**.

```
>
```

So far we have seen how to use the **table** command to create both an empty and a non empty

table and we have seen how to add indexed items (i.e. key-value pairs) to a table by using an assignment statement. We can also remove items from a table by unassigning the item's indexed name. Let us remove an item from the table named **t**.

```
> t[math] := 't[math]';
> eval( t );
>
```

We can create a table implicitly (as opposed to explicitly by using the **table** command) by assigning a value to an indexed name where the header of the indexed name is unassigned. Let us make sure that the name **T** is unassigned.

```
> T := 'T';
```

The next command (implicitly) creates a table named **T**.

```
> T[something] := anything;
> eval( T );
```

If we now unassign the name **T[something]**, that will not destroy the table named **T**, it will just make **T** an empty table.

```
> T[something] := 'T[something]';
> eval( T );
```

To get rid of a table (whether it is empty or not), we need to unassign the table's name.

```
> T := 'T';
> eval( T );
>
```

Now let us look more carefully at the contents of a table data structure. A table data structure needs to store items and an index for each item. Recall that our table data structure named **t** holds three key-value pairs.

```
> eval( t );
```

So how many operands does this data structure hold, given that the table has three key-value pairs?

```
> nops( eval(t) );
```

This table is considered a data structure with two operands in it. But that does not really seem to make sense for this table. Let us see what these two operands are.

```
> op( eval(t) );
```

Again, this does not seem to make sense. The above command returned only one operand, a list. Let us ask for each of the two operands from the table data structure individually.

```
> op( 1, eval(t) );
> op( 2, eval(t) );
```

The second data item in our table is a list of equations. It is this list that holds the data that we are interested in, the table items and their respective indexes. Notice that the first data item in our table is NULL, the empty expression sequence. In general, the first data item of a table is either NULL or the name for something called an index function. We will discuss index functions in a later section.

In general, a table data structure holds two items. The first item is either the name of an index function or NULL, and the second item is a list of key-value pairs stored in equation data structures.

```
> 
```

Notice that when we were analyzing the structure of the table named **t**, we always used the **eval** command inside the **nops** and **op** commands. Here is why. A table is one of the data structures for which Maple uses last name evaluation.

```
> whattype( t );
> whattype( eval(t) );
```

So without the **eval** command, **t** evaluates to the name **t**, and **nops(t)** will return 1.

```
> nops( t );
```

But the **op** command evaluates **t** in an unusual way. If **op(t)** were to use last name evaluation for **t**, then **op(t)** would return **t**, since a symbol is a data structure that contains just one operand which is the symbol itself. If **op(t)** were to use full evaluation for **t**, then **op(t)** would return the operands of the table **t**. But **op(t)** does neither of these two things.

```
> op( t );
```

The command **op(t)** returns the table itself, so **op(t)** acts like **eval(t)**. On the other hand, the command **op(0,t)** uses last name evaluation.

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

And the command **op(1,t)**, like **op(t)**, acts like **eval(t)**.

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

We will see this kind of behavior from the **op** command again when we look at the array data structure in the next section and the procedure data structure in the next worksheet.

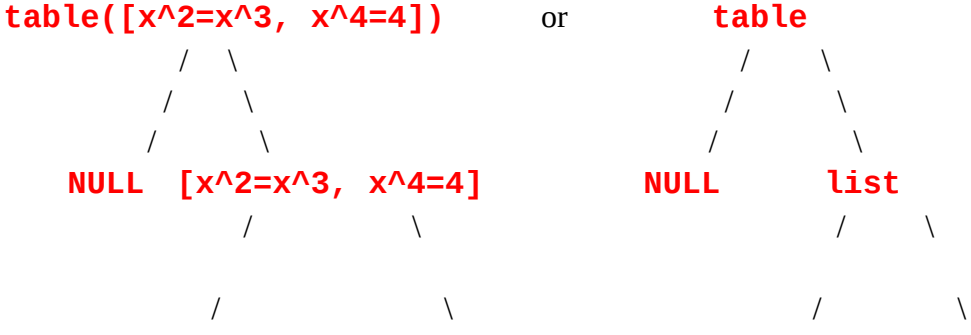
```
> 
```

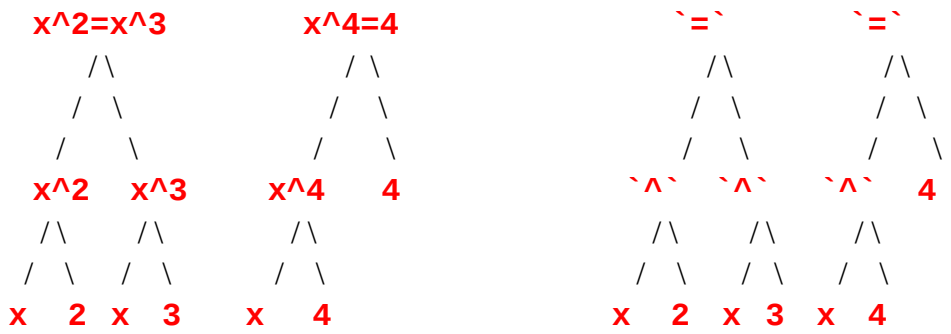
The table data structure acts differently from other data structures with respect to some other commands also. Consider how the **subs** command works with tables. Here is a table named **tt** with two key-value pairs.

```
> tt := table( [x^2=x^3, x^4=4]);
```

Here is what we would expect the data structure tree for this table to look like.

```
> 
```





Let us try doing some substitutions with this table. The expression **x^3** is in the table data structure, so we can substitute for it.

```
> subs( x^3=sin(x), eval(tt) );
```

The expression **4** is in the table data structure twice, but we can only substitute for one occurrence of it.

```
> subs( 4=5, eval(tt) );
```

The expression **x^2** is in the table data structure, but we cannot substitute for it.

```
> subs( x^2=new_index, eval(tt) );
```

The expression **x^4=4** is also in the table data structure, but we cannot substitute for it either.

```
> subs( (x^4=4)=(new_index=new_item), eval(tt) );
```

It seems that we can only substitute for expressions that are on the right hand side of each equation data structure within the table data structure.

```
>
```

Recall that we described nesting indexed names in the subsection on unevaluated indexed name data structures. Here are two ways that we can nest indexed names.

```
> 't1[m[n]]';
> 't2[m][n]';
```

Notice that the structure of these two names is hardly distinguishable when they are typeset. Here are their structures as unevaluated indexed name data structures.

```
> op( 0, 't1[m[n]]' );
> op( 't1[m[n]]' );
> op( 0, 't2[m][n]' );
> op( 't2[m][n]' );
```

Let us assign a value to each of these indexed names (which will implicitly create two new tables).

```
> t1[m[n]] := Pi;
> t2[m][n] := Pi;
```

From the structure of the unevaluated indexed name **'t1[m[n]]'**, we would expect the first assignment to create a table named **t1** with a single index named **m[n]** and a value of **Pi**. Let us check this.

```
> eval( t1 );
```

From the structure of the unevaluated indexed name **'t2[m][n]`**, we would expect the

second assignment to create a table named **t2[m]** with a single index named **n** and a value of **Pi**. Let us check this.

```
> eval( t2[m] );
```

Notice that we now have an indexed name, **t2[m]**, with a value (its value is a table). That means that we must also have a table named **t2** with an index named **m** and a value that is a table. Let us check this.

```
> eval( t2 );
```

So when we assign a value to the nested indexed name **t2[m][n]**, we create nested table data structures, but when we assign a value to the nested indexed name **t1[m[n]]** we create a table data structure but not a nested one. One thing that is a bit odd about this is that the nested indexed name **t2[m][n]** does not really look nested, but it creates nested tables, while the nested indexed name **t1[m[n]]** really does look nested, but it creates a simple, un-nested table.

```
>
```

Exercise: Look again at the table named **t1**.

```
> eval( t1 );
```

There is an indexed name m_n in the table with the value π. Why doesn't this mean that there is a table named **m** with an index named **n** and a value of **Pi**?

```
> eval( m );
>
```

Exercise: Explain the structure of each of the following tables. (Notice how, in the typeset version of each assignment statement, there is almost no distinguishable difference in the form of the indexed name on the left hand side of the assignment operator.)

```
> t3[m[n]][o] := Pi;
> t4[m][n[o]] := Pi;
> t5[m][n][o] := Pi;
> eval( t3 );
> eval( t4 );
> eval( t5 );
>
```

Here is an interesting difference between lists and tables. We have seen that a table is what is sometimes called a **dynamic** data structure. That means that the size of a table can be changed after the table is created. We can add items to and delete items from a table. On the other hand, a list is not dynamic. We say that a list is a **static** data structure. Once a list has been created, we cannot change the length of the list by adding to or deleting from it. For example, here is the list **l** that we created at the beginning of this section.

```
> l;
```

Suppose that we try to put a fourth item in the list the same way we that would put an item in a table.

```
> l[4] := fourth;
```
There is a way to make `l` a name for the list `[first,second,third,fourth]`. Here is how we can do it.
```
> l := [ l[], fourth ];
```
But this command did not really "add `fourth` to `l`". It used the old list `l` to create a new list `l` and then the old list `l` was in fact "destroyed". As this example shows, adding to and deleting from a list can be cumbersome (think about adding an item to the middle of the list `l`). One of the reasons that tables are so useful in Maple is that they are dynamic data structures and so it is easy to add items to and delete items from them.. In the general study of computer science, the differences in working with static and dynamic data structures is an important topic.
```
>
```

Exercise: Add the item `zeroth` to the beginning of the list `l`. Does it really become the zero'th item in the list?
```
>
```

```
>
```

12.10.2. Arrays

An array data structure is, like a table, another way to generalize the list data structure. Like lists and tables, an array data structure holds multiple items and the name of each item in an array is given by the name of the array and an index. Like tables, an array can have multiple indices, but like lists, the indices must be integers. Like tables (and unlike lists) an array can be created without putting anything into it but like lists (and unlike tables) an array is a static data structure, not a dynamic one, meaning that the size of an array is fixed when the array is created.

