Appendix A

OCaml Walkthrough

> If it's called Ocaml, why don't they use Camel Case?
>
> Anonymous Student

This document is made so you can follow through examples and see some remarks. I would suggest you take the examples here, modify and play around with them yourself. This was made primarily because classes have been canceled due to weather. Why not a video you may ask? If changes need to be made, it would be easier to do instead of making a whole new video every time. Also, working with code examples, written format is probably better. Also this forces you to at least copy and paste things to play around with things as opposed to watching me code.

A.1 Getting Started

There are two ways you can go about this. You can either use `utop` directly, or (what I would suggest as things get more complicated), writing in a file that you then can import into `utop`. If you wish to use the file method, place your code in a file, say "myfile.ml". Then in `utop` you can type `#use "myfile.ml"` (don't forget the `#` symbol) For example:

The file `myfile.ml`:

```
(* myfile.ml *)
42
```

In `utop`:

```
utop # #use "myfile.ml";;
- : int = 42
```

You must reload the file using the `#use` directive every time you modify the file.

Important Note: Because whitespace and newlines do not affect OCaml execution, you must end any command in `utop` with `;;`.

A.2 Expressions, values and Basic Data Types

Let's start with understanding the output o f`utop`. We can use the example from above: `42`. When we type this into utop (adding `;;`) or into a file which we load form `utop`, we should get the following output:

```
- : int = 42
```

As you probably deduced, the output tells the value we entered and the data type it concluded the value to be. So we can try a few values:

```
true
- : bool = true
2.0
- : float = 2.
"hello"
- : string = "hello"
'a'
- : char = 'a'
```

This seems right, but a little misleading. We stated that in OCaml, things are *expressions*, and expressions evaluate to *expressions*. So what happens if we put in an expression that is not a value? Let's try `42 + 42`:

```
42 + 42
- : int = 84
```

In this case, we still get the type OCaml concluded this expression is, and we get the value the expression evaluated to. Hence why we say values are expressions, or rather expressions that evaluate to themselves.[1]. We can try a few more expressions:

```
true || false
- : bool = true
"hello" ^ " world"
- : string = "hello world"
1.0 + 2.6
Error: the constant 1.0 has type float but an expression was expected of type int
```

A.3 Typing of Functions

Let's take another look at the error we got in the last section:

```
1.0 + 2.6
Error: the constant 1.0 has type float but an expression was expected of type int
```

We know that math that functions are things that take in input and return output. We know from C and Java that we typically have to know the input and output type. Here is a Java example,

```
1 int myfunc(char x, double y){ //returns int, input of char and float
2     return (int)y + (int)x;
3 }
```

This function has inputs of `char` and `double` and returns an `int`.Had we tried to use values of not this type, we would get a compile error. The same is true with OCaml. All this means is that built in functions (and user defined functions) all have data types that must be adhered to. So let's go back to the error we got:

```
1.0 + 2.6
Error: the constant 1.0 has type float but an expression was expected of type int
```

For some reason, when using the `+` function, OCaml is expecting the `1.0` to be an int, not a float. This is because in OCaml, unlike other languages, you cannot use `+` on non-ints. Consider the following errors:

```
1.0 + 2
Error: the constant 1.0 has type float but an expression was expected of type int
1 + 2.3
Error: the constant 2.3 has type float but an expression was expected of type int
2 + 3
- : int = 5
```

[1] From an ontological viewpoint they are different. The value `42` is different from the expression `42`, and we would say the latter evaluates to the former. But from a lexicographical view we use them interchangeably.

We can also verify this by checking the type of the function directly:

```
(+)
- : int -> int -> int = <fun>
```

This tells us the type and the value however there is some new things to consider. This is a function, which is denoted by `<fun>`. All functions show this, because there is no real other way to denote the value the function is. As for the type, a function's type is defined by the input and the output and given in what looks like a list like format. The last item of the list will always be the return type, while everything else is types of the input(s) in the order they are defined. So in this example, the `+` function has two inputs, both ints, and one output, which is an int. If we wished to add floats together, we would have to use a different function. This function is `+.`.

```
1.1 +. 2.04
- : float = 3.14
(+.)
- : float -> float -> float = <fun>
```

