# MACAULAY2 - RTG SEMINAR

AUGUST 30TH, 2023

**Step 1 (find a partner):** Try to find someone new to work with if you can. In a couple months, I'd like us to break into groups to work on projects. It's good to meet potential people to work with!

Our goal is to learn about functions in Macaulay2. This assignment might be slightly shorter than the last one, so if you get done early, go work on the last assignment, or go help some other people with this assignment.

**Step 2 (create a file, make a function):**

Perhaps name it as something like `FunctionTest.m2`.

The syntax for a function is something like this.

```
myFunction = i -> (
    j := 2*i + 1;
    j
);
```

Type this function into your file, save, and then load your file into Macaulay2 (`load "FunctionTest.m2"`), and then run the following in Macaulay2. (Try some other values too).

```
myFunction(17)
```

Some things to notice and try:

*Local variables:* I used a `:=` when defining `j`. This makes `j` a *local variable*. This means you cannot access `j` after you run the function. Typically in functions you don't want to set global variables because you might *overwrite* something accidentally. For example, if the user already had `j` set to something, and if you changed the `:=` to a `=`, it would overwrite what the user had saved!

Try setting a global variable in your function, and see if you can access it. After you are done playing with this, run

```
restart
```

in Macaulay2 to reset the memory. You'll have to load your file again though.

*Errors:* After loading your file again, if necessary, try executing the following commands:

```
R = QQ[x,y];
I = ideal(x^2 - y);
myFunction(R)
```

You should get an error. This is because Macaulay2 doesn't know how to make sense of 2 times a ring. Execute `break` to *get out of the debugger.*

```
    FunctionTest.m2:2:14:(3):[1]: error: no method for binary operator * applied to objects:
--            2 (of class ZZ)
--      *     R (of class PolynomialRing)
```

The number 2 after `FunctionTest.m2:` tells me the line number where the error occurred. That is, `FunctionTest.m2:`*2* says the error is in Line 2 of `FunctionTest.m2`.

Note `myFunction(I)` does not create an error, since the operations above make sense for ideals. `functions` in Macaulay2 do not type-check their inputs. In a couple weeks we'll learn about `methods` that do.

*Returning values:* In `myFunction`, the command `j` *with no semicolon* is what returned the value. In particular, the last entry in a function, if there is no semicolon, will be what the function returns. Another way to return values is to use the command `return`. Add another function to your file.

```
printingFunction = i -> (
    if ((i == 1) or (i == -1)) then (
        print "You gave me a unit";
        return 1;
    )
    else if (i % 2 == 0)  then (
        print "You gave me an even number";
        return i/2;
    )
    else (
        print "You gave me a non-unit odd number";
    )
)
```

Load this file into Macaulay2 and run commands like

```
i = printingFunction(1);
j = printingFunction(2);
k = printingFunction(3);
```

Then look at what `i, j, k` became with the commands `describe i`, `describe j`, `describe k`. In particular, `k` is not a number. Why not?

*Multiple inputs*: The following code will create a function that takes three inputs.

```
myNewFunction = (i,j,k) -> (
    i^2 + j^3 + k
)
```

Add that function, or a similar one, to your file. Try running it on some different kinds of inputs. Perhaps integers, rational numbers, or ideals.

Make sure your partner has finished their stuff with functions before continuing.

**Step 2 (lists and sequences):**

One of the important structures you will work with are lists. Run the following code in `Macaulay2`.

```
L1 = {1, A, "B"}
L1#0
#L1
S2 = (1, B, "C", L1, myNewFunction)
#S2
S2#3
```

Talk to your neighbor about what is going on here. We created two different objects, a `List` (made with { } brackets ) and a `Sequence` (made with `( )` parentheses). Note, lists and functions can be themselves elements of lists or sequences as in our example.

To change something from a list to a sequence use the command `toList` and `toSequence`. For example, run the following in Macaulay2.

```
L2 = toList S2
S1 = toSequence L1
```

*Combining lists and subsets*

There are various ways to combine lists and sequences with one another.

Try the following commands and see what they do (not all work!)

```
L1 | L2
S1 | S2
L1 | S2
L1 ** L2
S1 ** S2
subsets L1
subsets S1
subsets(L1, 2)
subsets(S1, 2)
append(L1, 101)
insert(2, "cat", L1)
```

Note `subsets S1` failed because Macaulay2 interpreted that command as trying to apply the (Method)Function `subsets` to the list `S1` and treating each element of `S1` as an argument.

*Lists as vectors:*

You can also treat lists as vectors.

Try the following code.

```
V1 = {3,1,2}
V2 = {7,8,9}
V1 + V2
5*V1
V1/3
```

Some other useful commands that you should try include:

```
sum V1
L3 = {V1, V2, {0,1,0}}
```

```
sum L3
sort V1
```

Lots of useful commands return lists. For example, run the following code in Macaulay2:

```
R = ZZ/5[x,y,z]
I = ideal(x^3 - y*z^2, z^2-x*y)
primaryDecomposition I
```

**Step 3 (a small project)**

The other day you created code to find all the rational points on an curve. Improve this code as follows.

Write a function that takes in an equation (homogeneous or non-homogeneous, up to you), and returns a list of all rational points on that function.

**Step 4 (other ways to use lists):**

We'll finish up with a couple more ways to use or interact with lists.

*Mutable Lists:*

In Macaulay2, you cannot change the elements of a list. While you can always just make a new list, sometimes you really want to change the entries of an existing list. For this you need *mutable lists.*

To make a mutable list, you normally start with a list, and run something like the following.

```
L = new MutableList from {1,2,3,4}
peek L
```

Note, by default, Macaulay2 only displays something like `MutableList{...4...}`. To inspect the elements, you use `peek`. This is because a `MutableList` can contain itself as an element. Let's do that, let's make the second element of `L` into `L` itself.

```
L#1 = L
peek L
```

*Applying functions to lists:*

Frequently, in Macaulay2, you will want to apply a function to the elements of a list, and see what you get.

For example, I might want to apply the function `sin` to the list of numbers { `0, 0.5, 1.1, 3.14`} . I could make a loop that does this, or I could use the following. (Generally, the methods below are faster than loops).

There are syntaxes that do that. You can run either:

```
apply({0, 0.5, 1.1, 3.14}, sin)
{0, 0.5, 1.1, 3.14}/sin
```

More commonly, you might create your own in-line function. You can do this by the example code.

```
apply({7, 3, 18}, i -> nextPrime(i^2))
```

Finally, sometimes you just want to apply to the list of integers `0, 1, ..., n`. You can accomplish this via the following command.

```
apply(5, i->(i+1)^2)
```