We use the `array` command to create array data structures. Since the size of an array data structure is fixed when it is created, we need to tell the `array` command how big of an array to create. We do this by giving the `array` command a range of integers. The following command creates an array that holds five items, and the items are indexed with the integers from 1 to 5.
```
> array( 1..5 );
```
The last command created an empty, anonymous array. Notice that the array displayed itself as containing five subscripted question marks. This is because the array does not yet have a name that can be subscripted and there are not yet any values assigned to the slots in the array. Let us give the array a name.
```
> a := %;
```
Notice how the array redisplayed itself in a completely different way. It used a range to denote the size of the array and it used an empty list to denote that no values have been assigned to the array yet. The next command shows a third way for this array to display itself, this time using the name for the array as part of the name for each entry in the array.
```
> print( a );
```
Surprisingly, the next two commands display the array the same way as when it was first created, before it had a name.

```
> eval( a );
> op( a );
```
Let us put a value in the array.
```
> a[3] := sqrt(Pi);
```
The next three commands show three different ways to display the array. They all show that the array holds five items, indexed from 1 to 5, and that only the third entry in the array holds a value. In addition, the first display also shows that the array is named **a**.
```
> print( a );
> op( a );
> op( eval(a) );
```
Since an array is not a dynamic data structure (like a table), and the array **a** has length five and is indexed with the integers from 1 to 5, we can only make assignments to the indexed names **a[1]**, **a[2]**, **a[3]**, **a[4]**, and **a[5]**. If we try to use the header **a** with any other integer index, we get an error message.
```
> a[6];
> a[0];
```
If we use the header **a** with a symbolic index, Maple does not complain, it just treats the name as an unevaluated indexed name.
```
> a[six];
> whattype( % );
```
This is because the unassigned name **six** may later be given an integer value between 1 and 5. But if we try to assign a value to the indexed name **a[six]** (which we could do if **a** were a table), then we get an error message.
```
> a[six] := 6;
>
```

When we created the array **a** we used the range **1..5** to specify two things, that the array should hold five items and that the items should be indexed with the integers from 1 to 5. We can create another array that also holds five items but is indexed with a different range of integers. The next command creates an array of size five with the items in the array indexed with the integers from -2 to 2. Notice that this array finds still a fourth way to display itself (this display looks more like the display for a table).
```
> array( -2..2 );
```
The following command creates an array of size five indexed from 119 to 124.
```
> array( 119..124 );
```
We can give this array a name.
```
> b := %;
```
And we can place a couple of values into the array.
```
> b[120] := sqrt(2);
> b[123] := sin(Pi/9);
```
The following three commands show three different ways to display this array.
```
> print( b );
```

```
> op( b );
> op( eval(b) );
```
We can remove a value from an array by unassigning the value's indexed name.
```
> b[123] := 'b[123]';
```
Now the array **b** has only one value in it.
```
> op( eval(b) );
```
We can destroy an array by unassigning its name.
```
> b := 'b';
> op( eval(b) );
>
```

We mentioned above that unlike lists and like tables, arrays can have more than one index. We create an array with more than one index by giving the **array** command more than one range of integers. The following command creates an array that holds 12 items and the items are indexed with two integers, one index running from 1 to 3 and the other index running from 1 to 4.
```
> array( 1..3, 1..4 );
```
Notice that the array was displayed in a "matrix" form, with three rows and four columns. When we give the **array** command two ranges, the first range determines the number of rows and the second range determines the number of columns. And from looking at the way the indices are placed on the question marks in the array we see that the first index in an indexed name is the row index and the second index is the column index. Now let us give this anonymous array a name.
```
> c := %;
```
Let us assign a value to the second entry in the third row of the array.
```
> c[3,2] := Pi/sqrt(2);
```
Here is what the array looks like now.
```
> print( c );
```
In principle, arrays can have any number of indices. But arrays with one or two indices are the most important and the most common. Arrays with only one index are called **one-dimensional arrays** and arrays with two indices are called **two-dimensional arrays**. In general, an array with *n* indices is called an *n*-dimensional array.
```
>
```

Exercise: Does anything seem unusual about the result of the third of the following three commands?
```
> array( 1..3, 1..3 );
> %[2,2] := 5;
> m := %%;
>
```

When we use the **array** command to create an array we can also tell the command to initialize

some, or all, of the entries in the array. The following command creates a one-dimensional array of size five, indexed from 1 to 5, with only the first three entries in the array initialized.

```
> array( 1..5, [1,4,9] );
```

Notice that the initialization values were placed in a list. If we give the **array** command a list of initialization values without a range, then the array will, by default, have the same size as the initialization list and be indexed from 1 to whatever is the length of the initialization list.

```
> array( [1,4,9,16] );
```

Notice that the last command looks like its input is a list and its output is again a list. But the output is not a list, it is a one-dimensional array, as the next command demonstrates.

```
> op( % );
```

The fact that one-dimensional arrays can look exactly like lists can cause some confusion. We will return to this idea in the section on vectors and matrices.

```
>
```

We can also initialize two-dimensional arrays when they are created. We do this with a list of lists. Consider the following list called **data**.

```
> data := [ [1950,18.6], [1960,19.3], [1970,20.4],
>           [1980,21.6], [1990,25.4], [2000,28.1] ];
```

The following **array** command uses **data** to initialize a two-dimensional array. Each sub-list of **data** is considered a row of the resulting array. Since **data** has six sub-lists, the array has six rows. And since each sub-list from **data** has two entries, each row of the array has length two. So the array is a 6 by 2 array.

```
> array( data );
> op( % );
```

Notice, by way of contrast, that the **table** command interprets **data** as a list of lists and uses **data** to initialize a one dimensional table whose values are lists of length two.

```
> table( data );
>
```

Exercise: With **data** as the list from the last example, compare how the following two commands interpret **data**.

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

Exercise: Let **data** represent the following list.

```
> data := [[1,2], [2,3], [3,4]];
```

Explain the difference between the following two commands.

```
> array( data );
> array( 1..3, data );
>
```

Exercise: Explain the differences between each of the following three commands.

```
> a := array( [1,2,3] );
> b := array( [[1,2,3]] );
> c := array( [[1],[2],[3]] );
```

For each of **a**, **b**, and **c**, what would be the appropriate **array** command to create an empty array with the same form?

```
>
```

Exercise: In the following command, it seems as if **array** simply strips off one pair of brackets from its input to create its output. Give a better explanation of what this command does.

```
> array( [[ 3]] );
```

Hint: What do the brackets in the input represent and what do the brackets in the output represent?

```
>
```

Exercise: What conclusion can you draw from the following two commands?

```
> array( [[0,1,2,3],[0,1,2],[0,1]] );
> array( [[0,1],[0,1,2],[0,1,2,3]] );
>
```

There is another way to initialize an array besides using a list of lists. We can use a list of equations within the **array** command to assign initial values. On the left of each equation we put an index and on the right a value. Here is an example.

```
> array( 1..4, [1=Pi, 4=sqrt(2)] );
```

This has the advantage of allowing us to selectively initialize any of the entries in the array (we would not have been able to initialize the above array using just a list of values). One disadvantage of this method is that the syntax for it is a bit odd when we use it with multidimensional arrays. Here is how we might initialize a two-dimensional array.

```
> array( 1..3, 1..3, [(3,1)=Pi, (2,3)=sqrt(55), (1,1)=-1] );
```

Notice that each index was contained in a pair of parentheses, not brackets as one might expect.

```
>
```

Now let us look at the contents of an array data structure. Recall that a table data structure contains two items, an expression sequence of names of index functions (which can be NULL) and a list of equations that hold key-value pairs. An array data structure contains three items. The first is an expression sequence of names of index functions (which may be NULL), the second is an expression sequence of integer ranges, and the third is a list of equations holding key-value pairs (where the keys are always sequences of integers). Here is an example of an array data structure.

```
> a := array( symmetric, sparse, 1..3, 1..3, [[5,6,7]] );
```

Let us analyze this data structure.

```
> nops( eval(a) );
> op(1, eval(a) );
```

```
> op(2, eval(a) );
> op(3, eval(a) );
> op( eval(a) );
```

An array, like a table, is a data structure for which Maple uses last name evaluation. That is why in the last five commands we needed to use **eval(a)** to force the full evaluation of the name **a**.

```
>
```

```
>
```

12.10.3. Last name evaluation and the copy command

Maple uses last name evaluation for names that evaluate to tables and arrays. Last name evaluation has some non obvious consequences for the use of tables and arrays and in this section we look at some of these consequences. This will help give us an understanding of why Maple uses last name evaluation for these two kinds of data structures.

Before turning to tables and arrays, let us look at a very simple example that uses a list. Let **a** be a name for a list.

```
> a := [1,2,3,4];
```

Now assign **a** to **b**.

```
> b := a;
```

Make a change in **b**.

```
> b[3] := 0;
```

Only **b** is changed, **a** is left unchanged.

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

Now let us replace the list with an array. Let **a** be a name for a (1-dimensional) array.

```
> a := array( [1,2,3,4] );
```

Now assign **a** to **b**.

```
> b := a;
```

Make a change in **b**.

```
> b[3] := 0;
```

Now notice that both **a** and **b** have changed.

```
> eval( a );
> eval( b );
>
```

Let us go through all of these steps again very carefully to see what was different between the list case and the array case. Make **a** a list again.

```
> a := [1,2,3,4];
```

Now assign **a** to **b**.

```
> b := a;
```

In the last command Maple used full evaluation on the right hand side of the assignment operator so **a** was evaluated to the list **[1,2,3,4]** and then this list was assigned to **b**. The key idea here is that what was assigned to **b** was a copy of the list that is named by **a**. In other words, we now have two lists that are copies of each other. If we make a change to one of these lists, the change does not affect the other list.

Now let us return to the array case. Make **a** an array again.

```
> a := array( [1,2,3,4] );
```

And assign **a** to **b**.

```
> b := a;
```

In the last command, Maple used last name evaluation on the right hand side of the assignment operator, so **a** evaluated to **a** and then the name **a** was assigned to **b**. Notice that it is the name **a** , not the array that **a** is a name for, that is assigned to **b**. Now when we use the name **b**, it will evaluate to **a** which will evaluate to the array. So an assignment like **b[3]:=0** causes a change in the array named by **a**. The key idea here is that, because of the use of last name evaluation, there is no copy made of the array. There is only one array. This array has the name **a**, and **b** is indirectly a name for the array. Any change made to the array using either one of the names **a** or **b** will show up through the other name, since both names point (eventually) to the array.

In the case of a list, full evaluation on the right hand side of the assignment **b:=a** created a copy of the list named by **a**. What if we try forcing full evaluation on the right hand side of the assignment in the case of an array? Will that create a copy of the array? Let us try the following command.

```
> c := eval( a );
```

This looks like it might have made **c** a name for a copy of the array. But that is not what happened. Make an assignment using **c**.

```
> c[3] := Pi;
```

Now notice that **a** has also changed.

```
> eval( a );
```

So the names **a** and **c** are still names for the same array, there is no extra copy. But is the name **c** any different from the name **b**? Was there any difference between the assignments **b:=a** and **c:=eval(a)**? In fact, there is a subtle difference between these two assignments that makes **c** different from **b**. If we evaluate **b** to one level, we get the name **a** (since, by last name evaluation, **b** was assigned the unevaluated name **a**).

