CMSC330 Fall 2024 Quiz

Proctoring TA: ______________________ **Name:** ______________________

Section Number: ______________________ **UID:** ______________________

Problem 1: Basics

[Total 4 pts]

	True	False
In OCaml, all values are expressions but not all expressions are values everything is an expression, but `2 + 3` is not a value	● T	○ F
In OCaml, all expressions are values but not all values are expressions everything is an expression, but `2 + 3` is not a value	○ T	● F
`map (fun x -> x + 1) a` will modify the list a in-place map doesn't modify anything in place because lists are immutable in OCaml	○ T	● F
Having mutable variables can make it hard to reason about how a program runs Side effects occur when we have mutability, this can be difficult to reason about	● T	○ F
Having immutable variables can make it easy to reason about how a program runs Its very easy to write mathematical proofs about our program if there is no mutability	● T	○ F
A function with type `int -> float -> bool` returns 2 things: a float and a bool Functions return only 1 thing ever	○ T	● F
A function with type `int -> bool -> float` could be interpreted as returning a `bool -> float` function Currying allows for this interpretation	● T	○ F
`let f x = x + 3` is an example of a higher order function this function has type `int -> int` so it is not	○ T	● F
`let f x = x 3` is an example of a higher order function because we use x as if it was function name, OCaml will say this is a `(int -> 'a) -> 'a` type	● T	○ F
An OCaml function can return different types depending on how it's called A function can only return 1 type, (or 1 polymorphic type)	○ T	● F
`let x = 3 in let x = 4 in x` is an example of variable shadowing This returns 4 and variables are immutable so shadowing does occur	● T	○ F
`let x = 3 in let y = 4 in y + x` is an example of variable shadowing there is no two variables with the same name so no shadowing occurs	○ T	● F
`let x = 3 in let y = 4 in y` is an example of variable shadowing there is no two variables with the same name so no shadowing occurs	○ T	● F

Problem 2: OCaml Typing and Evaluating

[Total 6 pts]

Give the type for the following functions `f` and give what the following function call evaluates to. **If there is a type error in the function**, put "TYPE ERROR" for the type, and put "ERROR" for the evaluation. If the function call does not follow the type of `f`, put "ERROR" for the evaluation.

(a) [2 pts]

```
let f x y = match x with
   [] -> []
  |x::xs -> y :: xs ;;

f [] [1;2;3] ;;
```

Type of f: 'a list -> 'a -> 'a list

Evaluation: []

f takes in 2 arguments. We know that x will be a list, and y will be an element of x. We have no operations that force a type. Hence we get `'a list -> 'a -> 'a list`

(b) [2 pts]

```
let f x y = match x with
   [] -> [1]
  |x::xs -> y :: xs ;;

f [] [1;2;3] ;;
```

Type of f: int list -> int -> int list

Evaluation: ERROR

f takes in 2 arguments. We know that x will be a list, and y will be an element of x. We know that `y::xs` will have to match the type of `[1]` so we get `int list -> int -> int list`

(c) [2 pts]

```
let f x y = match x with
   [] -> [14]
  |x::xs -> y :: xs ;;

f [1] [1;2;3] ;;
```

Type of f: int list -> int -> int list

Evaluation: ERROR

f takes in 2 arguments. We know that x will be a list, and y will be an element of x. We know that `y::xs` will have to match the type of `[14]` so we get `int list -> int -> int list`

(d) [2 pts]

```
let f x y = match x with
   [] -> [4]
  |x::xs -> y :: xs ;;

f [] [1;2;3] ;;
```

Type of f: int list -> int -> int list

Evaluation: ERROR

f takes in 2 arguments. We know that x will be a list, and y will be an element of x. We know that `y::xs` will have to match the type of `[4]` so we get `int list -> int -> int list`

(e) [2 pts]

```
let f a b =
    if b <> false then 1.0
    else a + 2.0 ;;

f 2.0 false;;
```

Type of f: ERROR

Evaluation: ERROR

We try adding 2.0 with the + operator, but we need to use the +. operator. This is an error

(f) [2 pts]

```
let f a b =
    if b = "false" then 1
    else a + 2 ;;

f 2 "true";;
```

Type of f: int -> string -> int

Evaluation: 4

f takes in 2 arguments. We know that a will be added to 2, and b is being compared to the string "false" so we get `int -> string -> int`

(g) [2 pts]

```
let f a b =
    if b > 5 then a
    else true ;;

f 2.0 false;;
```