We can see we also get an error when we don't use floats:

```
1 +. 2.3
Error: The constant 1 has type int but an expression was expected of type float
Hint: Did you mean 1.?
1.1 +. 2
Error: The constant 2 has type int but an expression was expected of type float
Hint: Did you mean 2.?
```

We can also test the previous used example of string concatenation:

```
(^)
- : string -> string -> string = <fun>
```

A.4 Let Bindings

Now if we wanted to make our own functions, we can use a let binding to do so.

```
let myfunc x y = int_of_float y + int_of_char x
val myfunc : char -> float -> int = <fun>
```

Now we have a variable `myfunc` which OCaml has denoted with `val` and we have the type: `char -> float -> int`. We now know that this function returns `int` (the last part of the list), and the first input is `char` and the second input is `float`. We can make a different function with a different type and different number of inputs:

```
let myotherfunc z = string_of_float (z *. 3.4)
val myotherfunc : float -> string = <fun>
```

What happens if we make a function with no input?

```
let zeroinput = 4 + 5
val zeroinput : int = 9
```

In this case, we are not actually defining a function. We are defining a variable. When defining a function verses defining a variable look similar however they are different in terms of when things are evaluated. Everything in OCaml is an expression which means it evaluates to a value. However, there is a bit of wiggle room as to *when* this evaluation occurs. Let's consider the `print_string` function which we saw in the hello world example in the notes.

```
print_string "hello\n"
hello
- : unit = ()
```

Ignore Unit for now, we will come back to it. But we can see that when we evaluate this `print_string` expression, hello is printed out. So let's bind this expression to a variable, and use this expression inside a function.

```
let myvar = print_sting "hello\n"
hello
val myvar : unit = ()
let myprintfunc _x = print_string "hello\n"
val myprintfunc : 'a -> unit = <fun>
```

First notice that `"hello"` is only printed when we bind to a variable, not the function. This makes sense because we don't actually want to evaluate or run the body of a function until the function is called. But when we are making a variable, we want to evaluate what the variable should be bound to.

```
myprintfunc 2
hello
- : unit ()
myprintfunc true
hello
- : unit ()
myvar
- : unit ()
```

Notice that `print_string` is re-evaluated each time the function is called. However it is not re-evaluated when the variable is used. This is because `print_string` is evaluated and the return value of the `print_string` function is bound to the variable. Since the return type is bound to the variable, there is no reason to re-calculate the value when we use the variable. Before we talk more about that I want to address the `'a` data type.

A.5 Type Inference

The `'a` data type represents a polymorphic type. It is typically used when OCaml cannot determine the type of the input used for a function. Notice that in all the examples, we never had to say what was an int, what was a float, etc. This is because OCaml infers the type of variables and values. How OCaml does this is a future lecture, but let's consider the following:

```
let mytypedfunc x y = x + y
val mytypedfunc : int -> int -> int
```

OCaml determines that for this to compile, because you are using the known typed function of `+`, both `x` and `y` must be ints. If we consider the other function

```
let myprintfunc _x = print_string "hello"
val myprintfunc : 'a -> unit = <fun>
```

OCaml does not have enough information about what `x` must be (because we don't use it).

A.6 Referential Transparency and Side Effects

Now that we know about type inference, we can go back to talk about the `unit` type and why the when of evaluation is different in our previous examples. One aspect of functional programming is that functions should be, well, functions. We know that a function is something that takes in input and return output. But the other quality a function has is determinism. A function cannot have 2 different outputs for the same input.[2]. This is really helpful when writing proofs, and substituting in mathematics. Imagine if we say

[2]The square root function is different from the square root relation for this very reason

something like `f(3) + 5`. If we know `f(3) = 4`, then it should follow that `f(3) + 5 = 4 + 5 = 0`. If we said otherwise, then something went very very wrong. This property of being able to replace functions with their result and not have any change in behavior is called referential transparency.

So if we know that `myvar = print_string "hello\n"`, and that `print_string "hello\n" = ()`, then it should follow that `myvar = ()`. In this case, there should be no reason to recalculate `print_string "hello\n"` everytime we want to use `myvar`.

This is different from `f(x) + 5`. In this case, we cannot say what this value is until we know what `x` is. So because `myprintfunc x` is dependent on `x`, we cannot know what should be calcualted until we know what `x` is. Thus, we cannot evaluate the body of a function (and consequently the `print_string` function in the body) until the function is actually called with an input.

