Chapter 1

OCaml

> We will do so much Ocaml, you could call it Ocamlot
>
> Cleff

This is another programming language chapter so it has two (2) main things: talk about some properties that the OCaml programming language has and the syntax the language has. If you want to code along, all you need is a working version of OCaml and a text editor. You can check to see if you have OCaml installed by running `ocaml -version`. At the time of writing, I am using Ocaml 4.14.0. `ocaml` is repl which you can use to play around with OCaml but please use `utop`. It's a wrapper for `ocaml` and is much easier to use.

1.1 Introduction

OCaml will probably look (and act) unlike any other language you have come across in the CS department up to this point. This is due to one key difference: OCaml is a declarative programming language, opposed to an imperative language. We will talk about this all in the next section, but for now let's just write our very first program.

```
1 (* helloWorld.ml *)
2 print_string "hello world"
```

Despite this being very simple, we observed five (5) things.

- Comments are surrounded by `(* ... *)`
- no semicolons to denote end of statement*
- `print_string` is used to print things out to stdout
- Parenthesis are optional when calling functions*
- OCaml file name conventions are `camlCase.ml`
- Strings exists in the language (most languages do, but some do not)

Now you may have noticed the $*$ symbol over point 2 and 4. That is because these observations are actually False. Or at least, not entirely true. For now though, let's just roll with it. In order to have the above code run, you can run

```
ocamlc helloWorld.ml
./a.out
```

Congrats, you have just made your first program in OCaml! There are some things to note however:

- OCaml is a compiled Language

- `ocamlc` is the ocaml compiler (some compilers like to take the name of the language and add 'c' to the end: javac, ocamlc, rustc").
- If you run an `ls` you will notice that along with the executable `a.out`, two other files were generated as well, `helloWorld.cmo` and `helloWorld.cmi`. The `.cmo` is the object file and can be thought of as analogous to the `.o` file generated by gcc. The `cmi` file is the interface file and can be thought of as analogous a compiled down `.h` file in c.
- `ocamlc` is wrapped in a nice program called dune. dune will allow you to compile, run, and test your OCaml programs without much overhead. We use dune to help manage your projects.

1.2 Functional Programming

According to Wikipedia, "functional programming is a programming paradigm where programs are constructed by applying and composing functions". This probably means nothing to you, so let's make our own definition. First let's define a few words, or rather one in particular: paradigm. Depending on the field, it has many different definitions. I want to not the lingustic's definitions, despite that is the one used here. I want to take the more general one: a set of thoughts and concepts related to a topic. With this defintion, I will say this: **Functional programming is a way of programming that focuses on creating functions rather than listing out steps to solve a problem.**. I'm sure that many people will disagree and dislike this definition but oh well.

1.2.1 Declarative Languages

Functional languages are typically described as declarative. This means that values are declared and the focus is declaring **what** a solution is, rather than describing **how** a solution is reached. To see this let's do a real quick comparison in English for a process that finds even numbers in a list

Imperative
+ make an empty list called results
+ Look at each item in the list
+ Divide the item by 2 and look at the remainder
+ if the remainder is 0, add the value to the results list
+ return the results list after you looked at all list items

Declariative
+ Take all the values that are even divisible by 2 from the list
+ Return those values

Notice that the imperative instructions tell you *how* to do something and does so in steps. The declarative instructions tells you what you are looking for and assumes you can just figure out how to do it. If we translate the imperative code to Python we get something like

```
1 #imperative.py
2 def evens(arr):
3     results = []
4     for x in arr:
5         remainder = item % 2
6         if remainder == 0:
7             results.push(item)
8     return ret
```

Now we haven't learned enough (or really any) OCaml at this point to write a solution to this yet[1] , but let's look at a declarative python example:

```
1 #declarative.py
2 results = [x for x in arr if x % 2 == 0]
```

Here, we don't tell python *how* to solve the problem, we tell it what we want and python figures out the rest.

