OCaml Environment Setup
Ocaml is available for download here: http://caml.inria.fr/download.en.html

There are installation instructions provided for Windows and Unix based operating systems. Once installed, type `ocaml` in a terminal window to start the interactive 'top level' OCaml environment. If top level starts up, you will see something like:

```
            Objective Caml version 3.10.0
#
```

Primitive Types
The basic OCaml types are `int, float, bool, char, string` and `unit` (which is a return type similar to 'void' in C). When naming values, we do not have to declare any of these types explicitly thanks to OCaml's type inference engine. So to name a character for example, we use the `let` keyword as follows:

```
            #let x = 'a';;
```

In the interactive Top Level environment, the compiler will respond to the above statement with:

```
            val x : char = 'a'
```

This tells us that the name `x` is associated with type `char` and holds the value 'a'.

The statement

```
            #let x = 2;;
```

will define an integer type with the value 2. If we intend to define a `float`, we use the statement

```
            #let x = 2.0;;
```

This is important to note especially because OCaml is statically typed, meaning we will not be able to simply convert `x` from an `int` to a `float` after it is defined. On a related note, the standard arithmetic operators are not overloaded in OCaml as they are in many languages. Addition of integers is expressed using the symbol `'+'` while the symbol `'+.'` is used to express addition of floats.

OCaml will not implicitly cast an integer to a float, so

```
            #2.0 +. 2;;
```

yields the type error

```
            This expression has type float but is here used with type float
```

(The language was developed and written in French. Some of the translations sound like Yoda speak.)

We can, however, explicitly cast an `int` to a `float` and *vice versa* using the built in functions `float_of_int` and `int_of_float` respectively.

Addition of floats:

```
#2.0 +. (float_of_int 2);;
- : float = 4.
```

Addition of integers:

```
#(int_of_float 2.0) + 2;;
- : int = 4
```

Other casting functions:

- `char_of_int`
- `int_of_char`
- `string_of_int`

The 'of' naming convention may seem odd at first, but it makes sense when we consider that these functions are mathematically defined functions. That is, when we say "a function *f* of a set *X*" we are speaking of a function whose domain is the set *X*.

Also note that no parentheses or brackets are required around parameters in function calls. If more than one parameter is passed into a function, a space is used as a delimiter.

Let It Be

The `let` keyword is also used to define functions. Since OCaml infers type, we don't specify the types of input parameters. The parameter types are inferred from the function definition.

```
#let square x =
 x * x;;
val square : int → int <fun>
```

The type inference engine knew that the input and output are both of type `int` because of the `*` operator. In the case that no type specific type is required in the function definition, any type of input type is permitted. This function takes two input parameters and returns a list containing them:

```
#let toList x y =
 [x; y];;
val toList : 'a -> 'a -> 'a list = <fun>
```

No return statement is required. Instead, the last expression in the function definition is returned as output.

We can also define a local substitution of sorts using the combination `let` <*name*> `=` <*expression*> `in.` This statement essentially means 'substitute <*expression*> for <*name*> for every following occurrence of <*name*> until the symbol `;;`'. For example:

```
#let average x y z =
     let sum =
     x +. y +. z in
     sum /. 3.0;;
```

In the last line, `sum` is replaced with `x +. y +. z` to compute the average.

References

So far all of the memory allocations we've made have been immutable. If we need to be able to change a variable's value, we need to name a reference.

```
# let x = ref 5;;
```

Now we have a named reference which holds the address of an integer with the value 5. If we "change the value" of `x`

```
# x := 20;;
```

we are actually changing the address held by `x`. The reference now holds the address of an integer with the value 20.

Recursion

Recursive functions are essential to the functional programming paradigm. To define a recursive function, you must use the `rec` keyword in the function definition as follows:

```
# let rec fib n =
if n<=0 then 0
else begin
      if n=1 then 1 else (fib (n-1)) + (fib (n-2))
end;;
```

This function returns the nth Fibonacci number. We haven't addressed `if...then` statements yet, but they are similar to what you have probably seen before. Notice that we don't any notation to group simple statements, and that we use `begin` and `end` to group more than one statement in an `if` or `else` block.

Modules

All compiled code is contained in a module. Let's escape the top level environment (ctrl+z in Unix, probably ctrl+c in Windows) and create a module. Using your favorite text editor, create a file named helloWorld.ml. Then enter the following code:

```
print_string "Hello World"
```

In a command-line environment, navigate to the file you just created and type

(If you're using Windows...)

```
> ocamlc -o helloWorld.exe helloWorld.ml
```

(If you're using Unix...)

```
>ocamlc -o helloWorld helloWorld.ml
```

Then call the executable you just created.

```
>helloWorld.exe
```

or

```
>helloWorld
```

You should receive the expected one line of output. Congratulations on creating your first OCaml module.

Lots of modules have been made available to you via your OCaml installation. To use another module in your code, you can either use the `open` keyword at the top of the file, or simply use dot notation to specify the module name and method to call. For example, we can create simple graphical displays using the Graphics module. (In a windows environment, you need to first create a custom top level by running this command in a command window: `ocamlmktop -o ocaml-graphics graphics.cma`).

Create a file named graphicsTest.ml and open it in a text editor. First, we want to include the graphics module, and show a graph:

```
open Graphics;;

open_graph " 640x480";;
```

Now we'll draw the simplest recognizable figure I could think of. The `draw_circle` method takes three arguments - the first two are the center in x,y coordinates (where the origin is the bottom left of the graph) and the last argument is the radius. The arc is similar, but with a vertical and horizontal radius, and a beginning and ending angle in degrees. (You can read the documentation for the graphics module here: http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html).

```
draw_circle 250 250 20;
draw_circle 350 250 20;
draw_circle 300 200 120;
draw_arc 300 200 60 60 180 360;

read_line ();;
```

To compile this, we need to include the graphics module. On Windows:

```
ocamlc graphics.cma graphicsTest.ml -o graphicsTest.exe
```

Linux/Unix:

```
ocamlc graphics.cma graphicsTest.ml -o graphicsTest
```

Then just run the output file to see the graph!