If we wanted to have a function that has no input, or no output, (like `void` or `null` in other languages), OCaml uses what is called `unit`. `unit` is the type name, whereas the lexicographical representation is `()`. The `print_string` function then, returns void but has what we call a side effect, of printing it's input to stdout. Typically to denote side effects, we use `unit`.

A.7 Expressions Continued

So now that we know that functions have a type that must be followed, that OCaml uses type inference, and that functions can only have one output per input, we can see how this works with more complicated expressions. However before we begin, we need to explicitly talk about expressions, types and how they combine.

As we have been seeing, expressions have types, and are evaluated to values which also have types. We also know that functions expect particular inputs and return a particular type. When thinking about how to combine or write larger OCaml expressions, we can consider the following:

```
1 + 2 + 3
- : int = 6
```

Let's break this down to smaller parts so we can see how all of what we learned plays a part.

First, we know the `+` function expects two inputs, both ints, and returns an int type. Second, we know that expressions evaluate to values which have a type. So let's consider: `1 + 2 + 3` is the same as `(1 + 2) + 3`. We know that the expression `1 + 2` consists of the `+` function that takes in two inputs and returns an output. We also know that `1` and `2` are values which are ints and we can say that `1 + 2 = 3` (and `3` is an int) If `1 + 2 = 3` then `1 + 2 + 3 = (1 + 2) + 3 = 3 + 3`. We now have a new expression of `3 + 3` which also uses the two input function of `+` and returns an int. Knowing basic maths says that `3 + 3 = 6`. So the entire expression `1 + 2 + 3 = 6` which is an int.

We can use this to calculate the evaluated value of the expression `1 + 2  + 3 + 4`. `1 + 2 + 3 + 4` is the same as `(1 + 2 + 3) + 4` and we just calcualted that `1 + 2 + 3 = 6` so using subsitution, we get `6 + 4` which we can then evaluate to `10`. To be very explicit, we can say the following:

```
(e1:int + e2:int):int
```

This says that we can add any two expressions `e1` and `e2` together and get an expression of type int as long as `e1` and `e2` are ints themselves. So coupling this with the base case of

```
v:int
```

we can now recursively represent any expression that is a combination of `+`. If we learn more expressions and types, we can make larger expressions.

```
n:int
(e1:int + e2:int):int
```

```
(e1:int - e2:int):int
(e1:int * e2:int):int
(e1:int / e2:int):int
```

Knowing the types of these functions means we can write longer expressions:

```
3 * 4
- : int = 12
6 / 3
- : int = 2
1 + 12
- : int = 13
13 - 2
- : int = 11
(* if above is true, then below follows *)
1 + 3 * 4 - 6 / 3
- : int = 11
```

A.7.1 More functions

So let's expand our repitoir with more functions and expressions. Let's start with comparison:

```
(e1:'a = e2:'a): bool
(e1:'a <> e2:'a): bool
(e1:'a <= e2:'a): bool
(e1:'a < e2:'a): bool
(e1:'a >= e2:'a): bool
(e1:'a > e2:'a): bool
```

The comparison functions (`=,<>,<=,<,>=,>`) take in two inputs and return a bool. The two inputs are polymorphic, but they must be the same type. See below:

```
2 = 3
- : bool = false
"hello" <> "world
- : bool = true
true <= false
- : bool = false
3.14 > 2
Error: The constant 2. has type int but an expression was expected of type float
Hint: Did you mean 2.?
```

When two inputs share the same polymorphic identifier (in this case `'a`), they must be the same type, but what that type is in particular doesn't really matter. Let's see this in action:

```
let difftypes _x _y = 5
val difftypes : 'a -> 'b -> int
let sametypes x y = x <> y
val sametypes : 'a -> 'a -> bool
let diffandsametypes a b c d = (a = c) && (b <> d)
val diffandsametypes : 'a -> 'b -> 'a -> 'b -> bool
```

