1. **Intro to OCaml**

OCaml is a language with a great deal of functional programming. To understand what that means, its best to juxtapose it against imperative programming:

- Imperative programming often deals with memory, and has a lot of destructive modification. Functional programming doesn't, and in fact avoids any functions that have "side effects" (aliasing, printing, etc.)
- Imperative programming primarily uses loops. Functional programming primarily uses recursion.
- Imperative programming is all about changing the program state with a sequence of commands to reach some desired effect. Functional programming is all about evaluating a bunch of expressions, in a practically mathematical way.
- In imperative programming functions are often treated as second-class objects, meaning you can't manipulate them or use them in expressions the way you can with say integers or strings. In functional programming, functions are first-class objects, meaning that you can.

OCaml is not *purely* functional, just like how C++ is not *purely* imperative, but we will avoid discussing most of OCaml's imperative features.

To install OCaml, go to **this link**.

Once you have installed it, you can open an OCaml interpreter by going to the terminal and typing `ocaml`. From there you can directly type in OCaml code (though you will have to end each line with two semicolons `;;`). If you have an OCaml filed named `fileName.ml` in the same directory, then you can run it from the OCaml interpreter with the command `#use "fileName.ml";;`.

2. Basic syntax

To declare values in OCaml, we use the syntax `let varName = expression`, as shown below:

```
let x = 6
let y = x * 7
```

If we run this code in the interpreter, we will see the following show up:

```
val x : int = 6
val y : int = 42
```

Notice, we never specified that x and y are ints, OCaml *inferred* the type. OCaml's **type inference** system will look at any expressions you write and infer the type in a logical way. It will also strictly enforce these types, so if you try to add an integer to a string, it will complain.

Let bindings can also be used to declare functions:

```
let isEven x =
    if x mod 2 = 0
    then true
    else false

let e4 = isEven 4
let e5 = isEven 5
```

```
val isEven : int -> bool = <fun>
val e4 : bool = true
val e5 : bool = false
```

Let bindings can also be used to create local variables. You can use the syntax `let v = x in e`, where `e` is another expression, to declare a variable `v` which only exists for the purposes of evaluating `e`. Below are some examples:

```
let hypotenuse (a, b) =
    let aSquared = a *. a in
    let bSquared = b *. b in
    sqrt (aSquared +. bSquared)

let h = hypotenuse (3.0, 4.0)
```

```
val hypotenuse : float * float -> float = <fun>
val h : float = 5.
```

The type `float * float -> float` means that hypotenuse takes in a **product type** (a.k.a. a tuple) of two floats and returns a float. Note that `*.` and `+.` are multiplication and addition of floats rater than ints.

You can also provide type annotations, if you wish, to explicitly declare the intended type of a variable/-function. OCaml will make sure that your code adheres to these types.

```
let sayHi (s : string) : string = "Hello " ^ s ^ "!"

let (s1 : string) = "World"
let (s2 : string) = sayHi s1
```

```
val sayHi : string -> string = <fun>
val s1 : string = "World"
val s2 : string = "Hello World!"
```

The type annotation of `sayHi` means that its input `s` must be a string, and its output must also be a string. Note that `^` is the string concatenation operator.

3. **Recursion and Pattern Matching**

To define a recursive function, we must use the `rec` keyword. Consider the definition of the factorial function below:

```
let rec factorial n =
    if n = 0
    then 1
    else n * (factorial (n - 1))

let f1 = factorial 6
```

```
val factorial : int -> int = <fun>
val f1 : int = 720
```

Instead of using the `if condition then e1 else e2` syntax, we can use something else called **pattern matching**. Pattern matching takes in an expression and matches it to one of several patterns and returns the corresponding code.

