Ozyegin University
CS 321 Programming Languages
Sample Problems on Functional Programming

1. Given the following OCaml code.

```
let x = 3;;
let f y = x * y;;
let x = 5;;
let z = f 2;;
let k = f x;;
x = 10;;
let w = f x;;
let x = "hi";;
```

   (a) What are the values of `z`, `k`, and `w`?

   Solution: 6, 15, 15

   (b) Does the last line cause a type error? If not, what is the final value of `x`?

   Solution: No error. The value is `"hi"`.

2. 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
```

3. 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
```

4. 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
```

5. 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 []
```

6. 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)
```

7. 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`.

8. 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
```

9. 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 []
```

10. 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
```

11. 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 []
```

12. 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
```

13. 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.

14. 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.

15. 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 [])
```

16. 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)))
```

17. Write an OCaml function named `pick` that takes an integer `n` and a list named `lst`. The function returns the first `n` elements of `lst`. If `lst` has less than `n` elements, all the elements are returned.

For this question, you have to use explicit recursion; you may not use any library function including '@'. Points will be deducted if your implementation unnecessarily traverses all the elements of `lst`.

```
# pick;;
- : int -> 'a list -> 'a list = <fun>
# pick 5 [8;3;7;1;0;9;2;6];;
- : int list = [8; 3; 7; 1; 0]
# pick 5 [8;3;7];;
- : int list = [8; 3; 7]
```

Solution:

```
let rec pick n lst =
  match lst with
  | [] -> []
  | x::xs -> if n <= 0 then []
             else x::pick (n-1) xs
```

18. Write an OCaml function named `assoc` that takes a value `a` and a list of pairs named `lst`. The function returns the **rightmost** value associated with key `a` in `lst`.

    That is, `assoc a [ ...; (a,b); ...]  = b` if `(a,b)` is the **rightmost** pair that contains `a` as its first item. If there is no value associated with `a` in the list `lst`, fail with the error message `"Not found"`.

    Implement `assoc` using explicit recursion. Your solution should do a **single pass** over the list. In particular, a solution that first reverses the list and then finds the leftmost association is not acceptable. You may want to use a helper function in this problem.

```
# assoc;;
- : 'a -> ('a * 'b) list -> 'b = <fun>
# assoc 5 [(8,'e'); (6,'s'); (5,'f'); (2,'t'); (5,'h'); (5,'p'); (9,'n')];;
- : char = 'p'
# assoc 4 [(8,'e'); (6,'s'); (5,'f'); (2,'t'); (5,'h'); (5,'p'); (9,'n')];;
Exception: Failure "Not found".
```

Solution:

```
let rec assoc a lst =
  let rec helper lst acc =
    match lst with
    | [] -> (match acc with
             | [] -> failwith "Not found"
             | b::bs -> b)
    | (x,b)::xs -> helper xs (if x = a then b::acc else acc)
  in helper lst []
```

19. Write an OCaml function named `flatten` that takes a list of lists, and returns a list where all the elements of the argument are concatenated in the same order.

    Implement `flatten` using `fold_right`.

```
# flatten;;
- : 'a list list -> 'a list = <fun>
# flatten [[4;5;8]; [2;1;9;8]; [3]; [8;5;7;6]];;
- : int list = [4; 5; 8; 2; 1; 9; 8; 3; 8; 5; 7; 6]
```

Solution:

```
let flatten lsts =
  List.fold_right (fun lst a -> lst@a) lsts []
```

20. Write an OCaml function named `sums` that takes a list and produces another where each element is the accumulative sum of the elements up to and including the corresponding element in the input list. **Implement the function using `fold_left` (and possibly other library functions), but without explicit recursion.**

```
# sums;;
- : int list -> int list = <fun>
# sums [6;3;9;1;7;2];;
- : int list = [6; 9; 18; 19; 26; 28]
```

Solution:

```
let sums lst =
  List.rev (snd(List.fold_left (fun a x -> (fst a + x, (fst a + x)::snd a)) (0, []) lst))
```

21. Run-length encoding (RLE) is a data compression technique in which maximal (non-empty) consecutive occurrences of a value are replaced by a pair consisting of the value and a counter showing how many times the value was repeated in that consecutive sequence. For example, RLE would encode the list `[1;1;1;2;2;2;2;3;1;1;1;1;1]` as `[(1,3);(2,4);(3,1);(1;5)]`.

Write a function `rle` that takes a list and encodes it using the RLE technique. You may not use any library functions.

Solution:

```
let rec rle lst =
    match lst with
    | [] -> []
    | x::xs -> (match rle xs with
                | [] -> [(x, 1)]
                | (y,n)::ys -> if x = y then (y, n+1)::ys
                               else (x, 1)::(y, n)::ys)
```

22. Define a data type to represent *playing cards*. Each playing card has a *suit*, which is one of ♣, ♠, ♢, ♡. A playing card is either ace, king, queen, jack, or an ordinary card. An ordinary card is associated with a number.