The first one has two different polymorphic types because OCaml does not have enough information to determine what types `x` and `y` are. The second one has the same polymorphic type because OCaml determines you are comparing the two inputs and you can only compare inputs of the same type. But what the exact types are is still unknown. The last example knows that the first and third input have the be the same because we are comparing them, and the second and fourth inputs must also be the same because we are comparing them. But there is nothing saying that the first and second inputs have to be the same. Additionally, we don't know the particular types of any of the inputs. We can change that with a few new examples:

```
let knowntype x y = x <= (4 - y)
val knowntype : int -> int -> bool
let diffknown a b c d = (a ^ "this" < c) || (b = d)
val diffknown : string -> 'a -> string -> 'a -> bool
```

A.7.2 If Expressions

The if expression is a common expression that you will need. We can take a look at it's breakdown here:

```
(if e1:bool then e2:'a else e3:'a):'a
```

This says there are three expressions (`e1`,`e2`, and `e3`) and three keywords (`if`, `then`, and `else`). The first expresssion must be a bool expression, while the other two expressions must be the same type, though what particular type is not enforced. See the following:

```
if true then 1 else 2
- : int = 1
if false then 1 else 2
- : int = 2
if true then "hello" else "world"
- : string = "hello"
if false then false else true
- : bool = true
if 0 then 1 else 2
Error: The constant 0 has type int but an expression was expected of type
       bool because it is in the condition of an if-expression
if true then 1 else false
Error: The constructor false has type bool but an expression was expected
       of type int
```

So Ocaml will error if the guard (`e1`) is not a bool, and error if the true branch (`e2`) and the false branch (`e3`) are not the same type. We can also see that when the expression is correctly typed, the expression has the same type of its branches and evaluates to the value of the true branch or false branch depending on the guard.

This last part is important because if the if expression is an expression that evaluates to a value and has a type, we can use it wherever we expect an expression of a particular type. For example, the guard must be a bool expression and we saw that `if false then false else true` is an expression that evaluates to true and has type bool. Which means we can nest it.

```
if false then false else true
- : bool = true
if (if false then false else true) then 1 else 2
- : int = 1
```

This also applies for the branches as long as the two branches are the same type as well.

```
if (if false then false else true) then (if false then 1 else 2) else 3
- : int = 2
```

Now because an if expression has the guard between the `if` and `then` keywords, the true branch between the `then` and `else` keywords, and the false branch after the `else` keyword, we don't need the parenthesis. White space and newlines don't affect the evaluation in OCaml so we can see the following:

```
if if true then false else true then if true && false then "hello" else "world" else
   if 3 > 4 then "a' else "b"
- : string = "b"
```

Keep in mind that since the if expression is an expression that evaluates to a value and has a type, we can combine it with any other expression where an expression is expected as long as we keep the types correct:

```
let squarebutkeepsign x = x * x * (if x < 0 then -1 else 1)
val squarebutkeepsign : int -> int
squarebutkeepsign 5
- : int = 25
squarebutkeepsign (-5)
- : int = -25
```

A.8 Let Expressions

So far we only talked about let bindings but we can use a let expression which limits the scope of the let binding. When we used the let binding before, we could use the binding anywhere after the binding occurred. However, we may only need the binding to exist for short time or just need to it exist for the purpose of some other nested expression. A let binding takes the form of

```
(let var = e1:'a in e2:'b):'b
```

Let expressions are expressions that evaluate to values and have type so like the if expression and the functions we talked about, we can use them wherever we expect an expression as long as the type is correct. It is important to note that with let expressions (and bindings) that bindings are immutable and when using a variable with the same name as another, you are shadowing it, not updating a binding.

```
let x = 5
val x : int = 5
x
- : int = 5
let x = 4 in x
- : int = 4
x
- : int = 5
```

As you can see, the binding of a variable in let binding only exists in the scope of the expression following the `in` keyword. After the end of the expression, the binding is lost. It is also important to note that during shadowing, if you use a shadowed variable, it will use the closest binding. Consider the following:

```
let x = 6 in let x = x + 4 in x
- : int = 10
let x = 6 in let x = let x = 5 in x + 3 in x
Warning 26 [unused-var]: unused variable x.

- : int = 8
```

In the last case, we can add parenthesis to see the scope of each x binding and why we get a warning of unused variable x.

```
let x = 6 in (let x = (let x = 5 in x + 3) in x)
(*                    |______x = 5_______|
             |__________x = 8 _________________|
or if we gave each x a name it would be the same as *)
let x1 = 6 in (let x2 = let x3 = 5 in x3 + 3) in x2)
```

In the last case, we can see that we have an unused variable `x1` which is what the warning is referring to.