Below, factorial is re-written using a very simple example of pattern matching. In this case, there are two patterns: `0`, which will match only if `n` is 0, and `_` (called a wildcard) which always matches.

```
let rec factorial2 n =
    match n with
            0 -> 1
        |   _ -> n * (factorial (n - 1))
```

Pattern matching becomes more useful when we introduce types which have patterns. One of these is **option types**. The type `int option` has two patterns, meaning that if `v : int option` then `v = None` or `v = Some x` where `x : int`. Below is an example of a function that uses pattern matching to handle these cases of option types.

```
let incrementOption a = match a with
     None -> Some 1
  | (Some x) -> Some (x + 1)

let o1 = incrementOption (None)
let o2 = incrementOption (Some 41)
```

```
val incrementOption : int option -> int option = <fun>
val o1 : int option = Some 1
val o2 : int option = Some 42
```

Below is another example that uses pattern matching to identify cases within a tuple of type `int * string * int` representing a simple mathematical expression. If it is well formed, it returns the result as an option type. Otherwise (i.e. zero division or an unknown operator), it returns None.

```
let simpleEval t = match t with
    (x, "+", y) -> Some (x + y)
  | (x, "-", y) -> Some (x - y)
  | (x, "*", y) -> Some (x * y)
  | (_, "/", 0) -> None
  | (x, "/", y) -> Some (x / y)
  | _ -> None

 let e1 = simpleEval (15, "+", 112)
 let e2 = simpleEval (7, "/", 0)
 let e3 = simpleEval (100, "/", 4)
 let e4 = simpleEval (8, "$", 6)
```

```
val simpleEval : int * string * int -> int option = <fun>
val e1 : int option = Some 127
val e2 : int option = None
val e3 : int option = Some 25
val e4 : int option = None
```

4. **Algebraic Datatypes**

Algabraic datatypes are "composites" of other types. You have already seen two types of algabraic datatypes: tuples and option types.

Another is lists. Int lists have two patterns: `[]` (a.k.a. `nil`), the empty list, and `x::rest`, where `x : int` is the first element of the list and `rest : int list` is the remaining elements of the list. We can think of the list `[1; 2; 3]` as `1::2::3::[]`. Below is a function that uses pattern matching on lists to obtain the sum of a list.

```
let rec sum l = match l with
    [] -> 0
  | (x :: rest) -> x + (sum rest)

let s1 = sum []
let s2 = sum [1; 5; 1; 1; 2]
```

```
val sum : int list -> int = <fun>
val s1 : int = 0
val s2 : int = 10
```

Another example: this function takes in a list in the form `[a; b; c; d; ...]` and returns `[a*b; c*d; ...]`

```
let rec multiplyAdjacent l = match l with
    [] -> []
  | [x] -> [x]
  | (x::y::rest) -> (x * y)::(multiplyAdjacent rest)

let a1 = multiplyAdjacent [1; 2; 3; 4; 5; 6; 7; 8, 9]
```

```
val multiplyAdjacent : int list -> int list = <fun>
val a1 : int list = [2; 12; 30; 56; 9]
```

You can also define your own algabraic datatypes. Consider the `point` type defined below. It has four patterns: `Origin`, `Polar (radius, angle)`, `Cartesian(x, y)`, and `Midpoint (p1, p2)`:

```
type point = Origin
           | Polar of float * float
           | Cartesian of float * float
           | Midpoint of point * point
```

(Notice that one of these patterns, `Midpoint`, takes in values of type `point`. This means that the type is a recursive type, just like lists!)

We can pattern match on values of type `point` just like with tuples or lists or option types. The example below takes in a point and returns its distance to the origin:

```
let rec distance (p : point) : float = match p with
    Origin -> 0.0
  | Polar (radius, _) -> radius
  | Cartesian (x, y) -> Float.sqrt (x *. x +. y *. y)
  | Midpoint (p1, p2) -> ((distance p1) +. (distance p2)) /. 2.0

let d1 = distance Origin
let d2 = distance (Polar (4.5, 2.1))
let d3 = distance (Cartesian (3.0, 4.0))
let d4 = distance (Midpoint (Origin, Polar (2.0, 0.0)))
```

```
val distance : point -> float = <fun>
val d1 : float = 0.
val d2 : float = 4.5
val d3 : float = 5.
val d4 : float = 1.
```