Type of f: bool -> int -> bool

Evaluation: ERROR

f takes in 2 arguments. We know that a will have to match the type of true, and b is being compared to 5 so we get `bool -> int -> bool`

(h) [2 pts]

```
let f a b =
    if b > false then a
    else 2.3 ;;

f 2.0 false;;
```

Type of f: float -> bool -> float

Evaluation: 2.3

f takes in 2 arguments. We know that a will have to match the type of 2.3, and b is being compared to false so we get `float -> bool -> float`

(i) [2 pts]

```
let rec f g lst = match lst with
   [] -> []
  |x::xs -> (x, g x)::(f g xs) ;;

f (fun x -> x mod 2 = 1) [1;2;3] ;;
```

Type of f: ('a -> 'b) -> 'a list -> ('a * 'b) list

Evaluation: [(1, true); (2, false); (3, true)]

f takes in 2 arguments. We know that g is a function that will be called with x so we get `('a -> 'b) -> 'a list -> ('a * 'b) list`

(j) [2 pts]

```
let rec f g lst = match lst with
   [] -> []
  |x::xs -> (g x, x)::(f g xs) ;;

f (fun x -> x mod 2 = 0) [1;2;3] ;;
```

Type of f: ('a -> 'b) -> 'a list -> ('b * 'a) list

Evaluation: [(false, 1); (true, 2); (false, 3)]

f takes in 2 arguments. We know that g is a function that will be called with x so we get `('a -> 'b) -> 'a list -> ('a * 'b) list`

(k) [2 pts]

```
let rec f g lst = match lst with
   [] -> []
  |x::xs -> (g x, x)::(f g xs) ;;

f (fun x -> x +. 2.0) [1.0;2.0;3.0];;
```

Type of f: ('a -> 'b) -> 'a list -> ('b * 'a) list

Evaluation: [(3., 1.); (4., 2.); (5., 3.)]

f takes in 2 arguments. We know that g is a function that will be called with x so we get `('a -> 'b) -> 'a list -> ('a * 'b) list`

(l) [2 pts]

```
let rec f g lst = match lst with
   [] -> []
  |x::xs -> (x, g x)::(f g xs) ;;

f (fun x -> x *. 2.0) [1.0;2.0;3.0];;
```

Type of f: ('a -> 'b) -> 'a list -> ('a * 'b) list

Evaluation: [(1., 2.); (2., 4.); (3., 6.)]

f takes in 2 arguments. We know that g is a function that will be called with x so we get `('a -> 'b) -> 'a list -> ('a * 'b) list`

Problem 3: OCaml Expressions [Total 4 pts]

Write an expression that would have the following types.

(a) [2 pts]

`'a -> 'a -> bool list`

fun a b -> [a = b]

(b) [2 pts]

`('a -> 'a) -> 'a -> int`

fun f a -> if (f a) = a then 3 else 5

(c) [2 pts]

`float -> 'a -> ('a * float)`

fun a b -> (b,a +. 2.)

(d) [2 pts]

`string -> 'a -> ('a * float * 'a)`

fun s a -> (a, (float_of_string s), a)

(e) [2 pts]

`bool -> int -> (bool * int) list`

fun b i -> [b>true,i+3]

(f) [2 pts]

`(int -> 'a) -> 'b -> 'a`

fun f a -> f 3

(g) [2 pts]

`('a -> 'b) -> 'a -> 'b`

fun f b -> f b

(h) [2 pts]

`('a -> 'b -> 'c) -> 'a -> 'b -> 'c`

fun f a b -> f a b

Problem 4: Coding

[Total 6 pts]

Write a function encode that takes a `int list` and returns a `string list`, which consists of the string "1" repeated by each number in the int list. You may assume that all values in the input list are >= 0.

You **do NOT have to use map or fold**, but their definitions are given if you want to use them.
You can write helper methods.

```
(* Examples
    encode [0;1;2;3] = ["";"1";"11";"111"]
    encode [0;0;3] = ["";"";"111"]
*)
```

```
let rec map f l = match l with
   [] -> []
  |x::xs -> (f x)::(map f xs)

let rec fold f a l = match l with
   [] -> a
  |x::xs -> fold f (f a x) xs
```

```
(* Write your code below *)


 let rec encode lst =
    let rec repeat n =
        if n = 0 then
            ""
        else
            "1" ^ repeat (n-1) in
    map repeat lst
```