A.9 Compound Data Types I

We saw some basic data types like bool, float, int, char and string. The next thing to talk about is compound data types, or data types that are composed of other data types. The two basic compund data types in OCaml are lists and tuples.

A.9.1 Tuples

A tuple is a collection of values which are packaged together as a single data type. This can be helpful in functions that may need to return 2 values (for example a folding function, but that is a future chapter). Here are some examples of a tuple which has two values:

```
1,2
- : int * int = (1,2)
"hello",3
- : string * int = ("hello",3)
2.1 +. 1.04, if true then false else true
- : float * bool = (3.14, false)
```

Some things to point out, a tuple is an expression which evaluates to a value and has type. The tuple is seperated with comma(s) and the type is denoted with the star symbol (*). It is important to note the length of the tuple and the order is important in determining the type. Consider:

```
if true then (1,true) else (true,2)
Error: The constructor true has type bool
       but an expression was expected of type int

if true then (1,2) else (1,2,3)
Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type int * int
```

Because the if expression requires that both branches have the same type, we can see these are not the same type. But if we get the type correct, we can use a tuple like any other expression that evaluates to a value and has a type.

```
(true,1,"hello") < if true then (false,2,"a") else (true,6,"b");;
- : bool = false
```

It is important to note the "level" the tuple exists at. We can nest tuples with parenthesis but that makes them a different data type:

```
(true,false)
- : bool * bool = (true,false)
1,(true,false)
- : int * (bool * bool) = (1,(true,false))
(1,2,3),("hello",false,3.4)
- : (int * int * int) * (string * bool * float) = ((1, 2, 3), ("hello", false, 3.4))

(1,2) = ((1,2),3)
Error: This expression has type 'a * 'b but an expression was expected of type int
```

A.9.2 Lists

Lists are like tuples in the sense they are are also collections of other data types, but unlike tuples, the length of a list does not impact the type of the list, and lists must be homogeneous. That is, every element in a list must be the same type. Lists are wrapped in square backets `[]` and elements are separated with semicolons(`;`), not commas.

```
[1]
- : int list = [1]
[1;2;3]
- : int list = [1; 2; 3]
[true;false]
- : bool list = [true; false]
[1,true,3]
- : (int * bool * int) list = [(1, true, 3)]
```

The empty list is a list that has an unknown data type associated with it:

```
[]
- : 'a list = []
```

Lists are expressions that evaluate to values and have types, and also defined as such

```
[1 + 2;3 * 4; 5 / 2]
- : int list = [3; 12; 2]
[1.2 *. 3.4; 4.0 -. 3.; if true then 5.3 else 7.4]
- : float list = [4.08; 1.; 5.3]
if [1;2] > [5] then ["a";"b";"c"] else ["d";"e"]
- : string list = ["d"; "e"]
```

There are two main list functions `cons` and `append`. The `cons` opperator will `cons` (or add) a new element to the front of a list. Or rather, because lists are immutable, it will return a list with the new element in the front. Because lists in OCaml are internally represented as linked lists, it's just adding a pointer rather than deep copying the entire list so it's actually a very fast operation.

```
let list1 = [3] in
let list2 = 2::list1 in
let list3 = 1::list2 in
list3 = 1::2::3::[] && list2 = 2::[3]
- : bool = true
```

To be more explicit, the cons operator looks like

```
(e1: 'a :: e2: 'a list): 'a list
```

Based off its type, we can see that cons adds an eleement to a list, it does not combine lists togeher. That does not mean you cannot use it with lists, you just have to be aware of what the type of lists you are playing around with are:

```
[[1;2;3];[4;5];[6;7;8;9]]
- : int list list = [[1; 2; 3]; [4; 5]; [6; 7; 8; 9]]
[-2;-1;0]
- : int list = [-2; -1; 0]

[-2;-1;0] :: [[1;2;3];[4;5];[6;7;8;9]]
- : int list list = [[-2; -1; 0]; [1; 2; 3]; [4; 5]; [6; 7; 8; 9]]
```

This is different from the append function which does combine two lists.

```
(@)
- : 'a list -> 'a list -> 'a list = <fun>

[1;2;3] @ [4;5]
- : int list = [1; 2; 3; 4; 5]
```