5. **Higher Order Functions**

Before moving on, its worth noting that you can define a function without binding it to a name. This kind of expression is known as a lambda function. In the example below, a function is defined with the syntax `fun x -> e` and immediately called on an input, without ever storing the function with a variable name.

```
let y = (fun x -> x * x - 1) 5
```

```
val y : int = 24
```

It is possible to define functions that take in a function or return a function (these are called **Higher Order Functions**). The example below takes in a function f of type `int -> int` as well as an int x, and calls `f` on `x` twice:

```
let applyTwice ((f : int -> int), (x : int)) : int = f (f x)
let a = applyTwice ((fun x -> x + 1), 5)
let b = applyTwice ((fun x -> x * 2), 8)
```

```
val applyTwice : (int -> int) * int -> int = <fun>
val a : int = 7
val b : int = 32
```

Its also possible to have functions take in their arguments differently. So far, we have had functions with multiple arguments take them in as a tuple. Its possible to instead use a trick called **currying** which allows the function to take in its arguments 1-at-a-time. If a curried function `f` has two inputs `x` and `y`, and it is called only with `x`, then it will return a function that takes in a value for `y` before returning the output.

The example below does the same as `applyTwice`, except that it is curried and it calls `f` on `x` three times. Notice that the function can be called with both inputs at once, or with only one input in which case it returns a function.

```
let applyThrice (f : int -> int) (x : int) : int = f (f (f x))
let addThree = applyThrice (fun x -> x + 1)
let a = addThree 10
let b = applyThrice (fun x -> x * 2) 10
```

```
val applyThrice : (int -> int) -> int -> int = <fun>
val addThree : int -> int = <fun>
let a : int = 13
let b : int = 80
```

The example below extends this by taking in an integer n and returning a function that applies `f` n times:

```
let rec applyN (f : int -> int) (n : int) =
  if n = 0
  then fun x -> x
  else fun x -> f (applyN f (n - 1) x)
let x = applyN (fun x -> x * 2) 4 10
```

```
val applyN : (int -> int) -> int -> int -> int = <fun>
val x : int = 160
```

There are also several built-in Higher Order Functions. They are best explained by observing what they do in the examples below:

```
let l = [1; 2; 3; 4; 5; 6; 7; 8]
let m = List.map (fun x -> x * x) l
let n = List.filter (fun x -> (x mod 3) = 2) l
let o = List.fold_left (fun x y -> x + y) 0 l
```

```
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8]
val m : int list = [1; 4; 9; 16; 25; 36; 49; 64]
val n : int list = [2; 5; 8]
val o : int = 36
```

6. **Polymorphism**

It is possible to write generic functions in OCaml that operate on values of more than one type. Consider taking the length of the list. It would suck if you had to write a function `intListLen : int list -> int` for int lists, another `stringListLen : string list -> int` for string lists, and so on for every type. Instead, OCaml has one length function `List.length : 'a list -> int` where `'a` (pronounced "alpha") stands in for *any* type. Consider the example below:

```
let rec doubleList l = match l with
    [] -> []
  | (x::rest) -> x::x::(doubleList rest)

let d1 = doubleList [1; 2; 3]
let d2 = doubleList [(3, 'a'); (5, 'b')]
```

```
val doubleList : 'a list -> 'a list = <fun>
val d1 : int list = [1; 1; 2; 2; 3; 3]
val d2 : (int * char) list = [(3, 'a'); (3, 'a'); (5, 'b'); (5, 'b')]
```

When `d1` is evaluated, `'a = int`, but when `d2` is evaluated, `'a = int * char`.

