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

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list) : 'b =
begin match l with
| [] -> base
| hd :: tl -> combine hd (fold combine base tl)
end

let test () : bool =
fold (fun _ acc -> acc) 42 [] = 42
;; run_test "fold returns base case on empty list" test

type rgb = { mutable r: int; mutable g: int; mutable b: int }
1. Write the type of each of the following OCaml expressions in the blank provided, or ill-typed if the expression does not type check. You may want to refer to the definitions reproduced on the opposite side of this page.

(a) let a : ____________________________ = None
(b) let b : ____________________________ = print_endline
(c) let c : ____________________________ = print_endline "Hi"
(d) let d : ____________________________ = run_test
(e) let e : ____________________________ = { r = 0; g = 0; b = 128 }
(f) let f : ____________________________ = e.b <- 0

2. Rewrite each of the following functions using fold. You may use anonymous functions or define named helper functions, but you may not use explicit recursion (i.e., with the rec keyword). Feel free to write on the other side of the page if you need more space.

(a) (* Computes the product of a list of integers. *)
let rec product (l: int list) : int =
begin match l with
| [] -> 1
| hd :: tl -> hd * (product tl)
end

let product' (l: int list) : int =

(b) (* Composes together the list of functions and applies x to the composition. For example, compose 3 [g; f] should return g(f(3)). *)
let rec compose (x: int) (l: (int -> int) list) : int =
begin match l with
| [] -> x
| f :: fs -> f (compose x fs)
end

let compose' (x: int) (l: (int -> int) list) : int =

(c) (* Flattens a list of options into a list of all the values actually present. *)
let rec flatten (l: 'a option list) : 'a list =
begin match l with
| [] -> []
| hdo :: tl ->
    let rest = flatten tl in
    begin match hdo with
    | None -> rest
    | Some hd -> hd :: rest
    end
end

let flatten' (l: 'a option list) : 'a list =