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

But **c** evaluates at one level to the array itself.

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

In other words, **c** is itself a name for the array. Or, to put it another way, we now have one array that has two names, **a** and **c** (with a third name **b** that evaluates indirectly to the array).

```
>
```

So far we have seen that lists and arrays act quite a bit differently. When **a** was a name for a list, the assignment **b:=a** created a copy of the list and made **b** a name for the copy. When **a** was a name for an array, the assignment **b:=a** did not create a copy of the array, instead it made **b** a name for the name **a**. And the assignment **c:=eval(a)** also did not make a copy of the array, instead it made **c** another name for the array. So why does Maple treat arrays (and also tables) so much differently from lists (and the other data structures)?

The answer is that this is a way to make Maple more efficient. In real applications of Maple, it is very common to have very large arrays and tables. Arrays with tens of thousands of entries are very common. Copying an array of that size consumes a lot of time and a lot of computer memory so Maple is designed to do as little copying of arrays as possible. This idea of preventing the copying of arrays is common to many programming languages. Maple's handling of tables and arrays has much of the same effect as the use of pointers in the C and C++ programming languages and the use of reference variables in the C++ and Java languages.

```
[ >
```

But if Maple purposely avoids making copies of arrays and tables, what do we do if for some reason we need a copy? Maple has a command, **copy**, just for the purpose of making copies of arrays and tables. The following command makes a copy of our array named **a** and names the copy **b**.

```
[ > b := copy( a );
```

Now make an assignment using the name **b**.

```
[ > b[2] := 0;
```

The array named **b** is changed, and the (original) array named **a** is unchanged.

```
[ > eval( b );
[ > eval( a );
[ >
```

The **copy** command has an interesting quirk to it. The **copy** command does not make copies of tables or arrays that are contained in a table or array. Let us explain this with an example. The following command creates a 2-dimensional array named **a**.

```
[ > a := array( 1..2, 1..2 );
```

The next command creates another 2-dimensional array and assigns it to the upper left hand corner of the array named **a**.

```
[ > a[1,1] := array( 1..2, 1..2 );
```

We now have two 2-dimensional arrays, one of which is contained in the other. Here is what the array named **a** looks like.

```
[ > print( a );
```

The next command makes a copy of **a** and names the copy **b**.

```
[ > b := copy( a );
```

Now we want to show that the **copy** command did not make a copy of the sub-array contained

in **a**. In other words, we now have three arrays, not four as we might have expected. The three arrays are the array named **a**, the array named **b**, and the sub-array, which is a sub-array of both **a** and **b**. The following command assigns a value to the upper left hand corner entry of the upper left hand corner sub-array of **a**.

```
> a[1,1][1,1] := 0;
```

If we look at what is in both **a** and **b**, we see that they have both been changed by the last assignment.

```
> print( a );
> print( b );
```

So now we know that the upper left hand corner sub-array is common to both **a** and **b**. This sub-array was not copied by the **copy** command. Another way to put this is that the array named **a[1,1]** and the array named **b[1,1]** are the same array.

```
> print( a[1,1] );
> print( b[1,1] );
```

Now let us show that the arrays **a** and **b** really are distinct arrays. The next command assigns a value to the lower left hand entry in the array **a**.

```
> a[2,1] := 5;
```

If we look at the contents of both **a** and **b**, we see that the last assignment changed **a** but not **b**.

```
> print( a );
> print( b );
```

The kind of copy made by the **copy** command is called by computer scientists a **shallow copy** (or, a non-recursive copy). A copy command that would make a copy of an array and also make copies of any sub-arrays (and also any arrays contained in the sub-arrays, etc.) is called a **deep copy** (or, a recursive copy).

```
>
```

Exercise: Change the value of an entry in the matrix named **b[1,1]** and then show that the same entry is changed in the matrix named **a[1,1]**.

```
>
```

Exercise: Create an example that shows that the following command will make a deep copy of an array **a** that contains an array.

```
> b := map( copy, a );
```

Then create another example to show that this command does not make a deep copy of an array **a** that contains an array that contains an array (so the arrays are nested three deep).

```
>
```

We end this section with another example of a single array with two names. Create an anonymous array.

```
> array( 1..2, 1..2, [(2,1)=sqrt(Pi)] );
```

Now give this array two different names.

```
> A := %;
```

```
> B := %%;
```
Notice that the last two commands displayed the unassigned entries in the array using each of the array's names. The **print** command also displays the array this way.
```
> print( A );
> print( B );
```
But the **eval** command displays the unassigned array entries using a question mark.
```
> eval( A );
> eval( B );
>


>
```

12.10.4. Names, data structures, and garbage collection

The main theme of this section is that we should not confuse an object with its name. As a simple example, you are not your name. It is entirely possible for a person to live without a name. A person exits independently of their name. What has this got to do with computer programming? If we make an assignment like the following
```
> a := [1,2,3];
```
then we often use phrases like "**a** is a list" and "the list **a**". But that is confusing an object with its name. Your name is not a person, and **a** is not a list, **a** is a name for a list. It may seem very picky to say that "**a** is a list" is wrong and should be replaced with "**a** is a name for a list" and that "the list **a**" should be replaced with "the list named **a**". In fact, for list data structures the distinction is not too important. But for tables and arrays, making a distinction like this is very helpful. We have seen above that tables and arrays act differently than lists and the other data structures. When we make assignments like
```
> a := array( [1,2,3] );
> b := eval( a );
```
we know that there is only one array data structure and it has two names. But if we use phrases like "the array **a** and the array **b**", then it can become hard to remember that in fact there is only one array. On the other hand, the phrase "the array named **a** and the array name **b**" is reasonable even if we are referring to only one array. Here is another example. If we make the following assignment
```
> b[1] := 100;
```
then the phrase "the array **a** has been changed" is awkward, even if the idea behind it is true. But the phrase "the array named **a** has been changed" is very accurate, since it is the array that has actually been changed and this array just happens to have two names, **a** and **b**.

Here is another example of how an array should be distinguished from its names. Create an array named **a1**.
```
> a1 := array([10,11,12]);
```
Now unassign the name **a1**.
```
> a1 := 'a1';
```

Since the array is not its name, unassigning the name of the array does not really have any effect on the array. The array still exits. Let us change one of its entries.

```
> %%[1] := 1000;
```

Here are the contents of the array.

```
> %%%;
```

Change another one of its entries.

```
> %[2] := 2000;
```

View the array again.

```
> %%;
```

Make a (anonymous) copy of the array.

```
> copy( % );
```

Change an entry in the copy.

```
> %[3] := 3000;
```

View the original array.

```
> %%%;
```

View the (changed) copy.

```
> %%%;
```

We see that unassigning the array's name really did not effect the array itself. Again, the name is not the array. If we want to be careful about these things, we should never say "the array **a**", instead we should say "the array named **a**". Similarly, we should not say "**a** is an array", instead we should say "**a** is a name for an array".

```
>
```

Exercise: Redo the last example but with **a1** as a name for the list **[10,11,12]**. Show that the example works essentially the same with a list.

```
>
```

Unassigning an array's name brings up an interesting question. If unassigning a name for an array does not make the array go away, then what does make an array go away? First of all, why should we care about this? If we are working with very large arrays, say with tens of thousands of entires, it would not be good if every time we created such an array it stayed in Maple's memory storage for the rest of the Maple session. If we no longer are using the array, we would like Maple to reuse the array's storage space, otherwise, we may run out of memory. So we would like to know that Maple will somehow reclaim the storage space used up by any array that we are no longer using. This is what we really mean by saying that the array "goes away". The process of reclaiming storage used up by unneeded data structures is called by computer scientists **garbage collection**. So a more precise way to ask "what makes an array go away?" is "under what circumstances does Maple garbage collect an array data structure?". The answer is that Maple will reclaim the storage space taken up by an array (i.e., garbage collect the array) when the array no longer has any possible name. But we need to remember that possible names for an array can include the last result variables, **%**, **%%**, and **%%%**, and also entries in remember tables (which we will discuss in the worksheet on Maple procedures). But once Maple decides

that an array has no possible name, then it can garbage collect the space used by the array.

```
> 
```

Garbage collection does not occur immediately when an array (or table) losses all of its possible names. Maple has a way of periodically sweeping through all of its memory and looking for "garbage" data structures. But it is possible to force Maple to do this bit of internal housekeeping. For example, we may know that a very large array no longer has any names and we want to make sure that Maple reclaims its storage space. Maple has a command, **gc**, that forces garbage collection to take place.

```
> gc();
```

The **gc** command does not really have any noticeable effect in most Maple sessions since Maple's automatic garbage collection does a good enough job of cleaning memory up. Out of curiosity, we can ask Maple how many times it has done its garbage collecting since we started this Maple session.

```
> kernelopts( gctimes );
```

And we can ask how much "garbage" Maple collected the last time it did its garbage collecting.

```
> kernelopts( gcbytesreturned );
```

The last result is the number of bytes of memory space Maple recovered from unused data structures the last time it did its garbage collecting.

```
> 
```

```
> 
```

12.10.5. Vectors and matrices

Maple has two special types of array, called **vector** and **matrix**, that are especially meant for working with linear algebra and vector calculus. A one-dimensional array that is indexed starting with 1 is a vector. Here is an example of a one-dimensional array that is a vector.

```
> array( 1..2 );
> type( %, vector );
> type( %%, array );
```

Here is an example of a one-dimensional array that is not a vector.

```
> array( 0..2 );
> type( %, vector );
> type( %%, array );
```

A two-dimensional array that is indexed with both of its indices starting with 1 is a matrix. Here is an example of a two-dimensional array that is a matrix.

```
> array( 1..2, 1..2 );
> type( %, matrix );
> type( %%, array );
```

Here is an example of a two-dimensional array that is not a matrix.

```
> array( 0..2, 1..2 );
> type( %, matrix );
> type( %%, array );
```

Notice that the distinction of an array being a vector or matrix helps explain why arrays are displayed in sometimes very different formats. A one-dimensional array that is a vector is displayed in a format that looks just like a list. A two-dimensional array that is a matrix is displayed in a traditional matrix format. Arrays that are neither vectors nor matrices are displayed in a format that is much like a table.

```
> 
```

It is worth mentioning that the terminology "1-dimensional array", "2-dimensional array", and, in general, "*n*-dimensional array" can be a bit confusing when used in the context of vectors and matrices. Here are a couple of examples. Consider the following array called **v**.

```
> v := array( 1..3, [3, 7, -2] );
```

This array is a 1-dimensional array. But **v** can also be called a 3-dimensional vector since **v** can be used to represent a point in 3-dimensional space. So in one sense **v** is 1-dimensional and in another sense **v** is 3-dimensional. To say that **v** is 1-dimensional is more of a programmer's way of looking at **v**. To a programmer, the most important feature of **v** is that it has one index. To say that **v** is 3-dimensional is more of a mathematician's way of looking at **v**. To a mathematician, the most important feature of **v** is that it has three components. After a while, one gets very used to this dual way of looking at **v**. To help keep things straight, we will be careful to refer to **v** as a 1-dimensional *array* and as a 3-dimensional *vector*.