[1] If you wanted a solution here is one:
`let rec even lst = match lst with []-> []|h::t -> if h mod 2 == 0 then h::(even t) else (even t) in even lst`

1.2.2 Side effects and Immutability

One thing that functional programming aims to do is to minimize this idea of a side affect. Consider the following Python Code:

```
# side_effects.py
count = 0
def f(node):
  global count
  node.data = count
  count+=1
  return count
```

Functional programming wants to treat functions as, well, a function. This means that something like the following should be true:

`f(x) + f(x) + f(x) = 3 * f(x)`

However, if we run the code above, then `f(x) + f(x) + f(x) = 1 + 2 + 3` and `3 * f(x) = 3 * 1`. This unpredictability is called a side effect (The true definition of a side effect is when non local variables get modified). When side effects occur, it becomes harder to reason and predict the behaviour of code (which means more bugs!). To combat this, OCaml makes all variables immutable to help maintain **referential transparency**. Referential transparency is the ability to replace an expression or a function with it's value and still obtain the same output. To simplify, OCaml wants to minimize the amount of outside contact your code has to make everything self contained. The more you rely on outside information or context, the more complicated your code becomes. Now we need to address the question, "What is an expression or function's value?"

1.2.3 Expressions and Values

In functional languages, one of the core ideas is ability to treat functions as data. Which means much like in python, we can pass functions in as arguments, or use them as return values to other functions. But we are getting a little ahead of ourselves. Let's first see what data is in OCaml.

In OCaml, we say that almost everything is an expression. Expressions are things like `4 + 3` or `2.3 < 1.5`. We say that expressions evaluate to values. A value is something like 7 or `false`. All values are expressions in and of themselves, but not all expressions are values. Like a square and rectangle situation. All expressions also have a (data) type. The expression 3+4 evaluates to 7 so we say that both expressions have type `int`. For the purpose of these notes I will use e to represent an expression, t for type, and the structure $e : t$ to say that the expression e evaluates to type t. Consider the following:

```
(* expressions.ml *)
true (* is a value, has type bool *)
3 * 4 (* is an expression, has type int *)
"hello" ^ "world" (* is an expression of type string *)
5.4 (* a value of type float *)
```

Now, we said that almost everything is an expression but we do have one thing that I would not consider an expression: the binding of expressions to variables. This can get confusing because there are these things called `let` *`bindings`* and `let` *`expressions`*. We will talk about the former first.

A `let binding` is not an expression and just binds an expression to a variable. Here is an example:

```
(* letBinding.ml *)
let x = 3 + 4
(* syntax *)
(* let variable = e*)
```

It is important to note that OCaml uses static and latent typing. Also recall that variables in OCaml are immutable. When we run the above code in a repl like *utop* we are actually setting a top level variable which can be used to refer in other places. We ultimately want to try and avoid this to maintain more strict referential transparency so we have this expression called a `let expression`.

A `let expression` is like setting a local variable to be used in another expression.

```
(* letExpression.ml *)
let x = 3 + 4 in x + 1
(* syntax *)
(* let variable = e1 in e2 *)
```

In this case, we add the `in` keyword and follow up with another expression. In this case, this is an expression so it does have a value it will evaluate to and also has a type. It's type is dependent on what the second expression's type is.

```
(let variable = e1:t1 in e2:t2):t2
```

We can of course nest these, and since data is immutable in OCaml, variables are overshadowed.

```
(* scoping.ml *)
let x = 3 in let y = 4 in x + y (* 7 *)
let x = 3 in let x = 4 in x (* 4 *)
let x = 3 in let z = 4 + x in let x = 1 in x + z (* 8 *)
(* implicit parenthesis *)
let x = 3 in (let z = 4 + x in (let x = 1 in x + z))
```

Here, whenever we look consider a variable's value, it is always the closest preceding binding. Also, just to reiterate, these are expressions so something like the following is also possible:

```
let x = if true then false else true in let y = 3 + 4 in let z = if true then 2 else 6 in if
    x then y else z
(* implicit parenthesis *)
let x = (if true then false else true) in let y = (3 + 4) in let z = (if true then 2 else 6)
    in (if x then y else z)
```

Now that we have an an idea of what an expression is and how to determine some basic values and types, we can build larger expressions. I think of this as analogous as taking statement variables p and p and then building larger statements like $p \vee q$. And since we know how to evaluate basic expressions and find out their types and values, it's like knowing the truth values of p and q and being able to then conclude the truth value of $p \vee q$.

1.2.4 The if Expressions

Let us consider the very basic `if expression`. Now the `if expression` is an expression which means it has a value and type. But first let us consider it's syntax.

