WEEKS 3 and 4: OCaml

-- General --
Circle one of the bolded key terms and understand what each one means:

A) OCaml is a **functional / imperative** language
B) OCaml has **implicit / explicit** declarations
C) OCaml is **dynamically / statically** typed

Give an example that demonstrates OCaml's strong emphasis on **immutability**.

Not all variables in OCaml are immutable T / F

-- Expressions and Types --
Equality in OCaml:
= and <> represent **structural equality** and inequality in OCaml
== and != represent **physical equality** and inequality in OCaml

Example:
```
let x = "apple";;
let y = "apple";;
x != y;;          => true          (* be careful with this!!! *)
x <> y;;          => false
```

True or False:

A) If … then … else … in OCaml is an expression because it returns a value T / F
B) Every "if" must have an "else" to compile in OCaml T / F
C) "if 1 = 1 then "success" else false" is a type error T / F

Functions also have types because OCaml is a **high order programming language**!
This means functions are treated as data and therefore can be
- assigned to variables
- passed as arguments to functions
- returned from functions

Types of functions are structured like:
type arg1 -> type arg2 -> … -> type argn -> return type

We can create a function in two ways. We can declare a function anonymously as

```
fun arg1 arg2 ... argn -> body
```

or bind our function to a name with

```
let name arg1 arg2 ... argn = body
```

Examples of functions and their types:

```
fun x -> x + 1                                          int -> int
fun x -> x +. X                                       float -> float
fun x -> x ^ “somestr“                              string -> string
```

Explanation: (+) in OCaml is a itself a function that takes two int arguments and returns an int while (+.) is specific to float arguments. Therefore, these operators allow OCaml to make a type inference of int or float. The last example demonstrates string concatenation.

```
fun x y -> if x then y else 10                    bool -> int -> int
fun x y -> if x = y then “y” else “n”             ‘a -> ‘a -> string
fun x y z -> if x = y then z else z             ‘a -> ‘a -> ‘b -> ‘b
```

Explanation: when the type of an argument can be anything, we represent its type as a generic ‘a. In the second example, x and y must be ‘a because they both must be of the same type to compare and equality gives us no hints as to which type it could be. We introduce ‘b for types that also cannot be inferred, but may be different from another polymorphic type since they have no dependency on each other. In the third example, we cannot infer the type of z but also cannot say it is ‘a because it may not be the same type as x and y.

```
fun x -> (x 1) + 1                                (int -> int) -> int
fun x y -> (x y) + 1                          (‘a -> int) -> ‘a -> int
fun x y -> (x y)                                (‘a -> ‘b) -> ‘a -> ‘b
```

Explanation: in each of these examples, x is being treated as a function because it is being passed an argument. The type of x in each example is therefore parenthesized to indicate that it is a function. For the last example, x takes in some unknown type as its only argument and returns some unknown type because there is nothing to infer. Thus, the type of x is (‘a -> ‘b). Notice the return may or may not be different from the type of its argument. Then the type of the function continues as (‘a -> ‘b) -> ‘a -> ‘b because y must be the corresponding ‘a type that is being passed into x and the result of the function call to x is also the result of the anonymous function ‘b.

Lists
→ create with square brackets
→ homogenous (can hold one single type)
→ insert by cons (::) operator
→ elements separated by semicolons

[1; 2; 3] ⇒ int list

Tuples
→ like lists, but are heterogeneous (can hold multiple types)
→ more like hashes in that we can store elements in pairs
→ not limited to just two elements however
→ elements separated by commas

(1, “two”, 3.0) ⇒ int * string * float

Records
→ similar to C structs
→ like tuples except can give each type a name
→ defined with curly braces
→ access elements with dot notation

type profile = { name : string; age : int }
let p = { name = “Bob”; age = 30 } ⇒ profile
p.name ⇒ “Bob”

Try to determine the types of the following yourself (note these are meant to be a little hard):

```
fun x -> x = 2
fun x -> x::(x^”ok”)::[]
fun x y -> (x, y)::[]
fun x y -> (x, y::[7])
fun x y -> ((x,y),(y,x))
fun (x,y) -> (x 1) +. y
fun x y -> (x y y)
fun x y -> (y (y x))
fun x y z -> (x (y z))
fun x y z -> (x, (x y), (z y))
```

-- Pattern Matching --
Pattern matching allows us to match data by its structure. In essence, we can deconstruct a piece of data such as a list with the following:

```
match lst with
| [] -> ...       (* equivalent to matching an empty list *)
| [h] -> ...      (* equiv to matching a list with exactly 1 ele *)
| h::t -> ...     (* equiv to matching a list with at least 1 ele *)
| a::b::c -> ...  (* equiv to matching a list with at least 2 ele *)
| a::b::[] -> ... (* equiv to matching a list with exactly 2 ele *)
| _ -> ...        (* matches anything *)
```

Can match other data types too such as tuples.
Suppose we had a tuple of lists such as ([1;2], [3;4])

```
match tup with
| ([], []) -> ...
| ([], h::t) -> ...
| (h::t, []) -> ...
| (h1::t1, h2::t2) -> ...
```

Notice when we pattern match, we must be exhaustive, meaning we must cover all cases. Otherwise, the compiler will throw an error. In the list example, matching against [] and h::t would cover all cases because the first case tests for an empty list and the second case tests for a list with at least one element.

For map and fold see class_notes section (disc4)

-- Challenge --
Determine what the following function f does.

```
let rec g lst x = match lst with
      | [] -> [x]
      | h::t -> if x > h then h::(g t x) else x::h::t
in
let f lst = fold g [] lst
```

Think about how this would look without fold.
Also think about the time complexity of f.