In a similar vein, consider the following array called **m**.

```
> m := array( 1..3, 1..4, [[1,2,3,4], [5,6,7,8], [9,10,11,12]]
  );
```

This is a 2-dimensional array. But **m** can also be called a 3 by 4 matrix. So we have three different "dimensions" attached to **m**, 2, 3, and 4. As before, to say that **m** is 2-dimensional is more of a programmer's way of looking at **m**, and to say that **m** is 3 by 4 is more of a mathematician's way of looking at **m**. We will be careful to refer to **m** as a 2-dimensional *array* and as a 3 by 4 *matrix*.

```
> 
```

Vectors and matrices are common enough that Maple has two special commands for creating them, **linalg[vector]** and **linalg[matrix]**. These commands have a couple of advantages over using the **array** command to create vectors and matrices. First of all, if we want to create, say, a 3-dimensional vector, we just need to give the **vector** command the dimension **3**, we do not need to give it the whole range **1..3**, since the beginning index of **1** is understood.

```
> vector( 3 );
```

Similarly, if we want to create a 3 by 4 matrix, we only need to give the **matrix** command the parameters **3** and **4**, not the whole ranges **1..3**, and **1..4**.

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

Notice that even though **vector** and **matrix** are contained in the **linalg** package, we do not need to make any reference to this package in order to use the two commands (**linalg** is

an abbreviation for "**lin**ear **alg**ebra").

```
> 
```

Vectors and matrices created with the **vector** and **matrix** commands can be given initial values using either a list or a list of lists, just as with the **array** command. A nice advantage of the **vector** and **matrix** commands is that they also allow for the use of a function to initialize a vector or matrix. For example, if **h** is a function of one variable, and we use **h** as a parameter to the **vector** command, then the vector that is created is initialized with the values **h(i)** for **i** from 1 to whatever the dimension of the vector is. Here are a few examples.

```
> vector( 3, sin );
> vector( 4, i -> i^2 );
> vector( 5, i -> x^i );
```

Notice how the last two examples used anonymous functions as the initializing function. An important special case of this kind of initialization is the following command.

```
> vector( 4, 10 );
```

In the last command, the constant **4** is interpreted as the dimension of the vector to be created and the constant **10** is interpreted as the constant function **i->10**.

```
> 
```

Similarly, if **h** is a function of two variables and we use **h** as a parameter to the **matrix** command, then the matrix that is created is initialized with the values **h(i,j)** for **i** from 1 to the row dimension of the matrix and with **j** from 1 to the column dimension. Here are a few examples.

```
> matrix( 3, 2, h );
> matrix( 2, 3, (i,j)->sin(i)/cos(j) );
> matrix( 3, 3, 3);
> matrix( 3, 3, (i,j)->x^i*y^j );
```

Much more elaborate initialization functions can be written using the ideas of Maple procedures and Maple programming that we will cover in the next few worksheets.

```
> 
```

Exercise: Create a 4 by 4 matrix whose value for each entry is the difference between the entry's indices.

```
> 
```

Exercise: Create a 50-dimensional vector whose initial values are 50 equally spaced samples of the **sin** function between 0 and 2 π.

```
> 
```

In the last section we looked at the way that Maple handles names that evaluate to an array. Because of the way that Maple handles arrays, and therefore also vectors and matrices, Maple needs a special command for evaluating algebraic expressions involving vectors and matrices.

Here is an example of why we need a special command. Let **A** and **B** be names for two 2 by 2 matrices. (Notice how these matrices are being initialized in slightly different ways.)

```
> A := matrix( 2, 2, [1,2,3,4] );
> B := matrix( [[5,6], [7,8]] );
```

Let us try to compute the sum **A+B**.

```
> A + B;
```

That did not get us anywhere because of last name evaluation. Maple will only evaluate **A** to the name **A** and **B** to the name **B**, so we get left with the symbolic sum **A+B**. Let us try to force full evaluation of the names **A** and **B**.

```
> eval( A+B );
```

That did not do any good. Let us try full evaluation before doing the addition.

```
> eval( A ) + eval( B );
```

Maple still did not do the matrix addition. So here is how we can get Maple to add these matrices. We need to use the **evalm** command (for "**eval**uate **m**atrices").

```
> evalm( A+B );
```

Here is another example of the need for **evalm**. Let us try to multiply a matrix with a scalar.

```
> -2*A;
> -2*eval( A );
```

We can get Maple to do the multiplication by using **evalm**.

```
> evalm( -2*A );
```

Here is another example of the need for **evalm**.

```
> A^2;
> eval( A )^2;
```

We need **evalm** to do the matrix exponentiation.

```
> evalm( % );
>
```

Here is another way in which matrix algebra is different from the algebra of numbers. In general, multiplication of matrices is not commutative. So if A and B are matrices, then in general AB will not be equal to BA, and ABA will not equal $A^2 B$ or BA^2. But look what Maple does with the product **A*B*A** of our matrices **A** and **B**.

```
> A*B*A;
```

The culprit here is automatic simplification. Maple will automatically simplify any expression *before* it tries to do any evaluation. So without even knowing that **A** and **B** evaluate to matrices, Maple simplified **A*B*A** to **A^2*B**. Then, because of last name evaluation, Maple does not do any further evaluation. If we now apply **evalm** to the last result, then we get an incorrect result for the calculation of ABA.

```
> evalm( % );
```

To prevent Maple from making inappropriate simplifications like these, Maple has a special, noncommutative, form of the multiplication operator, **&***. Here is the correct way to get Maple to do the multiplication ABA.

```
> A &* B &* A;
```

```
> evalm( % );
>
```

Here is a (possibly dangerous) quirk of the **evalm** command. If we ask it to multiply two matrices and inadvertently use the commutative multiplication operator, then we get an error message.

```
> evalm( A*B );
```

But if we ask it to multiply three matrices and use the commutative multiplication operator, then there is no error message and the resulting calculation is not what we want.

```
> evalm( A*B*A );
```

Using the commutative multiplication operator with matrices is an easy mistake to make, but do not depend on the **evalm** command to catch it.

```
>
```

We have seen that Maple needs a **copy** command for creating a copy of a matrix and it needs an **evalm** command for evaluating algebraic expression involving matrices. Now we will show that, for the same reason that Maple needs the **copy** and **evalm** commands, Maple also needs a special command, **equal**, for comparing two matrices (or vectors) to decide if they are equal or not. Before going over what **equal** does, let us see why it is needed. Create a matrix and name it **A**.

```
> A := matrix( 2, 2, [1,2,3,4] );
```

Make **B** another name for the matrix named **A**.

```
> B := eval( A );
```

Create a copy of the matrix and name the copy **C**.

```
> C := copy( A );
```

Now we have two matrices that are exact copies of each other and we have three names for these two matrices. The equation **A=B** should be true because **A** and **B** are two names for the same matrix, and the equation **A=C** should be true since **A** and **C** are names for matrices that are copies of each other. But if we evaluate these equations with **evalb** we do not get the desired results.

```
> evalb( A=B );
> evalb( A=C );
```

The reason that these equations are false is last name evaluation. Since Maple does not evaluate the names **A**, **B** and **C**, the equations **A=B** and **A=C** are false since **A** is not the same name as either **B** or **C**. So let us try forcing full evaluation of the names **A**, **B**, and **C**.

```
> evalb( eval(A) = eval(B) );
```

That seemed to work as it should, since **A** and **B** are really two names for the same matrix.

```
> evalb( eval(A) = eval(C) );
```

But the last command did not return what we expected, even though the matrices named **A** and **C** are exact copies of each other. So the **evalb** command does not work for comparing the equality of two matrices. Instead, we need to use the **equal** command from the **linalg** package.

```
> linalg[equal]( A, B );
> linalg[equal]( A, C );
>
```

The **equal** command compare two matrices (or vectors) entry by entry to see if they have the same values. However, the **equal** command does a shallow comparison, much like the **copy** command does a shallow copy. The **equal** command will not check the entries of a matrix contained within a matrix. Here is an example that uses vectors. Let **u** and **v** be names for two vectors that are exactly alike and also contain a sub-vector.

```
> u := vector( 2, [vector( 2, [10,20]), 300] );
> v := vector( 2, [vector( 2, [10,20]), 300] );
```

We would expect the following command to return true, since every entry of **u** is equal to the corresponding entry of **v**.

```
> linalg[equal]( u, v );
```

The reason that the last command returned false is somewhat subtle. The second entry of **u** is obviously the same value as the second entry of **v**. And the first entry of **u** is equal to the first entry of **v**, as the next command shows.

```
> linalg[equal]( u[1], v[1] );
```

But the first entry of **u** is not the same vector as the first entry of **v** so the first entry of **u** does not have the same value as the first entry of **v**. The **equal** command does not check if these two sub-vectors are equal.

In fact, the **equal** command does not really work properly with nested vectors and matrices. Here is a simple example.

```
> u := vector( 2, [vector( 2, [10,20]), 300] );
> v := copy( u );
> linalg[equal]( u, v );
```

If two vectors that are copies of each other are not equal, then something is wrong with the **equal** command.

```
>
```

In a previous section we mentioned that vectors looking exactly like lists can cause some confusion. Here is an example. Let us define two vector valued functions represented as lists of expressions.

```
> v1 := [t, t^2, t^3];
> v2 := [t^3, t^2, t];
```

Now compute the cross product of **v1** and **v2**.

```
> v3 := linalg[crossprod]( v1, v2 );
```

Even though **v3** looks a lot like **v1** and **v2**, it is fundamentally different because **v1** and **v2** are lists but **v3** is a vector.

```
> whattype( v1 );
> whattype( v2 );
```

```
> whattype( op(v3) );
```

Here is an example of where the distinction between lists and vectors is important. Let us compute the derivative of each of the functions, **v1**, **v2**, and **v3**.

```
> diff( v1, t );
> diff( v2, t );
> diff( v3, t );
```

Notice that the last result is not correct. The **diff** command does not work directly with vectors. Here is how to differentiate a vector (i.e., an array).

```
> map( diff, v3, t );
```

When doing vector calculus with Maple, we need to be careful which of our "vectors" are lists and which are Maple vectors.

```
>

>
```

12.10.6. Index functions

Recall that we mentioned above that the first item in a **table** or **array** data structure is either NULL or the name of an index function. An index function is a function associated to a table (or array) that allows the table (or array) to compute, rather than look up, the value associated to certain keys.

