1 <3 ocaml

CMSC330 Spring 2025 Quiz 1 Solutions

Problem 1: Basics

[Total 4 pts]

	True	False
In OCaml, the terms values and expressions can be used interchangeably	T	**F**
Using mutable variables can cause side effects	**T**	F
Due to OCaml's type constraints, you cannot make a list of functions	T	**F**
`fold_left`'s accumulator cannot be a tuple	T	**F**
`fold_left`'s accumulator can be a list	**T**	F
`fold_left`'s accumulator can be a tuple	**T**	F
In OCaml, the terms values and expressions cannot be used interchangeably	**T**	F
`fold_left`'s accumulator cannot be a list	T	**F**

Problem 2: OCaml Typing and Evaluating

[Total 6 pts]

Give the type for the following functions `foo` 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 causes an error for any reason, put "ERROR" for the evaluation.

(a) [3 pts]

```
let foo x y =
  match x,y with
    x::xs,y::ys -> y :: xs
  | _ -> [] ;;

foo [] [1;2;3] ;;
```

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

Evaluation: []

(b) [3 pts]

```
let foo lst = let total, _ =
    fold_left
    (fun (tot, idx) ele ->
        (tot + ele * idx, idx + 1))
    (0, 0)
    lst in total;;

foo [3;6;9] ;;
```

Type of foo: int list -> int

Evaluation: 24

(c) [3 pts]

```
let foo x y =
  match x,y with
    x::xs,y::ys -> y :: xs
  | _ -> (0,0) ;;

foo [] [1;2;3] ;;
```

Type of foo: Type Error

Evaluation: Error

(d) [3 pts]

```
let foo lst = let total, _ =
    fold_left
    (fun (tot, idx) ele ->
        (tot*ele*idx, idx + 1))
    (0, 0)
    lst in total;;

foo [7;4;23] ;;
```

Type of foo: int list -> int

Evaluation: 0

(e) [3 pts]

```
let foo x y = match x,y with
    x::xs,y::ys -> y :: xs
  | _ -> [9] ;;

foo [] [1;2;3] ;;
```

Type of foo: int list -> int list -> int list

Evaluation: [9]

(f) [3 pts]

```
let foo lst = let total, _ =
    fold_left
    (fun (tot, idx) ele ->
        (tot + ele * idx, idx + 1))
    (0, 0)
    lst in total;;

foo [2;4;8] ;;
```

Type of foo: int list -> int

Evaluation: 20

(g) [3 pts]

```
let foo x y = match x,y with
    x::xs,y::ys -> y :: xs
  | _ -> [] ;;

foo (1,3) [1;2;3] ;;
```

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

Evaluation: Error

(h) [3 pts]

```
let foo lst = let total, _ =
    fold_left
    (fun (tot, idx) ele ->
        (tot * ele * idx, idx + 1))
    (1, 0)
    lst in total;;

foo [1;3;5] ;;
```

Type of foo: int list -> int

Evaluation: 0

Problem 3: Filter

[Total 4 pts]

`filter` is another common higher order function that has type `('a -> bool) -> 'a list -> 'a list`. It applies a function to every item in a list and returns a list of the items that caused the function to return true. **using only `fold` (left or right, given below)**, write a function called `my_filter` which has the same functionality as `filter`. `f` will be the function that returns true or false, and `l` will be the list. **Note:** the original order must be maintained.

```
(* example: my_filter (fun x -> x > 3) [2;4;6] = [4;6] *)

let my_filter f l = fold_right (fun x acc -> if f x then x :: acc else acc) l []
---------- alternatively ----------
let my_filter f l = fold_left (fun acc x -> if f x then acc @ [x] else acc) [] l
```

Problem 4: Coding

[Total 6 pts]

Write a function call `last_sum` which takes in a `int list list` and returns the sum of the last elements in each `int list`. If list is empty, it adds nothing to the total.

