Homework: Programming with Higher-Order Functions

The goal of this assignment is to exercise your understanding of functional programming.

1 Part 1. Programming with Higher-Order Functions

In this section, you must adhere to the following restrictions:

- You may use the `foldr` and `unfold` functions from the first lecture as helper functions.
- You may not use any other built-in or library functions. (In particular, you must not use the `List` library.)
- You may not use explicit recursion (i.e., do not write `let rec`)
- You may not use loops, exceptions, or mutable state.

Now, implement the following functions:

- Write the `length alist` function, which produces the length of `alist`.
- Write the `filter p alist` function, which produces a list that contains the elements of `alist` that satisfy the predicate `p`, in order.
- Write the `build_list f n` function, which produces a list of length `n`, where the ith element is the result of `f i`. i.e., `build_list f 5` is `[f 0; f 1; f 2; f 3; f 4]`.
- Write the `is_empty lst` function, which produces `true` if `lst` is `[]` and `false` otherwise. **In addition to the restrictions above, you may neither use (in-)equality-testing operators nor `match .. with`.**
- Write the `zip [x1; x2; ..] [y1; x2; ..]` function, which produces the list of pairs `[(x1, y1); (x2, y2); ..]`. If the two lists have unequal lengths, the extra elements in the longer list are ignored.
- Write `map_using_fold f lst`, which should be equivalent to `map`. Use `fold` to write this function.
- Write `map_using_unfold f lst`, using `unfold`.
- Write `factorial n`.
- Write `insert n lst`, which inserts the integer `n` into `lst` . Assume that the list sorted in ascending order and the function should preserve the order.
- Write `insertion_sort lst`.

2 2. Functional Programming Patterns

Implement the functions below without using any imperative features (i.e., no loops, exceptions, mutable data structures, laziness, and so on.) Unlike the previous set of functions, you may use the `List` library and explicit recursion.

2.1 In-Order Traversal

The following code defines a type for binary trees and a function that produces a list of the elements *in-order*:

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree

let rec in_order (t: 'a tree): 'a list = match t with
  | Leaf -> []
  | Node (lhs, x, rhs) -> in_order lhs @ x :: in_order rhs


let t1 = Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf))
let t2 = Node (Leaf, 1, Node (Leaf, 2, Leaf), 3, Leaf)

assert (in_order t1 = in_order t2)
```

Given two trees, it is easy to check if they have the same elements in-order:

```
let t1 = Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf))
let t2 = Node (Leaf, 1, Node (Node (Leaf, 2, Leaf), 3, Leaf))

assert (in_order t1 = in_order t2)
```

However, suppose it is immediately evident to you that two trees do not have the same elements in-order:

```
let t1 = Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf))
let t2 = Node (Leaf, 1200, Node (Leaf, 2, Leaf), 3, Leaf)

assert (in_order t1 != in_order t2)
```

This approach does a lot of needless work, since it traverses both trees anyway.

Write a function `same_in_order tree1 tree2` that doesn't do this extra work. In particular, the function should traverse both trees in order at most once, but if the nth pair of elements are different, it should not examine $(n+1)$th pair of elements.

2.2 Building Lists

The following function builds a list of numbers from m to n inclusive:

```
let rec from_to (m : int) (n: int) : int list =
  if m = n then [n]
  else m :: from_to (m + 1) n
```

It should be obvious that the function takes $O(n - m)$ time and requires $O(n - m)$ space. Therefore, if $n - m$ is large enough, it will run out of memory on a finite memory machine. However, it turns out that this function runs of of *stack space* if $n - m \geq 1,000,000$, which is disappointing because a list of million items is quite small for a modern computer. The problem is that the recursive calls to `build_list` are allocated on the stack and not the heap, and stacks are much smaller than the heap.

Rewrite the function so that it does not run out of stack space. The key is to ensure that *all* function applications are in tail position. You will not receive any credit if you simply reverse the list.

3 Template and Hand In

A template file for the assignment is provided on the course web page. Solve the assignment in this file and submit only this file using Moodle.