Ozyegin University
CS 321 Programming Languages
Sample Problems on Functional Programming

1. Write a function `stringy : string list -> (string * int) list` that associates each string in its input with the length of the string. You may use `String.length` to find the length of a string.

```
# stringy ["a"; "bbb"; "cc"; "ddddd"];;
- : (string * int) list = [("a", 1); ("bbb", 3); ("cc", 2); ("ddddd", 5)]
```

Solution:

```
let rec stringy lst =
  match lst with
  | [] -> []
  | x::xs -> (x, String.length x)::stringy xs
```

2. Write a function `positivesOf : int list -> int list` that returns the positive numbers in its input.

```
# positivesOf [-4; 9; 2; -8; -3; 1; 0];;
- : int list = [9; 2; 1]
```

Solution:

```
let rec positivesOf lst =
  match lst with
  | [] -> []
  | x::xs -> if x > 0 then x::positivesOf xs
             else positivesOf xs
```

3. Write a function `gotcha : ('a -> bool) -> 'a list -> 'a` that takes a predicate function `p` and a list `lst`, and returns the first element `x` of `lst` for which `p(x)` is true. If there is no such element, the function should fail with the error message "No soup for you!".

```
# gotcha (fun n -> n > 5) [3; 4; 1; 2; 8; 4; 9; -8];;
- : int = 8
# gotcha (fun n -> n > 15) [3; 4; 1; 2; 8; 4; 9; -8];;
Exception: Failure "No soup for you!".
```

To make the program fail in the error case, use the (`failwith "No soup for you!"`) expression.

Solution:

```
let rec gotcha p lst =
  match lst with
  | [] -> failwith "No soup for you!"
  | x::xs -> if p x then x
             else gotcha p xs
```

4. Write a function `allUntil : ('a -> bool) -> 'a list -> 'a list` that takes a predicate function `p`, a list `lst`, and returns all the elements of `lst` up to the first element that does not satisfy `p`.

```
# allUntil (fun n -> n < 5) [3; 4; 1; 2; 8; 4; 9; -8];;
- : int list = [3; 4; 1; 2]
# allUntil (fun n -> n > 5) [3; 4; 1; 2; 8; 4; 9; -8];;
- : int list = []
# allUntil (fun n -> n < 15) [3; 4; 1; 2; 8; 4; 9; -8];;
- : int list = [3; 4; 1; 2; 8; 4; 9; -8]
# allUntil (fun s -> String.length(s) < 4) ["aa"; "bbb"; "c"; "dddd"; "eeeeeeee"; "fff"];;
- : string list = ["aa"; "bbb"; "c"]
```

Solution:

```
let rec allUntil p lst =
  match lst with
  | [] -> []
  | x::xs -> if p x then x::(allUntil p xs)
             else []
```

5. Write a function `interleave : 'a list -> 'a list -> 'a list * 'a list` that mixes its inputs by interleaving their elements. In this question, you may assume that the inputs will always have the same length; that is, I won't test your function with naughty inputs.

```
# interleave [1;2;3;4;5] [6;7;8;9;10];;
- : int list * int list = ([6; 2; 8; 4; 10], [1; 7; 3; 9; 5])
# interleave [2;3;4;5] [7;8;9;10];;
- : int list * int list = ([7; 3; 9; 5], [2; 8; 4; 10])
```

Solution:

```
let rec interleave lst1 lst2 =
  match lst1, lst2 with
  | ([], []) -> ([], [])
  | (x::xs, y::ys) ->
     let (left, right) = interleave xs ys
     in (y::right, x::left)
```

6. Write a function `enumerate : 'a list -> ('a * int) list` that enumerates the elements of its input with their index. The first element in a list is considered to be at index 0. You will want to write a helper function for this problem.

```
# enumerate ['a';'b';'c';'d';'e'];;
- : (char * int) list = [('a',0);('b',1);('c',2);('d',3);('e',4)]
```

Solution:

```
let enumerate lst =
  let rec aux lst index =
    match lst with
    | [] -> []
    | x::xs -> (x, index)::aux xs (index+1)
  in aux lst 0
```

In all problems below, you must NOT use explicit recursion; use the library functions `map`, `fold_left`, and `fold_right`.

7. Write a function `stringyWithMap` that is exactly the same as `stringy`, but this time use `map`.

Solution:

```
let stringyWithMap lst =
  List.map (fun s -> (s, String.length s)) lst
```

8. Write a function `stringyWithFoldRight` that is exactly the same as `stringy`, but this time use `fold_right`.

Solution:

```
let stringyWithFoldRight lst =
  List.fold_right (fun s acc -> (s, String.length s)::acc) lst []
```

9. Write a function `stringyWithFoldLeft` that is exactly the same as `stringy`, but this time use `fold_left`.

Solution:

```
let stringyWithFoldLeft lst =
  List.fold_left (fun acc s -> acc@[(s, String.length s)]) [] lst
```

10. Write a function `positivesOfWithFoldRight` that is exactly the same as `positivesOf`, but this time use `fold_right`.

Solution:

```
let positivesOfWithFoldRight lst =
  List.fold_right (fun x acc -> if x > 0 then x::acc else acc) lst []
```

11. Write a function `positivesOfWithFoldLeft` that is exactly the same as `positivesOf`, but this time use `fold_left`.

Solution:

```
let positivesOfWithFoldLeft lst =
  List.fold_left (fun acc x -> if x > 0 then acc@[x] else acc) [] lst
```

12. Write a function `enumerateWithFoldLeft` that is exactly the same as `enumerate`, but this time use `fold_left`.

Solution:

```
let enumerateWithFoldLeft lst =
  let f acc x =
    let (lst, index) = acc
    in (lst@[x,index], index+1)
  in fst(List.fold_left f ([], 0) lst)
```

In the problems below, you may use explicit recursion or the library functions such as `map`, `fold_left`, and `fold_right`. It is a good idea to try solving the problems using both approaches.

13. Implement the following functions: `rev`, `append`, `flatten`, `map2`, `exists`, `mem`, `partition`, `assoc`, `combine`.

    Their definitions are available in the `List` module:

    `http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`

In the problems below, your implementation is required to be tail-recursive.

14. Write a function `positivesOf : int list -> int list` that returns the positive numbers in its input.

```
# positivesOf [-4; 9; 2; -8; -3; 1; 0];;
- : int list = [9; 2; 1]
```

Solution:

```
let positivesOf lst =
  let rec aux lst acc =
    match lst with
    | [] -> acc
    | x::xs -> aux xs (if x > 0 then x::acc else acc)
  in List.rev(aux lst [])
```

15. Write a function `enumerate : 'a list -> ('a * int) list` that enumerates the elements of its input with their index. The first element in a list is considered to be at index 0.

```
# enumerate ['a';'b';'c';'d';'e'];;
- : (char * int) list = [('a',0);('b',1);('c',2);('d',3);('e',4)]
```

Extra exercise: Solve the same problem when the elements are enumerated from right to left. E.g:

```
# enumerate ['a';'b';'c';'d';'e'];;
- : (char * int) list = [('a',4);('b',3);('c',2);('d',1);('e',0)]
```

Solution:

```
let enumerate lst =
  let rec aux lst acc =
    match lst with
    | [] -> acc
    | x::xs -> let (eList, index) = acc
               in aux xs ((x, index)::eList, index+1)
  in List.rev(fst(aux lst ([], 0)))
```