Maple has five built in index functions, **symmetric**, **antisymmetric**, **diagonal**, **sparse**, and **identity**. Here is how we define a table that uses one of the predefined index functions.

```
> sym_t := table( symmetric );
```

Now **sym_t** is a table that is symmetric, that is for any pair of indices **i**, **j**, the value of **sym_t[i,j]** will be equal to the value of **sym_t[j,i]**. Here are some examples.

```
> sym_t[a,b];
> sym_t[b,a];
> sym_t[4,2];
> sym_t[a,b] := 2;
> sym_t[b,a];
```

Let us look at what is stored in the table.

```
> op( sym_t );
```

Notice that there is no entry in the table for the index **(b,a)**. Instead the index function uses the entry for the index **(a,b)** to compute the value for the index **(b,a)** (the index function makes it appear as if there is an entry for the index **(b,a)** when there really isn't one). Let us try to assign a value to the index **(b,a)**.

```
> sym_t[b,a] := 12;
```

Now let us look at what is in the table.

```
> op( sym_t );
```

The index function updated the value for the index **(a,b)** instead of putting in an entry for the

index **(b,a)**.
>

Each index function can be used to define a table that has a certain kind of structure. The **antisymmetric** index function can be used to define a table **t** for which **t[a,b]** is always equal to **-t[b,a]**. The **identity** index function can be used to define a table **t** for which **t[a,a]** is always equal to 1 and **t[a,b]** is always equal to 0 when **a** is not the same as **b**. The **diagonal** index function can be used to define a table **t** for which **t[a,a]** can be given any value but **t[a,b]** is always equal to 0 when **a** is not the same as **b**. And the **sparse** index function defines a table which returns the value 0 for any index that has not explicitly had a value assigned to it.
>

It is possible to create tables with other kinds of properties by defining our own index functions. Defining our own index function is also a good way for us to see just how an index function works. Below is an example of an index function that will give a table the property that it is odd in the first index variable, that is **t[-i,j]** will always be equal to **-t[i,j]**.

Note: The rest of this section will use a lot of the material from the worksheets about procedures and Maple programming. If you have not read those worksheets yet, then you can skip ahead in this worksheet to the section on structured data types.
>

An index function is a procedure that has a name that begins with **index/**. An index function has two formal parameters (though we will see that it might be called with three actual parameters). The first formal parameter is a list of indices. The second parameter is the name of the table that the function is working for. An index function is responsible for both looking up values from a table and for assigning values to a table. If an index function is being called to look up a value in a table, then the index function is called with two actual parameters (the indices and the name of the table). If an index function is being called to assign a value into a table, then the index function is called with three actual parameters. The third parameter is a list containing the value to be put in the table.
>

Here is an example of an index function. This index function defines a table (or array) that has the property of being odd in its first index. That is **a[-i]** will equal **-a[i]** for a table or array a defined to use this index function. Notice that the body of the procedure begins with a test on the number of actual parameter in the procedure call. This divides the body of the procedure into the two cases of looking up a value and assigning a value.

```
> `index/odd` := proc(indices, table)
>   if (nargs = 2) then  # get an entry from the table
>     if (assigned(table[op(indices)])) then
```

```
>       table[op(indices)];
>     else
>       -table[-op(1,indices),op(2..-1,indices)]
>     fi
>   else  # put an entry in the table
>     table[op(indices)] := op(1, args[3])
>   fi
> end;
```

Here is how we define a table that uses this index function (notice that in the **table** command we drop the **index/** from the name of the index function).

```
> t := table( odd );
```

Now let us test this index function. The table **t** is currently empty. If we look up a value, the index function will cause the indexed name to return "unevaluated" but it will do so in a way that indicates the structure of the table.

```
> t[a];
> t[-a];
```

Let us put a value in the table.

```
> t[a] := -5;
> op( t );
```

Let us retrieve the value and let us also look up the related index **-a**.

```
> t[a];
> t[-a];
```

Notice that, like the example of the **symmetric** table, there is no entry in the table for the index **-a**. The index function computes the value for this index and makes it appear as if there is an entry for this index.

```
> op( t );
```

Here are a few more examples of how this index function works.

```
> t[a,b,c];
> t[a,b,c] := d;
> t[-a,b,c];
> t[-a,-b,c];
> t[-x-y];
> t[x+y] := 1,2,3;
> t[-(x+y)];
> t[3] := -4;
> t[-3];
> t[-3] := Pi;
> op( t );
```

Notice that the last assignment broke the structure of the table, it is no longer odd. This is a flaw in our design of the index function. Here is a new version of the index function that fixes this bug. In this version, the index function is careful to check if the associated index is already assigned before it makes an assignment to the index given in the procedure call.

```
> 
> `index/odd` := proc(indices, table)
>   if (nargs = 2) then  # get an entry from the table
>     if (assigned(table[op(indices)])) then
>       table[op(indices)];
>     else
>       -table[-op(1,indices),op(2..-1,indices)]
>     fi
>   else   # put an entry in the table
>     if (assigned(table[-op(1,indices),op(2..-1,indices)]))
  then
>       table[-op(1,indices),op(2..-1,indices)] := -op(1,
  args[3])
>     else
>       table[op(indices)] := op(1, args[3])
>     fi
>   fi
> end;
```

Let us define a new table using this new version of the index function.

```
> t := table( odd );
```

Let us test this version of the index function.

```
> t[1] := 1;
> t[-1];
> t[-1] := sqrt(2);
> t[1];
> t[-2] := 0;
> t[2];
> t[2] := -100;
> t[-2];
```

Notice that, even though there are assignments to the names `t[-1]` and `t[2]`, there are entries in the table only for indices `1` and `-2`, since these indices were assigned to first.

```
> op( t );
> 
```

Exercise: Write an index function `` `index/even` `` that will define an even table, i.e., a table for which the value of `even_t[a]` is the same as `even_t[-a]` for any index `a`.

```
> 
```

Exercise: Write an index function `` `index/doubles` `` that defines a table so that if an even index 2 *n* has a value, then the odd index 2 *n* + 1 will have the same value.

```
> 
```

Exercise: A table can be defined with more than one index function (see the help page for index functions). Define tables using both the **symmetric** index function and the **even** index function from the above exercise. These two index functions can be combined two different ways; see the commands below. Analyze the structure of the resulting tables.

```
> t1 := table( symmetric, even );
> t2 := table( even, symmetric );
>
```

Exercise: Define a periodic index function **`index/periodic5`** that makes a table appear to be periodic with period 5. Have the index function use the **ERROR** command to produce an error message if a non-integer index is used when assigning to the table.

```
>
```

As with tables, we can define an array that uses an index function. The following command creates, and partially initializes, an array using Maple's built in index function **symmetric**.

```
> array( symmetric, 1..3, 1..3, [[5,6,7]] );
```

The next command shows that the array has only three values in it, even though the above display shows five values. The other two values are computed by the index function.

```
> op( % );
```

With one and two dimensional arrays, because of the way that Maple displays them, it is usually easier to see what an index function does than with tables. For example, the following command creates an "empty" antisymmetric array. From the display we quickly see that an antisymmetric array has zeros on the main diagonal and what is below the main diagonal is the negative of what is above the main diagonal.

```
> array( antisymmetric, 1..3, 1..3 );
```

The next command shows that this antisymmetric array really does not contain any values. The zeros on the diagonal are computed by the index function, and the index function determines that the entry $?_{2,1}$ is equal to $-?_{1,2}$.

```
> op ( % );
>
```

Exercise: Create some one and two dimensional arrays using the **even**, **odd**, **doubles**, and **periodic** index functions defined above. Try combining two or more of these index functions to get a quick, visual display of the kind of array that a combination will define.

```
>
```

Exercise: Write an index function **index/block2** so that if **i** is an odd integer, then array entires **a[i,i]**, **a[i+1,i]**, **a[i,i+1]**, and **a[i+1,i+1]** are all equal. Try combining the **block2** index function with the **symmetric** and **antisymmetric** index functions.

```
>
```

```
>
```

12.10.7. Comparing tables with functions

It is interesting and useful to compare tables with functions. They both do something similar, which is produce a unique output for a given input. Notice how functions and tables also use very similar notation to represent the output associated to a given input. Functions have functional notation using parentheses, **f(x)**, and tables have index notation using brackets, **t[x]**. The similarity between functions and tables is actually very strong. In fact, in a certain sense, there is almost no difference between them. As we will see in this section, a table with an index function and a function with a remember table are almost exactly the same thing.

Here is an index function that squares the (first component of the) index. We will see that a table with this index function acts just like the squaring function with a remember table.

```
> `index/squaring` := proc(indices, table)
>   if (nargs = 2) then
>     if (assigned(table[op(indices)])) then
>       table[op(indices)]; # look up the value from the table
>     else     # calculate the value and store it in the table
>       table[op(indices)] := op(1, indices)^2
>     fi
>   else  # make an assignment that bypasses the index function
  rule
>     table[op(indices)] := op(args[3])
>   fi
> end;
```

Here is a table defined to use this index function.

```
> t := table( squaring );
```

Here is the equivalent squaring function with a remember table.

```
> f := proc(x) option remember; x^2 end;
```

Now we want to show that the table **t** and the function **f** act essentially the same. In particular, for the table **t**, if there is no value in the table for an index, then the index function calculates a value and stores it in the table. If a value for an index is in the table, then the index function just looks it up. Also, we can put an entry in the table that override the table's index function rule. And for the function **f**, if there is no value in the remember table for an input, then the function calculates a value and stores it in the remember table. If a value for an input is in the remember table, then the function just looks it up. And we can put an entry directly in the remember table that overrides the function's rule. Here are specific examples of these operations. First, calculate a few values that are not in the tables (which are currently empty).

```
> t[3]; f(3);
> t[y]; f(y);
> t[3,0,1]; f(3,0,1);
```

Now let us look in the tables to see that the values that were just calculated are there.

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

Put values in the tables that override the squaring rule (i.e., 4 squared is not 10).

```
> t[4] := 10; f(4) := 10;
```

Check that these values are now in the tables.

