



CMSC330 Spring26 Quiz 1

Proctoring TA: _____ Name: Sols

Section Number: _____ UID: _____

Problem 1: Expressions

[Total 6 pts]

Write an expression that does the following. You cannot use type annotations. Try and make the expression as short as possible:

(a) Binds a list containing 3 ints to a variable called myvar

[2 pts]

```
let myvar = [1,2,3] in ...
```

(b) Has type `int -> float -> bool`

[2 pts]

```
fun x y -> (x = 1 && y = 1.0)
```

(c) Contains a shadowed variable named mys

[2 pts]

```
let mys = 5 in let mys = 6 in ...
```

(a) Binds a tuple consisting of 3 different types to a variable called mytup

[2 pts]

```
let mytup = (1,true,2.0) in mytup
```

(b) Has type `float -> bool -> int`

[2 pts]

```
fun f b -> if f = 0.0 && b then 1 else 0
```

(a) Binds a list containing 2 strings to a variable called lstvar

[2 pts]

```
let lstvar = ["a";"b"] in lstvar
```

(b) Has type `bool -> int -> float`

[2 pts]

```
fun i f -> i = 1 && f = 1.0
```

(a) Binds a tuple of length 4 consisting of 2 different types to a variable called tupvar

[2 pts]

```
let tupvar = (1,true,2,false) in tupvar
```

(b) Has type `int -> bool -> int`

[2 pts]

```
fun i b -> if b then i else i + 1
```

Problem 2: Tracing

[Total 9 pts]

Consider the following function mapxmap:

```
let rec mapxmap f1 f2 lst = match lst with
  x::y::xs -> (f1 x)::(f2 y)::(mapxmap f2 f1 xs)
  |x::xs -> [(f2 (f1 x))]
  |x -> x
```

	0	1	2	3
x	1	3	5	7
y	2	4	6	NA
xs	[3;4;5;6;7]	[7]	[]	

The process of performing a code trace helps debugging. If we start with

mapxmap (fun x -> x + 1) (fun x -> x * 10) [1;2;3;4;5;6;7]

then the **first** recursive call of mapxmap would be mapxmap f2 f1 [3;4;5;6;7]. What are the values of x, y, xs **during** the **third** recursive call? If they are not bound during that call, put NA.

x :

y:

xs:

What is the result of calling mapxmap (fun x -> x + 1) (fun x -> x * 10) [1;2;3;4;5;6;7]?

The process of performing a code trace helps debugging. If we start with

mapxmap (fun x -> x - 2) (fun x -> -x) [5;4;3;2;1]

then the **first** recursive call of mapxmap would be mapxmap f2 f1 [3;2;1]. What are the values of x, y, xs **during** the **second** recursive call? If they are not bound during that call, put NA.

x :

y:

xs:

What is the result of calling mapxmap (fun x -> x - 2) (fun x -> -x) [5;4;3;2;1]?

The process of performing a code trace helps debugging. If we start with

mapxmap (fun x -> x mod 2) (fun x -> x * x) [1;2;3;4;5]

then the **first** recursive call of mapxmap would be mapxmap f2 f1 [3;4;5]. What are the values of x, y, xs **during** the **second** recursive call? If they are not bound during that call, put NA.

x :

y:

xs:

What is the result of calling mapxmap (fun x -> x mod 2) (fun x -> x * x) [1;2;3;4;5]?

The process of performing a code trace helps debugging. If we start with

mapxmap (fun x -> x / 2) (fun x -> x + 7) [2;4;6;10;20;30]

then the **first** recursive call of mapxmap would be mapxmap f2 f1 [10;20;30]. What are the values of x, y, xs **during** the **third** recursive call? If they are not bound during that call, put NA.

x :

y:

xs:

What is the result of calling mapxmap (fun x -> x / 2) (fun x -> x + 7) [2;4;6;10;20;30]?

Problem 3: Typing and Evaluation

[Total 6 pts]

Give the type of the function `f`. If there is a type error, write "ERROR". Then, if the function `f` did not have an error, write the result of the function call. If there is an error, write "ERROR"

```
let f x = let y = 7 in x + y
```

Type of f:

`int → int`

```
let x = 20 in f x
```

function call =

`27`

```
let f x y = if x then y else  
           let x = 1 in x + 2
```

Type of f:

`bool → int → int`

```
f false 7
```

function call =

`3`

```
let f x y = match x,y with  
            x::xs,y::ys -> x + y  
            | _ -> -1  
f [1;2;3] (1,2)
```

Type of f:

`int list → int list → int`

function call =

`ERROR`

```
let f x = let y = 7. in x +. y
```

Type of f:

`float -> float`

```
let x = 2.0 in f x
```

function call =

`9.0`

```
let f x y = if x = y then y + 1 else  
           let x = 1 in x + 2
```

Type of f:

`int -> int -> int`

```
f false 7
```

function call =

`9`

```
let f x y = match x,y with  
            x::xs,y::ys -> x +. y  
            | _ -> -1.  
f [1.4;2.1;3.2] (1,2)
```

Type of f:

`float list -> float list -> float`

function call =

`error`

```
let f x = let y = true in x && y
```

Type of f:

`bool -> bool`

```
let x = false in f x
```

function call =

`false`

```
let f x y = if y then x else  
           let x = 1 in x + 2
```

Type of f:

