1 <3 ocaml

CMSC330 Fall 2025 Quiz 1

Proctoring TA: _____ **Name:** _____

Section Number: _____ **UID:** _____

Problem 1: Basics

[Total 4 pts]

	true	false
List.length x = (List.length (List.map f x)) for all valid f and x (i.e. assume List.map f x compiles)	T	F
if fold_left f a l compiles and results in value v then fold_right (fun x a -> f a x) l a should also result in v	T	F
In OCaml the entire function body is a single expression	T	F

OCaml lists are immutable. List 2 benefits of immutability in functional programming.

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 rec foo lst =
  match lst with
  h1::h2::t-> h2 :: h1 :: foo t
  |_-> lst;;

foo [1;2;3;4;5] ;;
```

Type of `foo`:

Evaluation:

(b) [3 pts]

```
let foo f x  = f (f x);;

foo (fun x-> [List.length x]) [3;6;9] ;;
```

Type of `foo`:

Evaluation:

Problem 3: Coding

[Total 6 pts]

For the following coding question, you may write helper functions, you may use recursion but aren't required to, **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. **Read the examples carefully**.

Given Functions:

```
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)
```

Define a function fold_if that behaves like fold_left, but takes an additional predicate argument pred : 'acc -> 'a -> bool During the fold, **before applying the function to the current element**, pred is checked with the current accumulator and the current element. If pred evaluates to false, the fold stops and returns the current accumulator.

```
(* Examples

(* stop when we encounter an element > 10 *)
let pred _acc x = x <= 10 in
let f acc x = acc + x in
fold_if pred f 0 [1;4;6;11;2]
(* returns 11 (1+4+6), stops at 11 *)

(* predicate that depends on accumulator: stop once acc >= 10 *)
let pred_sum acc _x = acc < 10 in
let f acc x = acc + x in
fold_if pred_sum f 0 [3;4;5;1]
(* returns 12 (3+4+5). it adds up 3 and 4, making acc to be 7.It gets to the element 5,
checks if acc is greater than 10 (it is not, 7 < 10), then adds 5 (acc is now 12). It then gets to
the element 1 and checks if acc > 10 (it is), which makes pred_sum evaluate to false,
so it immediately returns the current accumulator of 12 *)

(* Write your code below *)
let rec fold_if pred f init lst =
```