`(`**`if`** `e1:bool` **`then`** `e2:t e3:t):t`

What does this mean? As stated before, I will be using e to represent expressions and t for types, with $e : t$ meaing that e has type t. So in this case, the `if expression` has an expression $e1$ which must evaluate to a `bool`, and two other expressions $e2$ and $e3$ which must **both** evaluate to the same type t. The `if expression` as a whole then has that same type t. A little weird, let's see an example.

```
if true then 3 else 4
```

Here `true` is an expression of type `bool` and both 3 and 4 are expressions of type `int` which means this expression as a whole has type `int`. Now because we can substitute any valid expression for $e1$, $e2$, $e3$ as long as their types follow the above rules all the following are valid expressions:

```
if true then 3 else 4
if true then false else true
if 3 < 4 then 5 + 6 else 7 + 8
if (if true then false else true) then (if false then 3 else 4) else (if true then 5 else 6)
```

Unlike Python or C, the only things that evaluate to `bools` are `true` and `false` or expressions that evaluate to `true` and `false`. So `if 3 then 4 else 5` would be invalid. This idea of substituting any expression with the expected type can be used for any expression that has 'subexpressions'. So the following are valid with `let bindings` and `let expressions`:

```
let x = if true then false else true
let y = 3 + 4 - 10 in if true then y else y + 10
```

1.2.5 Functions as Expressions

We stated earlier that functional programming is one where we want to be able to treat functions as data. We have actually kinda saw this before. Consider the following:

```
x = 3
print(x)

def x():
    3
prints(x())
```

There is not much difference here as what is printed out or how we use the name `x`. In OCaml, I consider variables to be functions with no parameters that return a value very much like our `x` method above. This is because at the end of the day, if we recall out C and 216 days, a variable is just a way to refer to some specific memory address that holds data. That data could be a value, could be code. But an actual function definition looks like this:

```
(* functions.ml *)
let area l w = l * w
(* or to use a let expression where we call the function *)
let area l w = l * w in area 2 3
(* syntax *)
(* (let name e1:t1 e2:t2 ... ex:tx = e:ty):t1 -> t2 -> tx -> ty *)
```

Like `let bindings` a function definition by itself is not an expression. You will need the `in` keyword if you want it to be an expression. The type of a function is represented as a list of types that looks like `t1 -> t2 -> ... -> tx -> ty`, For example `let area l w = l * w in area` has type `int -> int -> int`. The last type in this list is always the return type, where the preceding types are the types for input.

The fun part is that since we know functions are expressions, and functions can take in expressions as input, then we can have functions that take in other functions.

```
(* functional1.ml *)
let area l w = l * w (* int -> int -> int *)
let apply f x y = f x y (* ('a -> 'b -> 'c) -> 'a -> 'b -> 'c *)
apply area 2 3
(* optional parenthesis *)
(* apply (area) (2) (3) *)
```

Now we have some new things to talk about here. Namely, what is `'a, 'b, 'c` and why is `area`'s type `int -> int -> int` and not something like `float -> float -> float`?

1.2.6 Type Inference

Let us consider the `area` function: `let area l w = l * w`. OCaml knows that this is a function with type `int -> int -> int`. Which means we cannot do something like `area 2.3 4.5`. Why is this and how does this work?

Type inference is a way for a programming language to determine the type of a variable or value. In some languages it's real easy because you explicitly declare types: `int x = 3;`. In OCaml, variables types are determined by the operations or syntax of the expression. So just like you can only use things like && and || on `bool`s, we can only use things like `+,-,*,/` on `int`s. If you wanted to do operations on `float`s then you need to use different operators. See the following:

```
(* operations.ml *)
2 + 3 (* int and int *)
1.3 +. 4.3 (* float and float *)
"hello" ^ " world" (* string and string *)
true || false (* bool and bool *)
int_of_char 'a' (* char input, int output *)
2 + 3.0 (* error *)
3 ^ 4 (* error *)
```

Some operators however, work on many different types. One such example is the > (greater than) operator. This operator along with `<, >=, <=, =` all can take in any two inputs as long as those two inputs are the same type. The output will always be of type `bool`.

```
(* compare.ml *)
2 > 4 (* false *)
"hello" <= "world" (* true *)
true = false (* false *)
```