For another example, consider the function below which finds the maximum value of a list according to some comparison metric provided.

```
let rec polyMax (cmp : 'a -> int) default l = match l with
    [] -> default
  | (x::rest) ->
      let res = polyMax cmp default rest in
      if (cmp x) > (cmp res) then x else res

let l1 = [13; -1; 21; 1; -89; 2; -3; -34; 5; 8; 55]
let l2 = ["Oh"; "what"; "a"; "beautiful"; "morning"]
let max1 = polyMax (fun x -> x) 0 l1
let max2 = polyMax Int.abs 0 l1
let max3 = polyMax String.length "" l2
```

```
val polyMax : ('a -> int) -> 'a -> 'a list -> 'a = <fun>
val l1 : int list = [13; -1; 21; 1; -89; 2; -3; -34; 5; 8; 55
val l2 : string list = ["Oh"; "what"; "a"; "beautiful"; "morning"]
val max1 : int = 55
val max2 : int = -89
val max3 : string = "beautiful"
```

Notice how it can find the "largest" value in a list of any type based on whatever metric is provided. All that is required is that the input to the `cmp` function, the default value, and the values within the list all have the same type. Consider the further examples below which use the fact that polyMax is curried.

`largestSum` uses `polyMax` to find the 1D list within a 2D list of integers with the largest sum.
`biggestDifference` uses `polyMax` to find the tuple within a `int * int` list with the largest difference between the two elements.

```
let largestSum = polyMax (List.fold_left (fun x y -> x + y) 0) []
let biggestDifference = polyMax (fun (x, y) -> Int.abs (y - x)) (0, 0)
let max4 = largestSum [[1; 2; 3]; [4]; [-5; 6; -7]];;
let max5 = biggestDifference [(1, 5); (1, 1); (2, 4); (3, 8)]
```

```
val largestSum : int list list -> int list = <fun>
val biggestDifference : (int * int) list -> int * int = <fun>
val max4 : int list = [1; 2; 3]
val max5 : int * int = (3, 8)
```

7. **Combined Example: Files and Folders**

Combining several of the ideas so far, below is an example of a recursive datatype representing files and folders. The type has two patterns: `File` (which contains the file's name and size) and `Folder` (which contains the folder's name and a list of its contents, which could also be files or folders).

```
type storage = File of string * int
             | Folder of string * (storage list)
```

Suppose we wanted to write the listFiles function from lecture. We could do it by casing on whether or not the current object is a File or a Folder. If its a File, return its name. If its a folder, we could get the file names of every file inside the list by recursively calling listFiles on each of them, then add all the lists together, then put the current folder name in front of each file name. Those operations are done in three steps:

- Use `map` to apply the listFiles function recursively to each item in the list
- Use `List.fold_left` to concatenate the lists (with the `@` operator)
- Use `map` to pre-pend each string with the folder name

```
let rec listFiles s = match s with
    File (name, size) -> [name]
  | Folder (name, contents) ->
      let subLists = List.map listFiles contents in
      let combined = List.fold_left (fun x y -> x @ y) [] subLists in
      let prefixed = List.map (fun x -> name ^ "/" ^ x) combined in
      prefixed

let fi1 = File ("Hello.py", 42)
let fi2 = File ("World.cpp", 100)
let fi3 = File ("Yay.txt", 15)
let fi4 = File ("Foo.txt", 112)
let fo1 = Folder("B", [fi1; fi2])
let fo2 = Folder("C", [fi4])
let fo3 = Folder("A", [fo1; fi3; fo2])
let paths = listFiles fo3
```

```
val listFiles : storage -> string list = <fun>
...
val paths : string list =
    ["A/B/Hello.py"; "A/B/World.cpp"; "A/Yay.txt"; "A/C/Foo.txt"]
```

8. **Other Stuff**

Some other things worth looking into if you want to know more about OCaml:

- Records
- Exceptions
- Modules (Structures, Signatures and Functors)
- Lazy programming
- Imperative programming in OCaml (semicolons, printing, references, loops, arrays)

The OCaml documentation **(linked here)** explains a lot of these quite well.