

CMSC330 Fall 2024 Quiz

Proctoring TA:	Name:		
Section Number:	UID:		
Problem 1: Basics			[Total 4 pts]
In OCaml, all values are expressions but not all expressions are va everything is an expression, but 2 + 3 is not a value	lues	True T	False F
In OCaml, all expressions are values but not all values are express everything is an expression, but 2 + 3 is not a value	ions	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 Side effects occur when we have mutability, this can be difficult to		T	F
Having immutable variables can make it easy to reason about how Its very easy to write mathematical proofs about our program if th		T	F
A function with type int -> float -> bool returns 2 things: a Functions return only 1 thing ever	float and a bool	T	F
A function with type int -> bool -> float could be interpret Currying allows for this interpretation	ed as returning a bool -> float function	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 $f = x$ 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) -> 'atype	1	F
An OCaml function can return different types depending on how it A function can only return 1 type, (or 1 polymorphic type)	s called	T	F
let $x = 3$ in let $x = 4$ in x is an example of variable share. This returns 4 and variables are immutable so shadowing does occ	-	T	F
let $x = 3$ in let $y = 4$ in $y + x$ is an example of variable there is no two variables with the same name so no shadowing oc		T	F
let x = 3 in let y = 4 in y is an example of variable sha there is no two variables with the same name so no shadowing oc		T	F

Problem 2: OCaml Typing and Evaluating

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
                                                                                 'a list -> 'a -> 'a list
    [] -> []
                                               Type of f:
  |x::xs -> y :: xs ;;
                                                                                          []
f [] [1;2;3] ;;
                                               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
                                                                                int list -> int -> int list
    [] -> [1]
                                               Type of f:
  |x::xs -> y :: xs ;;
                                                                                        ERROR
f [] [1;2;3] ;;
                                               Evaluation:
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
                                                                                int list -> int -> int list
    [] -> [14]
                                               Type of f:
  |x::xs -> y :: xs ;;
                                                                                        ERROR
f [1] [1;2;3] ;;
                                               Evaluation:
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]
<pre>let f x y = match x with [] -> [4] x::xs -> y :: xs ;;</pre>	Type of f:	int list -> int list
f [] [1;2;3] ;;	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 list

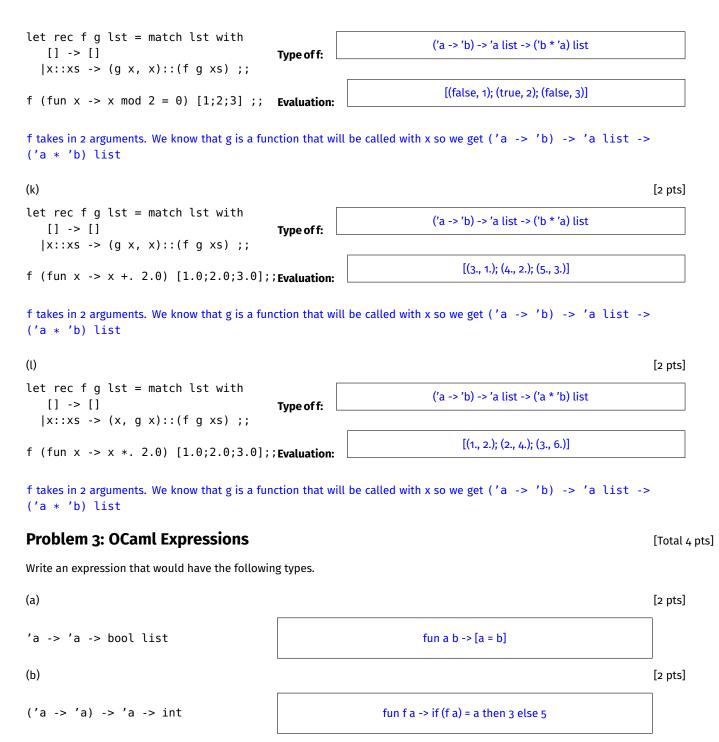
(e)

[2 pts]

<pre>let f a b = if b <> false then 1.0 else a + 2.0 ;;</pre>	Type of f:	ERROR
f 2.0 false;;	Evaluation:	ERROR
We try adding 2.0 with the + operator, but we ne	ed to use the +	•. operator. This is an error
(f)		[2 pts]
<pre>let f a b = if b = "false" then 1 else a + 2 ;;</pre>	Type of f:	int -> string -> int
f 2 "true";;	Evaluation:	4
f takes in 2 arguments. We know that a will be a string -> int	added to 2, and	d b is being compared to the string "false" so we get int ->
(g)		[2 pts]
<pre>let f a b = if b > 5 then a else true ;;</pre>	Type of f:	bool -> int -> bool
f 2.0 false;;	Evaluation:	ERROR
f takes in 2 arguments. We know that a will hav -> int -> bool	ve to match the	e type of true, and b is being compared to 5 so we get bool
(h)		[2 pts]
<pre>let f a b = if b > false then a else 2.3 ;;</pre>	Type of f:	float -> bool -> float
f 2.0 false;;	Evaluation:	2.3
f takes in 2 arguments. We know that a will have -> bool -> float	to match the t	type of 2.3, and b is being compared to false so we get float
(i)		[2 pts]
<pre>let rec f g lst = match lst with [] -> [] x::xs -> (x, g x)::(f g xs) ;;</pre>	Type of f:	('a -> 'b) -> 'a list -> ('a * 'b) list
f (fun x -> x mod 2 = 1) [1;2;3] ;;	Evaluation:	[(1, true); (2, false); (3, true)]
f takes in 2 arguments. We know that g is a fun ('a * 'b) list	nction that wil	l be called with x so we get ('a -> 'b) -> 'a list ->

(j)

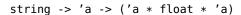
[2 pts]



```
(c)
```

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

(d)

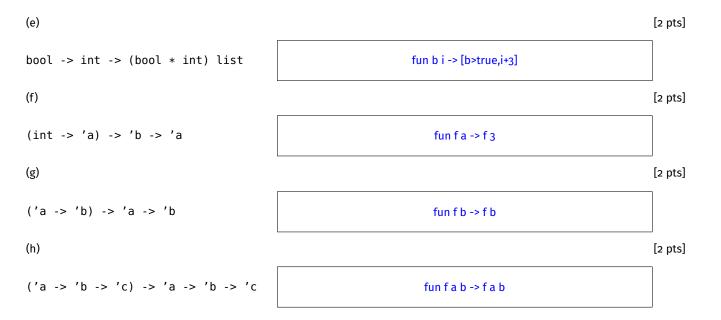


fun s a -> (a, (float_of_string s), a)

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

[2 pts]

[2 pts]



Problem 4: Coding

map repeat lst

[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
                                                   let rec map f l = match l with
    encode [0;1;2;3] = ["";"1";"11";"111"]
                                                      [] -> []
    encode [0;0;3] = ["";"";"111"]
                                                     |x::xs -> (f x)::(map f xs)
*)
                                                   let rec fold f a l = match l with
                                                      [] -> a
(* Write your code below *)
                                                     |x::xs -> fold f (f a x) xs
let rec encode lst =
    let rec repeat n =
        if n = 0 then
            0.0
        else
            "1" ^ repeat (n-1) in
```

```
5
```