One thing to note is that we use = for testing equality since we bind variables with `let ... = e`. So we can say something like `let x = 2 = 3` and OCaml knows that x is the varaible and anything after the first = sign is the expression. Anyway, this is important because then what type is inferred from a function like

```
(* typeInference0.ml *)
let compare x y = x > y in compare
```

Here we have no idea what type x and y have to be. In this case, OCaml uses a special type notation. The type of `compare` is `'a -> 'a -> bool`. That is, we have two inputs which must both be the same type, and we know the result will be of type `bool`. If we are given something even stranger like:

```
(* typeInference1.ml *)
let f x y = 3
(* this is equivalent to something like
def f(x,y)
    3
end
*)
```

OCaml will give this function type `'a -> 'b -> int`. We are returning an `int` but the input types could be anything. Since the input types don't even need to be the same type here, we give them different symbols. So let's go back to our `apply` function and break it's type down again.

```
(* typeInference2.ml *)
let apply f x y = f x y in apply (* ('a -> 'b -> 'c) -> 'a -> 'b -> 'c *)
```

First let's list out the parameter names: `f, x, y`. The first parameter is a function which has two inputs of unknown type. We also don't know if the two inputs have to be the same or different. At this moment we know that the function's type is `'a -> 'b -> ?`. Looking at the function we are given no information about what the return type of `f` is so we give it yet a another symbol. Thus we can say that the type of `f` is `'a -> 'b -> 'c`. Now we know that `f` is being called on parameters `x` and `y` so we know that `x` must be the type of `f`'s first argument so we can give `x` type `'a`. We then know that `y` is being used as `f`'s second argument so it should be of type `'b`. So at this point we know the type of `apply` is `('a -> 'b -> 'c) -> 'a -> 'b -> ?` (we put any function's type in parenthesis to show it's a function). Lastly we know that the returned value of `apply` is whatever `f x y` returns. Since we know that `f` returns some value of type `'c`, we can say `apply`'s return type is also `'c`. Thus the entire type of `apply` is `('a -> 'b -> 'c) -> 'a -> b -> 'c`

1.3 Ocaml Pattern Matching

The next feature that OCaml allows us to have is the ability to pattern match. Pattern matching is, if you squint, very closely related to a `switch` statement. While I could show you pattern matching with what we know, I find it easier to demonstrate once we know OCaml's built in data structure: `lists`.

1.3.1 Lists

In other languages you may used to having these data structures called Arrays. In OCaml we don't have arrays, we what we have instead is `lists`. Let's first see the syntax:

```
(* list.ml*)
[1;2;3;4] (* type: int list *)
```

```
(* syntax *)
(* [e1:t; e2:t; ... ex:t]*)
```

Looks a little like an Array but instead of being comma delimited, it is semi-colon delimited. Looking at the syntax we can also conclude that the `lists` must be homogeneous. Additionally, we can see that we do not need to put values, but rather we can put expressions. Here are some examples of other lists

```
(* list1.ml*)
[1;2;3;4] (* int list *)
[2.3;1.0] (* float list *)
["hello", "World"] (* string list *)
[2 + 3; 4-4; 7 * 9] (* int list *)
let f1 x = x + 1 in let f2 y = x + 1 in let f3 z = z + 1 in [f1;f2;f3] (* (int -> int) list
    *)
```

The last one is a list of `int -> int` functions, wild huh? Now it is important to note that `lists` are implemented as a linked list under the hood which means they are recursive.

1.3.2 Recursion

Now you may have noticed that I have not talked about `for, while, do while` or any other type of looping structure. That is because it does not exist in OCaml. We have something better: recursion. In OCaml, data structures are recursive and to do looping, we need to make recursive functions. We will talk about this is a bit. We first need to talk about lists. Now since `lists` are recursive, we need to consider how to define a `list`. If you recall from 132 your linked list data structure, you should recall that a linked list in Java is a 'node' which contains a piece of data and then points to another list or null. In OCaml, we don't use these words, but they can be used analogously. In OCaml we don't point to Null, but instead we point to an empty `list` which we call `Nil`. We then have an element which points to the rest of the list, which we use what we call the `Cons` operator.

```
[] (* Nil, the empty list *)
1 :: [] (* 1 cons Nil, add 1 to the empty list. Evaluates to [1] *)
1 :: 2 :: [] (* 1 cons 2 cons NIL. Evaluates to [1;2] *)
1 :: [2] (* 1 cons list of 2. Evaluates to [1;2] *)
(* syntax *)
(* e1:t :: e2:t list *)
```