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

Now have the table and the function return values that are from the tables (including the "incorrect" values).

```
> t[y]; f(y);
> t[3,0,1]; f(3,0,1);
> t[4]; f(4);
```

Correct the "incorrect" values using direct assignments.

```
> t[4] := 16; f(4) := 16;
> eval( t );
> op( eval(f) );
```

So it appears as though there is little, if any, difference between a table with this index function and a function with a remember table. Of course, in this example the index function was purposely written to emulate the action of a remember table. Not every index function will make a table act like a function with a remember table.

```
>
```

```
>
```

12.11. Structured data types (optional)

12.11.1. Data types in general

We have not been very precise about what a data type is. Each one of Maple's basic data structures has a data type, but there are a lot of other data types defined in Maple (well over 100 of them) and we can even define our own data types. So what exactly is a data type? A **data type** is a set of values. A data structure is said to have a certain data type if the value stored in the data structure is in the set of values defined by the data type. Let us try to understand this definition by looking at an example. Consider the data structures `1/2` and `0.5`. We know that `1/2` is stored in a fraction data structure and has type `fraction`, and `0.5` is stored in a float data structure and has type `float`.

```
> type( 1/2, fraction );
> type( 0.5, float );
```

But both `1/2` and `0.5` are a kind of number, and Maple has a data type `numeric` which is the set of all possible numeric values.

```
> type( 1/2, numeric );
> type( 0.5, numeric );
```

Both of these numbers are positive numbers. Maple has a data type `positive` that is the set of all positive values.

```
> type( 1/2, positive );
> type( 0.5, positive );
```

Both `1/2` and `0.5` are also constants, and Maple has a data type `constant` that is the set of all constant values.

```
> type( 1/2, constant );
> type( 0.5, constant );
```

So the data structure `1/2` has (at least) the data types `fraction`, `numeric`, `positive`, and `constant`. And the data structure `0.5` has the data types `float`, `numeric`, `positive`, and `constant`.

Notice that the data types `fraction` and `float` are fundamentally different from the data types `numeric`, `positive`, and `constant`. We have just seen that `1/2` and `0.5` each have (at least) four different data types, but the `whattype` command will return exactly one data type for each of them and it is always `fraction` for `1/2` and `float` for `0.5`.

```
> whattype( 1/2 );
> whattype( 0.5 );
```

The data type `fraction` not only describes the value stored in the data structure `1/2`, it also describes exactly how Maple stores the value. Similarly, the data type `float` describes exactly how Maple stores the value of the data structure `0.5`. On the other hand, a data type like `positive` tells us something about the value in a data structure, but it does not tell us much about how the value is stored, since a data structure with type `positive` could be an integer, a fraction, or a float data structure. The data types returned by the `whattype` command are called the **basic data types** and these are the data types that exactly match Maple's data structures. Maple has 29 basic data types but Maple has well over 100 pre-defined data types (and, as we will see, we can also define our own data types).

While we are on the subject of `type` vs. `whattype`, notice that the `type` command always uses full evaluation (that is, it does not use last name evaluation).

```
> s := t;
> t := table();
> type( s, table );
```

On the other hand, the `whattype` command will sometimes use last name evaluation instead of full evaluation.

```
> whattype( s );
> whattype( t );
```

With `whattype` we sometimes need to force full evaluation.

```
> whattype( eval(s) );
```

I do not know why these two commands should use different evaluation rules. It seems needlessly confusing.

```
>
```

Here is another example of a data structure with many data types. Consider the data structure `-2`. It has the following data types (among others).

```
> z := -2;
```

```
> type( z, integer );
> type( z, even );
> type( z, negative );
> type( z, negint );
> type( z, nonposint );
> type( z, rational );
> type( z, complex );
> type( z, numeric );
> type( z, literal );
> type( z, atomic );
> type( z, algebraic );
> type( z, constant );
> type( z, realcons );
> type( z, complexcons );
> type( z, scalar );
> type( z, algnum );
> type( z, radnum );
> type( z, monomial );
> type( z, polynom );
> type( z, ratpoly );
> type( z, anything );
```

This example is meant to give you an idea of how rich the supply of Maple data types is. If you want a precise definition for any of the above data types, just click on one of them and hit the F1 key (at the top of the keyboard) to call up Maple's help page for the data type.

If you do look at some of the data type definitions, you will see that data types are defined in various ways. Some data types are unions of other data types. For example, the **`numeric`** data type is the union of the **`integer`**, **`fraction`**, and **`float`** data types. Other data types are subsets of a data type. For example **`even`** and **`posint`** are subsets of the **`integer`** data type, and **`positive`** is a subset of the **`numeric`** data type. Then there are data types that have more complex definitions. For example, the data type **`polynom`** models the mathematical idea of a polynomial and it is not easy to express the **`polynom`** data type in terms of other data types. A polynomial can have a **`+`**, **`*`**, **`^`**, **`numeric`**, or **`symbol`** data type, but of course not every data structure with these data types is a polynomial.

```
> 3*x^2+x+1;
> type( %, polynom );
> whattype( %% );
> a*x^3;
> type( %, polynom );
> whattype( %% );
> x^3;
> type( %, polynom );
```

```
> whattype( %% );
> x;
> type( %, polynom );
> whattype( %% );
> 3;
> type( %, polynom );
> whattype( %% );
```

And some expressions that look mathematically like a polynomial do not have the **polynom** data type (see the following exercise).

```
>
```

Exercise: Explain why the expression $3\,x^n$ is not considered a polynomial by the **type** command.

```
> 3*x^n;
> type( %, polynom );
```

Hint: Carefully read the help page for **`type/polynom`**.

```
>
```

It is useful to compare data types with the **properties** used in Maple's assume facility. Both types and properties are ways of describing sets of values. A type is something that a data structure can have and it tells us something about the value stored in the data structure. A property is something that an unassigned variable can have and it tells us something about the value that the unassigned variable is supposed to represent. For example, let us make an assumption about the values that the unassigned name **x** can represent.

```
> x := 'x':
> assume( x, posint );
```

The **is** command allows us to ask about the value that **x** is supposed to represent.

```
> is( x, posint );
> is( x, positive );
> is( x, integer );
> is( x, rational );
> is( x, real );
```

The **type** command cannot tell us anything about what **x** is suppose to represent.

```
> type( x, posint );
```

An unassigned variable always has the type **name**.

```
> type( x, name );
```

Notice that the **is** command also works with data structures.

```
> type( 5, posint);
> is( 5, posint );
> type( x^2+3*x+6, polynom );
> is( x^2+3*x+6, polynom );
```

Unfortunately, the ways in which we describe types and properties in Maple are not compatible.

Every type can be used as a property, but there are ways of describing simple properties that do not work for types. For example, it is easy to define a property **posfrac** that describes positive fractions.

```
> posfrac := AndProp(fraction, RealRange(Open(0),infinity));
```

Here is an example of its use.

```
> assume( x, posfrac );
> is( x, positive );
> is( x , fraction );
```

We can use this property with the **is** command and a data structure.

```
> is( 2/3, posfrac );
```

But we cannot use this property with the **type** command.

```
> type( 2/3, posfrac );
```

I do not know of an easy way to define a simple type like **posfrac** (though we will return to this problem in the section below on defining types). It would be nice if Maple had a unified syntax for describing sets of values, i.e., types and properties.

```
>
```

```
>
```

12.11.2. Structured data types

Some data types describe the values in a data structure with more detail and precision than other data types. These more precise data types are called structured data types. In this section we explain what structured data types are and we explain how to define them.

Let us start our discussion of structured data types with an example. The expression **5*x** has the data type **`*`**.

```
> type( 5*x, `*` );
```

The above **type** command did not really tell us much about the actual value stored in the **`*`** data structure. Consider the following **type** command, which uses a structured data type.

```
> type( 5*x, numeric &* name );
```

The structured data type **numeric&*name** describes a product of a number and a name. The last **type** command told us quite a bit about the value in the **`*`** data structure. It told us that the value is some number times some name (though it does not tell us what the number and name are). The expressions **y*x** and **5*x*y** also have the type **`*`**, but they do not have the structured type **numeric&*name**.

```
> type( x*y, `*` );
> type( 5*x*y, `*` );
> type( x*y, numeric &* name );
> type( 5*x*y, numeric &* name );
```

Here are structured data types that more accurately reflect the values in the data structures **y*x** and **5*x*y**.

```
> type( y*x, name &* name );
```

```
> type( 5*x*y, `&*`(numeric,name,name) );
```
The structured data type `name&*name` describes a product of two names, and the structured data type `` `&*`(numeric,name,name) `` describes a product of exactly three things, a number and two names.
```
>
```

Here is another example. Consider the following list.
```
> l := [1, 1/2, 2, 3/2, 3];
```
This list has, of course, the data type of `list`.
```
> type( l, list );
```
But we can say more about this list. It is a list of rational numbers. It also happens to be a list of positive numbers. In addition, we can say that the list named `l` is not a list of integers nor is it a list of fractions but it is a list of integers and fractions. Here are `type` commands that express these last few sentences using structured data types.
```
> type( l, list(rational) );
> type( l, list(positive) );
> type( l, list(integer) );
> type( l, list(fraction) );
> type( l, list({integer,fraction}) );
```
Here are two lists of lists.
```
> list_1 := [ [a,b,c], [1,2], [apple,orange], [f] ];
> list_2 := [ [1.2, 3.9], [2.2, 12.0], [-3.2, 7.5] ];
```
The structured type `list(list)` describes lists of lists.
```
> type( list_1, list(list) );
> type( list_2, list(list) );
```
But `list_2` has a bit more structure than `list_1`. We can describe `list_2` as a list of pairs, and more specifically as a list of pairs of numbers. Here are structured data types that make these distinctions.
```
> type( list_1, list([anything,anything]) );
> type( list_2, list([anything,anything]) );
> type( list_2, list([numeric,numeric]) );
>
```

Structured data types are a special kind of Maple expression that have their own syntax. This syntax is described in the help page for structured data types, but it may be a bit difficult to read if you have never seen a syntax definition before. Instead of trying to formally define the syntax for structured data types (which is what is done in the help page), here we give examples and explanations of the most important forms of structured data types.

