COMP 302 Programming Languages and Paradigms

Assignment 3

Due Date: 7th March 2016

There are four questions on this assignment. They are all required. Please submit the three programming questions in a file called assignment3.fs using the template on the web site. Please submit Question 4 in a pdf file.

[**Question 1. 30 points**] This exercise shows you how to do low-level pointer manipulation in F# if you ever need to do that. We can define linked lists as follows:

```
type Cell = { data : int; next : RList}
and RList = Cell option ref
```

Notice that this is a *mutually recursive* definition. Each type mentions the other one. The keyword `and` is used for mutually recursive definitions.

Implement an F# function `insert` which inserts an element into a *sorted* linked list and *preserves the sorting*. You do not have to worry about checking if the input list is sorted. The type should be

```
val insert : comp:(int * int -> bool) -> item:int -> list:RList -> unit
```

Insert takes in three arguments: A comparison function of type `int * int -> bool`, an element of type `int` and a linked list `l` of type `RList`. Your function will **destructively** update the list `l`. This means that you will have mutable fields that get updated. Please note the types carefully. Here is the code I used to test the program.

```
let c1 = {data = 1; next = ref None}
let c2 = {data = 2; next = ref (Some c1)}
let c3 = {data = 3; next = ref (Some c2)}
let c5 = {data = 5; next = ref (Some c3)}

(* This converts an RList to an ordinary list. *)
let rec displayList (c : RList) =
  match !c with
    | None -> []
    | Some { data = d; next = l } -> d :: (displayList l)
```

```
(* Useful if you are creating some cells by hand and then converting
them to RLists as I did above.  *)
let cellToRList (c:Cell):RList = ref (Some c)

(* Example comparison function.  *)
let bigger(x:int, y:int) = (x > y)
```

You may find the displayList and cellToRList functions useful. Here are examples of the code in action:

```
> let l5 = cellToRList c5;;
(* Messy display deleted. *)
> displayList l5;;
val it : int list = [5; 3; 2; 1]
val bigger : x:int * y:int -> bool
> insert bigger 4 l5;;
val it : unit = ()
> displayList l5;;
val it : int list = [5; 4; 3; 2; 1]
> insert bigger 9 l5;;
val it : unit = ()
> displayList l5;;
val it : int list = [9; 5; 4; 3; 2; 1]
> insert bigger 0 l5;;
val it : unit = ()
> displayList l5;;
val it : int list = [9; 5; 4; 3; 2; 1; 0]
```

The program is short (5 lines or less) and *easy to mess up.* Please think carefully about whether you are creating aliases or not. You can easily write programs that look absolutely correct but which create infinite loops. It might happen that your insert program looks like it is working correctly but then displayList crashes. You might then waste hours trying to "fix" displayList and cursing me for writing incorrect code. Most likely, your insert happily terminated but created a cycle of pointers which then sends displayList into an infinite loop.

[**Question 2. 20 points**] In class, we have shown you a program which mimics transactions done on a bank account. For this we have first defined a data-type for transactions:

```
type transaction = Withdraw of int | Deposit of int | CheckBalance
```

Then, we defined a function make-account which generates a bank account when given an opening balance.

In this exercise, you are asked to modify this code and generate a password-protected bank account. Any transaction on the bank account should only be possible, if one provides the right password. For this, implement the function `make_protected_account`. This function takes in the opening balance as a first argument and the password as a second, and will return a function which when given the *correct* password and a transaction will perform the transaction. One crucial difference to be noted right away is that in the new code I want you to **print the balance on the screen** instead of returning it as a value.

```
val make_protected_account :
  opening_balance:int * password:string -> (string * transaction -> unit)
```

Now, two things may go wrong. The password could be incorrect and the amount to be withdrawn could be too big. In these cases I want you to print an appropriate message on the screen and not let the transaction go through.

Here are examples of the code in action; I have deleted some lines:

```
val make_protected_account :
  opening_balance:int * password:string -> (string * transaction -> unit)
> let harry = make_protected_account(1000,"expelliarmus");;
val harry : (string * transaction -> unit)
> let voldemort = make_protected_account(100,"avada kedavra");;
val voldemort : (string * transaction -> unit)
> harry("expelliarmus",Withdraw(150));;
The new balance is 850:
> voldemort("avada kedavra",Deposit(50));;
The new balance is 150
> harry("episkey",Withdraw(500));;
Incorrect password.
> harry("expelliarmus",CheckBalance);;
The balance is 850:
```

[**Question 3. 30 points**] In this question we work with trees where the number of children at each point can vary. Instead of having a fixed number of subtrees we will have at each node an item and a list of subtrees. The type definition is:

```
type ListTree<'a> = Node of 'a * (ListTree<'a> list)
```

Note that is is parametric in `'a`. I want you to implement a general purpose breadth-first traversal. This should be a function that takes another function f as argument and then takes a `ListTree`. The function f is to be executed at each node. The nodes must be visited in breadth-first order. I want this done *imperatively* using the built-in Queue collection. It is up to you to learn about Queues. Here are examples of the code in action.

```
val bfIter : f:('a -> unit) -> ltr:ListTree<'a> -> unit
  Node
    (1,
     [Node (2,[Node (5,[]); Node (6,[]); Node (7,[]); Node (8,[])]);
      Node (3,[Node (9,[]); Node (10,[])]);
      Node (4,[Node (11,[Node (12,[])])])])

> bfIter (fun n -> printfn "%i" n) n1;;
1
2
3
4
5
6
7
8
9
10
11
12
val it : unit = ()
```

[**Question 4. 20 points**]

What is the result of evaluating the following expression? Explain your answer drawing the relevant environment diagrams. Without the explanation I will give zero, even for a correct answer, which, by the way, is 7.

```
let result =
    let x = 2
    let y = 1
    let f =
        let x = y
        fun u -> (u + x)
    let y = 6
    f(y)
```

If you get confused about indenting, I have written the following equivalent version (courtesy Carl) `in` to avoid confusion over the offside rule. (**Please turn over**)

```
let result =
    let x = 2 in
        let y = 1 in
            let f =
                let x = y in
                    fun u -> (u + x)
            in
                let y = 6 in
                    f(y)
```