Total Points = 100

Problem 1. OCaml Types (Points = 40. Each question is worth 4 points. No partial points.)

Give the type of the following OCaml expressions / functions:

-4

1. `["apple"; "orange"]` string list
2. `["apple", "orange"]` string * string X
3. `Some 1` int option
4. `()` unit
5. `fun x -> x *. 2.0` float → float
6. `let swap (a, b) = (b, a)` 'a * 'b → 'b * 'a
7. `fun x -> String.length x` string → int
8. `let choose b x y = if b then x else y` bool → 'a → 'a → 'a
9. `let rec power x n = if n = 0 then 1 else x * power x (n-1)` int → int → int
10. `let rec repeat n s = if n <= 0 then "" else s ^ repeat (n-1) s`

    int → string → string

Problem 2. OCaml Functions (Points = 60. Each question is worth 6 points. Partial points allowed.)

1. Define a polymorphic type `'a tree` to represent a binary tree. The tree should either be:

- empty, or
- a node containing a value of type `'a`, along with left and right subtrees of the same type.

Example usage:

```
let t = Node (1, Node (2, Empty, Empty), Node (3, Empty, Empty));;
```

This creates a binary tree where the root is 1, with a left child 2 and a right child 3.

2. Implement a function `count_vowels : string -> int` that counts the number of vowels (a, e, i, o, u, *case-insensitive*) in a string.

```
# count_vowels "Hello";;
- : int = 2
# count_vowels "Ocaml";;
- : int = 2
# count_vowels "WHY";;
- : int = 0
```

```
1) type 'a tree = Empty | Node of of 'a * 'a tree * 'a tree

2) let count_vowels s = let lst = string-to-char_list s in
     let rec aux l =
     match l with
     [] -> 0
     | h::t -> if h='a' || h='e' || h='i' || h='o' || h='u' ||
                  h='A' || h='E'' || h='I' || h='O' || h='U'
               then 1 + aux t
               else aux t in

     aux lst ;;
```

3. Write a function safe_div : int -> int -> int option that divides two integers and returns Some result if the denominator is not zero, or None if it is.

```
# safe_div 6 2;;
- : int option = Some 3
# safe_div 5 0;;
- : int option = None
```

```
let safe_div x y = match y with
   0 -> None
   | _ -> Some (x/y) ;;
```

4. Write a function

```
replicate : int -> 'a -> 'a list
```

that returns a list containing n copies of the given value. If n <= 0, return the empty list.

Examples:

```
# replicate 3 "ocaml";;
- : string list = ["ocaml"; "ocaml"; "ocaml"]

# replicate 0 5;;
- : int list = []

# replicate (-2) true;;
- : bool list = []
```

```
let rec replicate n item =
    if n <= 0 then []
    else item :: replicate (n-1) item;;
```

```
# option_map (fun x -> x * 2) [1; 2; 3];;
- : int list option = Some [2; 4; 6]

# option_map (fun x -> x + 1) [];;
- : int list option = None
```

```
let option_map f lst =
    match lst with
    [] → None
    | _ → let rec aux l = match l with
                            [] → []
                            | h::t→(f h)::(aux t) in
          Some (aux lst);;
```

6. Implement a function `lookup : string -> (string * int) list -> int option` that finds the integer value associated with a key in an association list, returning `Some value` if found, or `None` if not.

```
# lookup "b" [("a", 1); ("b", 2); ("c", 3)];;
- : int option = Some 2
# lookup "x" [("a", 1); ("b", 2)];;
- : int option = None
```

```
let rec lookup s lst = match lst with
   [] → None
   | (x,y)::t → if s=x then some y
                          else lookup s t;;
```

7. Write a function

```
remove_if : ('a -> bool) -> 'a list -> 'a list
```

that removes all elements from a list that satisfy a given predicate.

Examples:

```
# remove_if (fun x -> x mod 2 = 0) [1;2;3;4;5;6];;
- : int list = [1;3;5]

# remove_if (fun x -> x > 0) [0; -1; 2; -3; 4];;
- : int list = [0; -1; -3]

# remove_if (fun _ -> true) [1;2;3];;
- : int list = []
```

```
let rec remove_if f lst = match lst with
   [] → []
   | h::t → if f h then remove_if f t
                else h::remove_if f t;;
```

8. Write a function

```
flatten : 'a list list -> 'a list
```

that takes a list of lists and concatenates them into a single list. Do not use `List.flatten`; implement it yourself recursively.

Examples:

```
# flatten [[1;2]; [3]; [4;5;6]];;
- : int list = [1;2;3;4;5;6]

# flatten [];;
- : 'a list = []

# flatten [[]; [1]; []];;
- : int list = [1]
```

```
let rec flatten lst_of_lst = match lst_of_lst with
    [] → []
  | h::t → let rec aux l = match l with
                              [] → []
                            | a::b → a :: aux b  in

              aux h @ flatten t ;;
```

useless?!

9. Write a function

```
split_even_odd : int list -> int list * int list
```

that takes a list of integers and returns a pair of lists:

- the first list contains all even numbers
- the second list contains all odd numbers
- The order of elements should be preserved.

Examples:

```
# split_even_odd [1;2;3;4;5;6];;
- : int list * int list = ([2;4;6], [1;3;5])

# split_even_odd [];;
- : int list * int list = ([], [])
```

```
let Spliteven_odd lst =
    let rec even_list l = match l with
        [] -> []
        | h::t -> if h mod 2 =0 then h::even_list t
                  else even_list t in
    let rec odd_list l = match l with
        [] -> []
        | h::t -> if h mod 2 =0 then odd_list t
                  else h:: odd-list t in

    (even_list lst, odd_list lst);;
```

10. Write a function

```
all_true : bool list -> bool
```

using `List.fold_left`, which returns `true` if all elements in the list are `true`, otherwise returns `false`. An empty list should return `true`.

Examples:

```
# all_true [true; true; true];;
- : bool = true

# all_true [true; false; true];;
- : bool = false

# all_true [];;
- : bool = true
```

```
let all-true lst = List.fold_left (fun acc b → acc && b)
        true lst;;
```