You can write helper functions, you may use the `rec` keyword, **you do not have to use map/fold** (however they are still given). You may not use any List module functions, except those provided. (cons and @ are fine). You may also not use any imperative OCaml.

```
(* last_sum has type int list list -> int *)
(* Examples
    last_sum [] = 0
    (* 3 + 6 + 9 *)
    last_sum [[1;2;3];[4;5;6];[7;8;9]] = 18
    (* 0 + 1 + 3 *)
    last_sum [[];[1];[2;3]] = 4
*)

(* Write your code below *)
```

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

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

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

```
 let rec last_sum mtx =
    let lasts = map (fun x -> fold_left (fun acc el -> el) 0 x) mtx in fold_left (+) 0 lasts
---------- alternatively ----------
let last_usm mtx = fold_left (fun a x -> a + (fold_left (fun a x -> x) 0 x)) 0 mtx
---------- alternatively ----------
let rec get_last lst = match lst with
| [] -> 0
| [x] -> x
| x :: xs -> get_last xs

let rec last_sum mtx = let lasts = map (fun x -> get_last x) mtx in fold_left (+) 0 lasts
---------- alternatively ----------
let rec last_sum mtx =
    let rec get_last lst = match lst with
          [] - > 0
        | [x] -> x
        | _::xs -> get_last xs in
    match mtx with
    [] -> 0
    |x::xs -> get_last x + last_usm xs
```

Write a function call `last_prod` which takes in a `int list list` and returns the product of the last elements in each `int list`. If list is empty, it multiplies the value by 1.

You can write helper functions, you may use the `rec` keyword, **you do not have to use map/fold** (however they are still given). You may not use any List module functions, except those provided. (cons and @ are fine). You may also not use any imperative OCaml.

```
(* last_prod has type int list list -> int *)
(* Examples
    last_prod [] = 1
    (* 3 * 6 * 9 *)
    last_prod [[1;2;3];[4;5;6];[7;8;9]] = 162
    (* 1 * 1 * 3 *)
    last_prod [[];[1];[2;3]] = 3
*)

(* Write your code below *)
```

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

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

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

```
  let rec last_prod mtx =
    let lasts = map (fun x -> fold_left (fun acc el -> el) 1 x) mtx in fold_left ( * ) 1 lasts

---------- alternatively ----------

let last_prod mtx = fold_left (fun a x -> a * (fold_left (fun a x -> x) 1 x)) 1 mtx

---------- alternatively ----------

let rec get_last lst = match lst with
| [] -> 1
| [x] -> x
| x :: xs -> get_last xs

let rec last_prod mtx = let lasts = map (fun x -> get_last x) mtx in fold_left ( * ) 1 lasts

---------- alternatively ----------

let rec last_prod mtx =
    let rec get_last lst = match lst with
          [] - > 1
        | [x] -> x
        | _::xs -> get_last xs in
    match mtx with
    [] -> 1
    |x::xs -> get_last x * last_prod xs
```

Write a function call `last_concat` which takes in a `string list list` and returns the result of concating the last elements in each `string list`. If list is empty, it should not modify the resulting string.

You can write helper functions, you may use the `rec` keyword, **you do not have to use map/fold** (however they are still given). You may not use any List module functions, except those provided. (cons and @ are fine). You may also not use any imperative OCaml.

```
(* last_concat has type string list list -> string *)
(* Examples
    last_concat [] = ""
    (*"b" ^ "d" ^ "f" *)
    last_concat [["a";"b"];["c";"d"];["e";"f"]] = "bdf"
    (* "" ^ "a" ^ "c" *)
    last_concat [[];["a"];["b";"c"]] = "ac"
*)

(* Write your code below *)
```

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

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

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

```
  let rec last_concat mtx =
    let lasts = map (fun x -> fold_left (fun acc el -> el) 1 x) mtx in fold_right ( ^ ) lasts ""

----------- alternatively -----------

let last_concat mtx = fold_left (fun a x -> (fold_left (fun a x -> x) "" x) ^ a) "" mtx

----------- alternatively -----------

let rec get_last lst = match lst with
| [] -> ""
| [x] -> x
| x :: xs -> get_last xs

let last_concat mtx = let lasts = map (fun x -> get_last x) mtx in fold_right (^) lasts ""

----------- alternatively -----------

let rec last_concat mtx =
    let rec get_last lst = match lst with
          [] - > 1
        | [x] -> x
        | _::xs -> get_last xs in
    match mtx with
    [] -> 1
    |x::xs -> get_last x ^ last_concat xs
```