A structured data type of the form `list(type1)` matches a list data structure, all of whose operands have `type1` (where `type1` could itself be another structured data type).
```
> type( [1,2,3], list(integer) );
```

```
> type( [u,v,w], list(symbol) );
```
A structured data type of the form **[type1,type2,type3]** matches a list with exactly three operands with types **type1**, **type2**, and **type3**, respectively. (This kind of structured data type can have any number of types inside of the brackets, not just three.)
```
> type( [1/3, 3/5, 0.5], [fraction,fraction,float] );
>
```

A structured data type of the form **function(type1)** will match any unevaluated function call with (any number of) inputs of type **type1**.
```
> type( g(1), function(integer) );
> type( g(1,2,3,4), function(integer) );
> type( f(), function(integer) );
> type( sin(Pi/2), function(constant) );
> type( 'sin'(Pi/2), function(constant) );
```
The structured data type **function(anything)** will match any unevaluated function call (it is really equivalent to the data type **function**).
```
> type( f(x, 2, apples=oranges, g(y)), function(anything) );
```
A structured data type of the form **anyfunc(type1,type2,type3)** will match any unevaluated function call with exactly three inputs with types **type1**, **type2**, and **type3** respectively. (This kind of structured data type can have any number of types inside of the parentheses).
```
> type( g(1,2,3), anyfunc(integer) );
> type( g(1,2,3), anyfunc(integer,integer,integer) );
> type( 'plot'(f, -1..2), anyfunc(name, `..`) );
```
A structured data type of the form **specfunc(type1,f)** will match an unevaluated function call of the function named **f** with (any number of) inputs of type **type1**. (Any name can be used in place of the name **f**.)
```
> type( g(1), specfunc(integer,g) );
> type( g(1,2,3,4), specfunc(integer,g) );
> type( f(1), specfunc(integer,g) );
> type( 'plot'(f, -1..2), specfunc(anything, plot) );
```
There is no structured data type for an unevaluated function call of a specific function with a specific number and type of inputs.
```
>
```

Structured data types of the form **`&+`(type1,type2,type3)** and **`&*`(type1,type2,type3)** match sum and product data structures (respectively) with exactly three operands of type **type1**, **type2**, and **type3**, respectively. (These structured data types can have two or more types inside of the parentheses.)
```
> type( x[1]+x[2]+y[0], `&+`(indexed) );
> type( x[1]+x[2]+y[0], `&+`(indexed,indexed,indexed) );
> type( 1/2*x*y[0], `&*`(fraction,symbol,indexed) );
```

```
> 
```

A structured data type of the form **{type1,type2}** means a union of the data types **type1** and **type2**. In other words, a data structure has the structured type **{type1,type2}** if it has **type1** or it has **type2**. (There can be any number of types inside of the braces.)

```
> type( 1/2, {fraction,name} );
> type( x[0], {fraction,name} );
> type( [1/2, x[0], y, 2/3], list({fraction,name}) );
> type( 3*x, `&*`({numeric,name},name) );
> type( a*x, `&*`({numeric,name},name) );
> 
```

A structured type of the form **type1^type2** matches a **`^`** data structure whose first operand has type **type1** and whose second operand has type **type2**.

```
> type( x^2, {name^numeric, name^name} );
> type( x^n, {name^numeric, name^name} );
> type( 2^x, {name^numeric, name^name} );
```

Notice that the following syntax seems to make sense, and should be equivalent to the above structured data type, but it does not work (in Maple VR5.1).

```
> type( x^2, name^{numeric,name} );
> 
```

A structured type of the form **type1=type2** matches a **`=`** data structure whose left hand side has **type1** and whose right hand side has **type2**.

```
> type( x^2-x-1=0, polynom=numeric );
```

Notice that the following works, even though **0** is not a name for a type.

```
> type( x^2-x-1=0, polynom=0 );
```

But the following does not work.

```
> type( x^2-x-1=z, polynom=z );
> 
```

A structured type of the form **name(type1)** matches an unevaluated name that evaluates (at one level) to a value of type **type1**.

```
> y := 'x'; x := 0;
> type( x, name(integer) );
> type( 'x', name(integer) );
> type( 'y', name(integer) );
> type( 'y', name(name) );
> type( 'y', name(name(integer)) );
> 
```

Exercise: Create a structured data type that represents lists of lists where the inner lists are pairs

or triples. So the list `[[a,b],[a,b,c],[1,2]]` will have this data type but the list `[[a],[a,b],[a,b,c]]` will not.
[>

Exercise: Create a structured data type that matches any unevaluated function call with exactly two inputs.
[>
Create a structured data type that matches any unevaluated function call with three or less inputs but no unevaluated function call with four or more inputs.
[>

Exercise: Write a structured data type that describes expressions where a variable is given a range of numbers, like `x=1..5`, `z=1..sqrt(5)`, or `theta=0..Pi`.
[>

Exercise: Look at the help page for structured data types and figure out what is wrong with the last two bullet items (just before the examples). Hint: Essentially the same mistake is made in each of the bullet items and it is a mistake in the logic.
[>

[>

12.11.3. Surface and nested data types

There are many ways to classify data types. For example, the data types that we have discussed so far all fall into one of three classes of data types. The first class is the 29 basic data types that directly reflect Maple's 29 basic data structures. The second class is the system data types, which are the (100 or so) data types that are defined and named by the Maple system. The third class is the structured data types that we define using the syntax from the last section.
[>

Another way to classify data types is as surface types or as nested types (but, as we will see, some data types are neither surface types nor nested types). We define surface and nested data types in terms of data structure trees. A **surface data type** is a data type that is checked by examining only the top node of a data structure tree. A **nested data type** is a data type that is checked by examining all of the nodes of a data structure tree.
[>

All of the basic data types are surface data types (why?). Many system types are also surface types. For example, the system type `numeric` is a surface type since it is the union of the three basic types `integer`, `fraction`, and `float`. Some structured types are surface types. For example, the structured type `[anything,anything]` is a surface type since we only need to check that the top node is a list data structure and that the top node has two operands. We do not need to know anything about these two operands, so we do not need to examine any level of the

data structure tree except the top node (i.e., the zero'th level).

```
> 
```

On the other hand, the structured type **[equation,equation]** is not a surface type nor is it a nested type. It is not a surface type since we need to examine the first level of a data structure tree to find out if the operands have the type **equation**. And this is not a nested data type since we do not need to know anything about the structure of the equation data structures in the list.

A good example of a nested data type is **constant**. To determine if an expression represents a constant, all of the parts of the expression need to be examined to see if there are any variables anywhere.

```
> type( sin(Pi/7)-5^Pi/(2+exp(2)), constant );
> type( sin(Pi/7)-5^Pi/(2+exp(x)), constant );
```

Another common example of a nested data type is **polynom**, i.e., polynomials. Small changes in the structure of an expression can change it from a polynomial into a non polynomial. So every part of an expression needs to checked to determine if the expression is a polynomial.

```
> type( (x+a)^5*(x^3-3*z^2)^3-w*x, polynom );
> type( (x+a)^5*(x^3-3*z^a)^3-w*x, polynom );
> 
```

Exercise: Give examples of structured types for which you need to examine three levels and four levels of a data structure tree, but not all of the data structure tree, in order to check for the structured types.

```
> 
```

Exercise: The help page for surface and nested data types claims that the system types **linear**, **listlist** (which is synonymous with the structured type **list(list)**), **monomial**, **point**, **radical**, and **sqrt** are surface data types. Is the help page correct?

```
> 
```

The distinction between surface and nested data types is interesting, but it does not seem to be very useful. We will never need to rely on this distinction in order to solve a problem or explain some other aspect of Maple.

```
> 
```

```
> 
```

12.11.4. Defining data types

We have mentioned that not only does Maple have a very rich collection of over 100 predefined data types, but we can also add to this list by defining our own data types. There are several ways to define our own data types, from very simple ways to quite sophisticated ways that make use of the material on procedures from later worksheets. In this section we give several

examples of defining data types. We give both simple examples and a couple of more sophisticated examples.

```
> 
```

We define a new data type by defining a new data type name that can be recognized by the **type** command. For example, suppose we want to define a new data type called **my_type**. In order to make **my_type** a data type recognized by the **type** command, we need to provide a definition for the name **`type/my_type`**.

Here is a specific example. The structured type **list(numeric)** matches a list of numbers. For this data type to match a single number, the number must be in a list, like this **[4]**. Suppose we wanted a data type that would match lists of numbers and also a single number that might not be put inside of brackets. Here is how we can define such a data type. We call this new data type **my_type**.

```
> `type/my_type` := {numeric, list(numeric)};
```

Let us test this new data type.

```
> type( [1,2,3,4], my_type );
> type( [a,2,3,4], my_type );
> type( [4], my_type );
> type( 4, my_type );
```

Note carefully that the name we defined to Maple was the name **`type/my_type`** but the name we use in the **type** command is **my_type**. Notice that the name **my_type** is unassigned.

```
> assigned( my_type );
```

What might happen now if we assign a value to the name **my_type**? Let us see.

```
> my_type := x*y;
```

Let us try using **my_type** in a **type** command.

```
> type( [1,2,3], my_type );
```

What went wrong? Let us use one of our delayed evaluation tricks to see what parameters were passed to the **type** command.

```
> 'type'( [1,2,3], my_type );
```

The **type** command uses full evaluation and the expression **x*y** is not a valid name for a data type, so we got an error message above. We can avoid the error message by preventing the evaluation of the name **my_type**.

```
> type( [4], 'my_type' );
```

Here is a good rule to follow. When we use a type name with the **type** command, it is wise to right quote the type name to protect ourselves from the possiblility that the type name has been assigned a value. If you look at the source code to most Maple commands (we will see how to do that in a later worksheet), you will see that this is a rule that is always followed by Maple programmers.

```
> 
```

Now suppose we want to define a data type similar to **my_type** so that it matches either sets of numbers or a single number not in a set. We decide to call the data type **our_type** and we define it like this.

```
> our_type := {numeric, set(numeric)};
```

Does this work? Let us try it out.

```
> type( {1,2,3,4}, our_type );
> type( {a,2,3,4}, our_type );
> type( {4}, our_type );
> type( 4, our_type );
```

It seems to work. But mindful of our rule stated just above, we should try the next command.

```
> type( {4}, 'our_type' );
```

Now it doesn't work. Why did it seem to work? Use delayed evaluation again to see what we were really doing.

```
> 'type'( {4}, our_type );
```

So when we used the name **our_type** as a "type name", what we were really doing was passing directly to the **type** command the structured data type named by **our_type**. The name **our_type** is not in fact a name for a data type recognized by the **type** command. This example shows that it is not realy wise to define "data types" by just assigning a name to a structured data type.

```
>
```

Exercise: Make **your_type** a name recognized by the **type** command for a data type that matches a list or set of numbers plus any number by itself.

```
>
```

The above example shows how we can use structured data types to define new data types known to the **type** command. But not all data types that we might be interested in can be defined using structured data types. For example, suppose we want to define a data type **posfrac** that describes the subset of **fraction** that is positive (much like **posint** describes the subset of **integer** that is positive). We might want to do this by defining a structured data type that is the intersection of **fraction** and **positive**. But the syntax for structured data types does not seem to allow for the intersection of two data types (it does allow for the union of two data types). So here is another way to define a data type. We need to define a boolean valued procedure named **`type/posfrac`** that takes a single parameter and returns **true** when the value of the parameter has the data type we want and returns **false** otherwise. Using ideas from the worksheets on procedures, here is how we can define this data type.

```
> `type/posfrac` := proc(x)
>   if type( x, fraction ) and type( x, positive ) then true
>   else false
>   fi
> end;
```