Notice that the syntax shows that we are using expressions which means we have some wierd expressions that represent `lists`. Also note that the `cons` operator's left hand operator is of type `t` and the right hand operator must be of type `t list`.

```
[2 + 3; 4 - 5] (* int list. Evaluates to [5;-1] *)
[if true then false else true; false;] (* bool list. Evaluates to [false;false] *)
[[1;2;3];[4;5;6]] (* int list list. Is a value *)
[print_string "hello"; print_string "world"] (* Unit List. Don't worry about Unit now, but
    notice that "worldhello" is printed. *)
```

Notice that when we put expressions into lists, they are evaluated and not stored as expressions, but also evaluated from right to left order (see the last example).

1.3.3 Pattern Matching

So now that we have an idea of what a `list` is and how to construct one, now we have to learn how to deconstruct a `list`. Pattern matching is the way to deconstruct any data structure in OCaml and is a language feature not found in all languages. In order to pattern match, we need to learn a new expression: the `match` expression.

```
let x = 5
match x with
    0 -> 0
```

```
4       |1 -> 1
5       |3 -> 2
6       |5 -> 3
7       |_ -> 4
8   (* Syntax *)
9   (* (match e1:t1 with
10       pattern1 -> e2:t2
11      |pattern2 -> e3:t2
12      | ...   ):t2
```

A `match` expression takes in an expression/value and then checks to see if it has the same structure as any of the cases. If it matches with a case, it will then preform the expression linked to the case. The last line is an underscore, which is used as a wildcard (match with anything else). Here is an analogous switch statement:

```
switch(5){
    case(0): return 0;
    case(1): return 1;
    case(3): return 2;
    case(5): return 3;
    default: return 4;
}
```

I don't want you to think of them as the same though, and pattern matching can do a lot more than a switch statement so just use this vaguely related but not the same. However like a switch statement, a match statement will check until the first pattern that satisfies the requirements and then not look at any of the other patterns. Additionally, notice the match expression is an expression which means it can be evaluated to a value and has a type. It's type is whatever each case returns, and so we need each branch to have the same return type. The next thing to discuss is the idea of a *pattern*. A pattern is not like a regular expression pattern, but it matches with how a piece of data could be represented. Consider all the ways we can represent a list of 2 items. We can use each of these in a math expression and they mean the same thing.

```
1   match [1;2] with
2       a::b::[] -> 0
3       |a::[b] -> 1
4       |[a;b] -> 2
```

In the above example, the expression evaluates to 0, but all of those patterns mean the same thing. Here is an example of pattern matching on a list where we return the length or 4 if longer than 3 elements.

```
1   (*let lst = ... some list *)
2   match lst with
3       [] -> 0
4       |[a] -> 1
5       |a::b::[] -> 2
6       |a::[b;c] -> 3
7       |h::t -> 4
```

In this example we are matching some previously defined list named `lst` and seeing if the structure is anything like what we have on lines 3-7. If we take a look at line 7, we will see this pattern `h::t`. Remember our syntax of a list: `e1:t :: e2:t list`. This pattern is just a single value cons to some list of some arbitrary size. Ultimately, as long as a list's size is greater than 1, this pattern would match, but since it's the last item, it will only be reached if the preceding patterns do not match.

1.3.4 Recursive Functions

Knowing all this, we can then make functions that find the head of a list, or the last item of a list, but first remember what I said earlier, there is no looping construct except recursion. So we need to make recursive functions. To make a recursive function is the same as how we construct any other function but we need the `rec` keyword. Let's unalive 2 creatures with 1 weapon:

```
(* Assume the list cannot be empty *)
let rec tail lst = match lst with
    [x] -> x
    |_::t -> tail t
```

First, notice a recursive function is similar to a normal function. We just need the `rec` keyword after the `let` keyword. Next, let's consider the patterns I used. If we assume the list is not empty, then the base case is a list with 1 item. In this case, just return that 1 item. Otherwise, if the list takes the form of `_::t`, or something cons list, then just recursively call `tail` on the rest of the list. Notice that since I did not need the head item, I did not need to bind it to a variable, so I could just use the wildcard character. Another recursive function example: sum up the values in an `int list`.

```
let rec sum lst = match lst with
    [] -> 0
    |h::t -> h + sum t
```

There are other data types that exist besides lists which you can use and pattern match on. Please refer to the Data Types and Syntax section for more (Section 1.4).