`int -> bool -> int`

```
f 7 false
```

function call =

`9`

```
let f x y = match x,y with  
            x::xs,y::ys -> x ^ y  
            | _ -> ""  
f ["1";"2";"3"] (1,2)
```

Type of f:

`string list -> string list -> string`

function call =

`error`

```
let f x = let y = "true" in x ^ y
```

Type of f:

`string -> string`

```
let x = "false" in f x
```

function call =

`"falsetrue"`

```
let f x y = if y then x else  
           let x = 1. in x +. 2.
```

Type of f:

`float -> bool -> float`

```
f 7. false
```

function call =

`9.0`

```
let f x y = match x,y with  
            x::xs,(y,ys) -> x + y  
            | _ -> -1  
f [1;2;3] (1,2)
```

Type of f:

`int list -> int * 'a -> int`

function call =

`2`

Problem 4: Debugging

[Total 9 pts]

Recall the `max_func_chain` function from exercise one. It has type `'a -> ('a -> 'a) list -> 'a`. It takes in an initial value and a list of functions, and decides to either apply each function or not to maximize the final return value. Below is a buggy implementation of it:

```
let rec max_func_chain init lst = match lst with
1   [] -> []
2   |f::fs -> let choice1 = max_func_chain init fs in
3               let choice2 = max_func_chain (f init) fs in
4               if init > choice1 then
5                   init
6               else if choice2 > init then
                    choice2
                else
                    choice1
```

There are at least 3 bugs. For 3 of them, write down the line number and rewrite the **entire** line to fix it. You can only modify numbered lines.

Line: 4	Fix: <code>if init > choice1 && init > choice2 then</code>
Line: 5	Fix: <code>else if choice 2 > init && choice 2 > choice1 then</code>
Line: 1	Fix: <code>[] -> init</code>

Note: Unless otherwise stated, the following questions cannot use imperative OCaml, nor anything from the List module unless given (see last page).

Problem 5: OCaml Coding 1

[Total 5 pts]

(The following question is on exercise one) Write a function merge (Type: 'a list -> 'a list -> 'a list), which takes two sorted lists, lst1 and lst2, and merges them into one sorted list.

Examples:

```
merge [1] [2] = [1;2]
merge [] [] = []
merge [1; 4] [2; 3] = [1; 2; 3; 4]
merge [1; 4; 5] [2; 3; 6; 7; 8; 9] = [1; 2; 3; 4; 5; 6; 7; 8; 9]
merge [1] [0] = [0; 1]
```

```
let rec merge lst1 lst2 =
```

match lst1, lst2 with

| [], _ -> lst2

| _, [] -> lst1

| h1::t1, h2::t2 -> if h1 <= h2 then h1::merge t1 lst2
else h2::merge lst1 t2

Problem 6: OCaml Coding 2

[Total 5 pts]

The following question is on exercise one. Write a function called zip which will merge two tuple lists, lst1 and lst2, and return the result as one tuple list. Tuples will only contain two elements. The final answer will have a length equivalent to the minimum of the input lengths.

```
(* example
zip [(1, 2); (3, 4); (5, 6)] [(7, 8); (9, 10); (11, 12)] =
  [(1, 2, 7, 8); (3, 4, 9, 10); (5, 6, 11, 12)]
zip [] [] = []
zip [(1, 4)] [] = []
zip [(1, 2); (3, 4)] [(7, 8)] = [(1, 2, 7, 8)]
*)
```

```
let rec zip lst1 lst2 =
```

match lst1, lst2 with

| (a,b)::t1, (c,d)::t2 -> [(a,b,c,d)] @ (zip t1 t2)

| [], _ -> []

| _, [] -> []

Problem 7: OCaml Coding 3

[Total 10 pts]

Write a function called `count_each_occ` which takes in a `'a list` and returns a `('a * int) list` which represents how many times each `'a` item occurs in the list. The resulting list should contain no duplicates. The order of the resulting list does not matter. (hint: `uniq`, `count_occ`, `is_there`, `is_present` from exercise one might help you create a plan but you shouldn't need to recreate all of them). You do not have to use `map/fold`. You can make recursive helpers. You may not use imperative OCaml.

```
(* examples
count_each_occ [1;1;2;3;2;1] = [(1,3);(2,2);(3,1)]
count_each_occ [true;false;false] = [(true,1);(false,2)]
count_each_occ [] = []
*)
```

```
let rec count_each_occ lst =
```

(not rly best way to do thing)
it's okay, see below :)

```
let rec count x lst =
  match lst with
  [] -> 0
```

```
  | h::t -> if h = x then 1 + (count x t) else
              count x t
```

```
in let rec remove x lst =
  match lst with
  [] -> []
```

```
  | h::t -> if h = x then remove x t else h::(remove x t)
```

```
in match lst with
[] -> []
```

```
  | h::t -> let c = count h lst in (h,c)::(count_each_occ (remove h t))
```

~)

```
let count_every_occ lst =
```

```
  fold_left (fun a x ->
```

```
    let updated = fold_left (fun a (v,c) ->
```

```
      if v = x then a@[(v,c+1)] else a@[(v,c)])
```

```
      [] a
```

```
    in
```

```
    if a = updated then (x,1)::a else updated)
```

```
  [] lst
```