Here are some examples using this type.

```
> type( 1/2, 'posfrac' );
> type( 2/(-3), 'posfrac' );
> type( 5, 'posfrac' );
>
```

Recall that every data type can also be used as a property with Maple's assume facility. So we can use our data type **posfrac** in the following way.

```
> is( 3/4, 'posfrac' );
```

But it is interesting to note that the syntax for properties allows for an easy way to form the intersection of two properties (while the syntax for structured data type does not allow for the intersection of two data types). Here is how we can easily get a property equivalent to our data type **posfrac**.

```
> is( 3/2, AndProp(fraction, positive) );
```

Unfortunately, the syntax for properties does not work with the **type** command.

```
> type( 3/2, AndProp(fraction, positive) );
```

As we mentioned earlier, it would be nice if Maple had a unified syntax for defining sets of values like data types and properties.

```
>
```

Exercise: Explain why the following structured data type does not define **posfrac** that way we want it defined.

```
> `type/posfrac` := fraction and positive;
> type( 1/2, 'posfrac' );
```

What kind of data structures does the structured type **fraction and positive** match?

```
>
```

Let us look at an example of using a procedure to define a "parameterized data type". Let us define a data type called **more_positive_than(n)** that takes a parameter **n**, which is a number, and the data type defines the subset of **numeric** with values greater than **n**.

```
> `type/more_positive_than` := proc(x, n::numeric)
>    if  type(x,numeric) and x > n then true
>    else false
>    fi
> end;
```

Notice carefully that the procedure **`type/more_positive_than`** has two parameters. The first parameter is for the object whose type is being checked and the second parameter is the parameterof the data type, the parameter used with the name of the data type in the **type** command. So in the following command, **7** is the actual parameter for the formal parameter **x** in the definition of **`type/more_positive_than`** and **2** is the actual parameter for the formal parameter **n**.

```
> type( 7, more_positive_than(2) );
> type( -1, more_positive_than(-12) );
```

```
> type( -1, more_positive_than(0) );
> type( -1, more_positive_than(n) );
>
```

Exercise: Earlier we mentioned that it is wise to right quote the name of a data type in a `type` command. Should we right quote a parameterized data type like `more_positive_than(n)` ?

```
>
```

Exercise: Write a definition for a parameterized data type `polynom_degree(n)` that matches polynomials with degree less than or equal to `n`.

```
>

>
```

12.12. Online help for data structures and data types

The basic commands for manipulating data structures are `op` and `nops`. They are both described in the same help page.

```
> ?op
```

The alternative notation for accessing individual data items in expression sequences, lists and sets, the bracket notation, is described in the next page.

```
> ?selection
```

We briefly mentioned the `length` command, which will compute the "length" of any data structure, which is not necessarily the same as the number of operands in the data structure.

```
> ?length
```

We kept checking the data type of a data structure by using the `whattype` command. The next command brings up the help page for the `whattype` command. This help page also lists all the **basic data types** in Maple.

```
> ?whattype
```

The following command brings up the help page for the expression sequence data structure.

```
> ?exprseq
```

The list and set data structures are described by the following help page (the following two commands are equivalent).

```
> ?lists
> ?sets
```

The `list` and `set` data types are described by the following help page (the following two commands are equivalent).

```
> ?type,list
> ?type,set
```

We looked at three kinds of numeric data structures. The following commands call up each of their descriptions.

```
> ?integer
> ?fraction
> ?float
```

There are numerous data types that include these numeric data structures. Here are a few, listed in order from simplest to most inclusive.

```
> ?type,integer
> ?type,fraction
> ?type,rational
> ?type,float
> ?type,numeric
> ?type,literal
> ?type,atomic
```

The following two commands both bring up the help page for the name (or symbol) data structure.

```
> ?symbol
> ?name
```

The following two commands will bring up the help pages for the **name** and **symbol** data types. Notice that the **name** data type is a bit more general than the **symbol** data type.

```
> ?type,symbol
> ?type,name
```

The following commands brings up the help pages for the string data structure and the **string** data type. Recall that this data type is new to Maple V Release 5. In previous versions of Maple, no distinction was made between symbols and strings.

```
> ?string
> ?type,string
> ?updates,R5,language
```

More information about strings can be found in the following example worksheet.

```
> ?examples,string
```

We have seen that Maple has a special name for the empty expression sequence, **NULL**. Maple also has the notion of an empty string and an empty name, i.e., a string or a name with no characters in it. Unfortunately, the empty name is called, in the online help, the "null string". It seems that, when converting from Release 4 to Release 5, this help page was not updated.

```
> ?nullstr
```

We have repeatedly said that expressions are examples of data structures. The kinds of expressions that we use to represent mathematical functions are usually algebraic expressions. The next help page defines algebraic expressions.

```
> ?algebraic
```

Algebraic expressions are described in Maple by the **algebraic** data type.

```
> ?type,algebraic
```

The following command brings up information about equation and inequality data structures.

```
> ?equation
```

The next two commands bring up information about the **equation** data type, which describes an equation data structure, and the **relation** data type, which describes both equation and inequality data structures.

```
> ?type,equation
> ?type,relation
```

The **equation** and **relation** data types are both considered to be special cases of the **boolean** data type. The **boolean** data type is used to describe boolean expressions. Boolean expressions are analogous to algebraic expressions, but instead of evaluating to a numeric value like algebraic expressions do, boolean expressions evaluate to either **true** or **false**. The next help page defines boolean expressions.

```
> ?boolean
```

The next help page defines the **boolean** data type.

```
> ?type,boolean
```

The **boolean** data type also includes as a special case the three **logical** data types.

```
> ?type,logical
```

The next command brings up the help page for a range data structure.

```
> ?range
```

The next command brings up information about the **range** data type, which describes a range data structure.

```
> ?type,range
```

There is also a **Range** function that can be used to test if a number is within a certain range.

```
> ?type,Range
```

These commands bring up information about the unevaluated function call data structure and data type.

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

The next help page describes the unevaluated dotted name data type.

```
> ?type,dot
```

The next help page describes the unevaluated indexed name data type.

```
> ?type,indexed
```

The next help page defines the series data structure and data type.

```
> ?type,series
```
Here is a command for creating a series data structure.
```
> ?series
```
There are some more specialized kinds of series data types.
```
> ?type,taylor
> ?type,laurent
```
Here are commands for creating taylor and laurent series.
```
> ?taylor
> ?numapprox,laurent
```
Here is the command for converting a **series** data structure into a sum.
```
> ?convert,polynom
```

The following help page describes unevaluated expressions without mentioning that they are a specific kind of data structure.
```
> ?uneval
```
The following help page describes the **uneval** data type without, again, any real mention of the underlying data structure.
```
> ?type,uneval
```

The **`::`** data structure has such a specific use in Maple that it does not even have a help page that defines the data structure or the data type. Here is a help page that describes how the **`::`** data structure is used in the definition of a procedure. We will say much more about this in the worksheets about Maple programming.
```
> ?procedure,paramtype
```
The **::** operator can also be used as a synonym for the **type** command, as mentioned near the beginning of the help page for **type**. And **::** is also used as part of the syntax for Maple's pattern matching commands **patmatch** and **typematch**.

The next help page defines the **table** data type
```
> ?type,table
```
Here is a command for creating a table data structure. This help page also describes the table data structure.
```
> ?table
```

The next command defines the **array** data type.
```
> ?type,array
```
Here is a command for creating an array data structure. This help page also describes the array data structure.
```
> ?array
```

Maple has a special command, **copy**, for creating a copy of an array or a table. This command is needed because of the special way that Maple evaluates the names for tables and arrays.

```
> ?copy
```

Recall that both tables and arrays can be given an index function. Here are help pages that describe the five predefined index functions that are available.

```
> ?symmetric
> ?antisymmetric
> ?diagonal
> ?sparse
> ?identity
```

You can also define your own index functions. Here is general information about indexing functions for tables and arrays including how to write your own.

```
> ?indexfcn
```

We mentioned that Maple also has the two data types **vector** and **matrix** which are special cases of the **array** data type. The next two help pages describe the **vector** data type and data structure.

```
> ?vector
> ?type,vector
```

There is a command in the linear algebra package, **linalg**, for creating a vector data structure.

```
> ?linalg,vector
```

The next two help pages describe the **matrix** data type and data structure.

```
> ?matrix
> ?type,matrix
```

There is a command in the **linalg** package for creating a matrix data structure.

```
> ?linalg,matrix
```

Maple has a special command, **evalm**, for evaluating algebraic expressions with variables that evaluate to matrices or vectors. This command is needed because of the special way that Maple evaluates names for arrays.

```
> ?evalm
```

We mentioned that matrix multiplication is noncommutative and so Maple needs a special symbol, **&***, to denote noncommutative multiplication. There does not seem to be a help page for this particular symbol, but **&*** is a special case of what Maple calls a **neutral operator**. Here is a help page about neutral operators in general.

```
> ?neutral
```

And Maple has the special command **equal**, from the **linalg** package, for determining the equality of two matrices or vectors.

```
> ?linalg,equal
```

The following command will bring up a general definition of a data type in Maple. The definition is pretty abstract so do not expect to fully understand it.

```
> ?type,defn
```

The definition of a data type in Maple refers to the **type** procedure. The next command brings up its documentation. This help page also includes a list of most of the data types predefined by Maple.

```
> ?type
```
Besides all the data types listed in the help page for the **type** procedure (over 100 of them) Maple also has predefined the data types described in the next help page.
```
> ?type,surface
```
The previous help pages did not list all of Maple's predefined data types. If you want to get a sense of how important data structures are to Maple and how many data types Maple has defined, go to the "Topic Search" menu item in the Help menu and enter "type"; you will get a list of about 140 entries, almost all of which are various predefined data types.

Maple also allows you to define your own descriptions of complex data structures. These descriptions are call **structured data types**. We will make some use of these when we get to the worksheets on procedures.
```
> ?type,structured
```

Finally, you might think that the next command would bring up general information about Maple's data structures, but it does not. Instead, it brings up a help page about one particular Maple data structure, the data structure that defines a "context menu", the kind of menu that appears when you right click on a Maple object. This shows two things: first, Maple's help pages are not organized as well as they might be, and second, almost everything in Maple really is a data structure, even parts of the graphical user interface!
```
> ?datastructures
```

```
>
```