1. [6 points] Give the types of the following OCaml expressions.

   a. [2 points] \((1.0, [1 < 0])\)
      
      \text{float * bool list}

   b. [2 points] \(\text{fun } x \ y \rightarrow x \ y\)
      
      \(\text{('a -> 'b) -> 'a -> 'b}\)

   c. [2 points] \(\text{fun } f (x, y) \rightarrow f \ x \ y\)
      
      \(\text{('a -> 'b -> 'c) -> 'a * 'b -> 'c}\)

2. [4 points] Give OCaml expressions with the following types.

   a. [2 points] \((\text{string * int}) \text{ list}\)
      
      \text{One possible answer: } [\text{("hi", 10)}]

   b. [2 points] \(\text{`a list} \rightarrow \text{string * `a}\)
      
      \text{One possible answer: let silly lst = match lst with | h::t -> ("hello", h);;}

      \text{*Note that this pattern matching is not exhaustive but that doesn’t matter since we only care about what type it is, not if it is useful*}
let rec map f = function
  | []       -> []
  | x :: xs   -> f x :: map f xs

let rec fold f v = function
  | []       -> v
  | x :: xs   -> fold f (f v x) xs

type 'a option =
  | None
  | Some of 'a

3. [4 points] Evaluate the following OCaml expression. There are no syntax or type errors.

let rec foo xs ys =
  match (xs, ys) with
  | (x :: xs, y :: ys) -> (x, y) :: foo xs ys
  | _                  -> [] in
fold (fun v (x, y) -> v +. x *. y) 0.0
  (foo [1.0; 0.5; 1.5] [1.0; 4.0; 2.0])

Answer: 6.0 (6 is not entirely correct since it is an int)
Quick Explanation:
- foo evaluates to [(1.0, 1.0); (0.5, 4.0); (1.5, 2.0)]
- The fold then adds up the multiplication of the tuple values so:
  - 0.0 + 1.0*1.0 + .5*4.0 + 1.5*2.0
  - 1.0 + 2.0 + 3.0
  - 6.0

4. [6 points] Write a function map_options: ('a -> 'b) -> 'a option list -> 'b list that accepts as arguments a function f: 'a -> 'b and list of optional values xs: 'a option list and returns the result of applying f to each element of xs ignoring None values. You may use Pervasives.(@) or List.append. You must use fold or map.

map_options (fun x -> x + 1) [None; None] = []
map_options (fun x -> x + 1) [Some 1; None; Some 2] = [2; 3]
Some possible answers:

```ocaml
define map_options f xs = fold (fun acc el -> match el with
    | None -> acc
    | Some c -> acc @ [ f c ]) [] xs

define map_options f xs =
    let rec filter_out_from_somes s = match s with
        | [] -> []
        | None::t -> filter_out_from_somes t
        | Some c::t -> c::(filter_out_from_somes t)
    in map f (filter_out_from_somes xs)

The idea is to either go through with a fold and apply and add if a Some and ignore if None OR filter out the None values and then do map over what is left.
```