Solution:

```
type suit = Club | Spade | Diamond | Heart

type card = Ace of suit
          | King of suit
          | Queen of suit
          | Jack of suit
          | Ordinary of suit * int

(* Alternatively: *)
type rank = Ace | King | Queen | Jack | Ordinary of int

type card = Card of suit * rank
```

23. Define an OCaml data type to represent numbers. There are three kinds of numbers:

- a *Real number*, which is defined by three integer values as its *significand*, *base*, and the *exponent*. E.g.:

$$12.3456 = \underbrace{123456}_{\text{significand}} \times \underbrace{10}_{\text{base}}{}^{\overbrace{-4}^{\text{exponent}}}$$

- a *Rational number*, which is defined as a quotient of two integers (i.e. the numerator and the denominator). E.g: $\frac{42}{79}$
- a *Complex* number, which is defined by a *real part* and an *imaginary part*, both of which are floating point numbers. E.g.:

$$\underbrace{3.14}_{\text{real}} + \underbrace{67.891}_{\text{imaginary}}\, i$$

Solution:

```
type number = Real of int * int * int
            | Rational of int * int
            | Complex of float * float
```

The problems below are based on the following definition of a binary tree:

```
type 'a binTree = BTLeaf of 'a
                | BTNode of 'a * 'a binTree * 'a binTree
```

24. Write a function `areIsomorphic` that takes two binary trees and determines whether the trees are isomorphic. Two trees are said to be isomorphic if their shapes are the same, regardless of the values in the trees.

Solution:

```
let rec areIsomorphic t1 t2 =
  match t1, t2 with
  | BTLeaf v1, BTLeaf v2 -> true
  | BTLeaf v1, _ -> false
  | _, BTLeaf v2 -> false
  | BTNode(v1,t11,t12), BTNode(v2,t21,t22) ->
    areIsomorphic t11 t21 && areIsomorphic t12 t22
```

25. Write a funcion `collect` that takes a binary tree `t` and a predicate function `p`, and returns all the elements of `t` that satisfy `p` in pre-order.

Extra challenge: Can you solve this problem without using the list append operator (`@`)?

Solution:

```
let rec collect t p =
  match t with
  | BTLeaf v -> if p v then [v] else []
  | BTNode (v, t1, t2) ->
    (if p v then [v] else []) @ collect t1 p @ collect t2 p
```

26. Write a funcion `isBST` that takes a binary tree `t` and determines whether `t` is a *binary search tree*. For a tree to be a binary search tree, all the elements to the left of a root must be less and all the elements to the right of a root must be larger than the root. Hint: Flattening the tree first may help you.

Solution:

```
let rec isBST t =
  let rec flatten t =
    match t with
    | BTLeaf v -> [v]
    | BTNode(v,t1,t2) -> flatten t1 @ [v] @ flatten t2
  in let rec isSorted lst =
    match lst with
    | [] -> true
    | [x] -> true
    | x::y::rest -> x < y && isSorted (y::rest)
  in isSorted (flatten t)
```

27. Write a function `gotcha` that takes a predicate `p` and a binary tree `bt`. The function returns the `Some` of the *first* element that satisfies `p` according to the in-order traversal of `bt`. (Reminder: in-order means left-root-right.) If there is no such element, the function should return `None`. Are you confused by `Some` and `None`? Then you have to read the “Data Types” slides.

Solution:

```
let rec gotcha p bt =
  match bt with
  | BTLeaf v -> if p v then Some v else None
  | BTNode (v, bt1, bt2) ->
     (match gotcha p bt1 with
      | Some v' -> Some v'
      | None -> if p v then Some v else gotcha p bt2)
```

The problems below are based on the following definition:

```
type cutelist = Empty
              | Cons of int * cutelist
```

Bart is a funny guy who likes to use his own definitions of data types as much as possible. Instead of the built-in lists, he decides to use a data type named `cutelist` (given above) to represent integer lists. For instance, instead of the list [1;2;3;4], Bart uses

```
Cons(1, Cons(2, Cons(3, Cons(4,Empty))))
```

28. Write an OCaml function `toCList` that takes an `int list` and returns the corresponding `cutelist` representation. **Implement `toCList` using `List.fold_right`. No explicit recursion is allowed!**

```
# toCList [1;2;3;4];;
- : cutelist = Cons (1,Cons (2,Cons (3,Cons (4,Empty))))
# toCList [3;6;8;2;7];;
- : cutelist = Cons (3,Cons (6,Cons (8,Cons (2,Cons (7,Empty)))))
```

Solution:

```
let toCList lst =
  List.fold_right (fun x acc -> Cons(x,acc)) lst Empty
```

29. Write an OCaml function `reverse` that takes a cutelist and returns its reverse. A solution that converts the cutelist to a regular list, then reverses the list, and finally converts the reversed list to a cutelist via `toCList` is NOT acceptable.

```
# reverse (Cons (3,Cons (6,Cons (8,Cons (2,Cons (7,Empty))))));;
- : cutelist = Cons (7,Cons (2,Cons (8,Cons (6,Cons (3,Empty)))))
```

Solution:

```
let rec reverse clst =
  let rec helper clst acc =
    match clst with
    | Empty -> acc
    | Cons(x,rest) -> helper rest (Cons(x,acc))
  in helper clst Empty
```

30. Write a function named `append` that takes two `cutelists` and returns their concatenation. Do NOT consider converting the `cutelist` values to built-in lists to solve this problem.

```
# append (Cons(3, Cons(4, Cons(5, Empty)))) (Cons(9, Cons(8, Empty)));;
- : int cutelist = Cons(3,Cons(4,Cons(5,Cons(9,Cons(8,Empty)))))
```

Solution:

```
let rec append clst1 clst2 =
  match clst1 with
  | Empty -> clst2
  | Cons(i, tail) -> Cons(i, append tail clst2)
```