1.4 Data Types and Syntax

1.4.1 Data Types

Basic Types

Data Structures

There are 4 main data structures that exist in OCaml. They are

- Lists
- Tuples
- Variants
- Records

. Each one of these things can be pattern matched and used to construct more complicated data structures. However in my experience I have rarely ever used records.

I talked about lists in an earlier section of this chapter so you can refer there for more info but here are some examples.

```
[1;2;3] (* int list*)
[] (* empty list, Nil, 'a lst *)
[2.0 +. 3.4] (* float list *)
let f x = x + 1 in let g y = y * 1 in [f;g] (* (int -> int) list *)
```

Now that we are refreshed on `lists`, let's talk about tuples. Tuples are ways for us to package data together to be a single 'value' so to speak. This can be useful since functions can only have one return value, so if we need to return multiple pieces of data, a tuple could be the way to go. But enough talking, here is an example:

```
(* tuples.ml *)
(3,4) (* int * int *)
(1,2,"hello") (* int * int * string *)
(* syntax *)
(* (e1:t1,e2:t2,...,ex:tx):t1 * t2 * ... tx *)
```

As you can see tuples are just expressions that comma delimited and placed in parenthesis. Some important things to note is that tuples are of fixed size and their type is dependent on the size and types of the subexpressions. For example, `(3,2)` is an `int * int` tuple which is different than `3,2,1)` which is an `int * int * int` tuple. We can pattern match to break apart tuples by using our match expression.

```
(* tuple-match.ml *)
let t = (1,2)
match t with
|(0,0) -> 0
|(1,1) -> 1
|(1,b) -> b + b
|(a,b) -> a * b
```

The next data type we can talk about are variants. These are similar but not the same as enums. They are ways we can give names and make our own types in OCaml.

```
(* variants.ml *)
type coin = HEADS | TAILS
let x = HEADS (* type is coin, value is HEADS *)
```

These types are then recognized by the rest of OCaml and we can write functions based on these types. To figure out what type you are using, you can use pattern matching.

```
(* variants.ml *)
type coin = HEADS | TAILS
let flip c = match c with HEADS -> TAILS | TAILS -> HEADS
(* flip is a function of type coin -> coin *)
type parity = Even | Odd
let is_even p = match p with Even -> true | Odd -> false
(* is_even is a function of type parity -> bool *)
```

These variants are helpful for just renaming or making data values look pretty. For each of these examples we could have just used bools or ints to represent data. However, variants also allow us to store data in our custom types. Consider the following:

```
(* variants.ml *)
type shape = Rect of int * int | Circle of float
let r = Rect 3 4 (* type is shape, value is Rect(3,4) *)
let c = Circle 4.0 (* type is shape, value is Circle(4.0) *)
```

Here I am saying to make a shape type and that shapes can either be `Rects` which hold `int * int` tuple information, or `Circles` which hold `floats`. We can pattern match to figure out what type we are talking about, and to pull out information.

```
(* variants.ml *)
type shape = Rect of int * int | Circle of float
let r = Rect 3 4 (* type is shape, value is Rect(3,4) *)
let c = Circle 4.0 (* type is shape, value is Circle(4.0) *)
let area s = match s with
Rect(l,w) -> float_of_int l * w (* need to cast this to float so return types match *)
|Circle(r) -> r * r * 3.14
```

This is useful if we want to make Trees or our own lists.

```
(* variants.ml *)
type int_tree = Node of int * int_tree * int_tree | Leaf
let t = Node(4,Node(3,Node(2,Leaf,Leaf),Leaf),Node(5,Leaf,Leaf))
(* tree that looks like
     4
    / \
   3   5
  /
 2
*)
```

1.4.2 Syntax