COMP 302 Programming Languages and Paradigms

Assignment 3

Due Date: 7th March 2017

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

[**Question 1. 30 points**] This exercise shows you how to do low-level pointer manipulation in F#. 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 `reverse` which implements **in-place** reverse of a list by changing the pointers. 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)
```

You may find the `displayList` and `cellToRList` functions useful for testing purposes but they play no role in the solution. Here are examples of the code in action:

```
> displayList (cellToRList c5);;
val it : int list = [5; 3; 2; 1]
>
val reverse : lst:RList -> RList

> let l5 = cellToRList c5;;
> reverse l5;;
> displayList l5;;
val it : int list = [1; 2; 3; 5]
> let l0: RList = ref None;;

val l0 : RList = {contents = null;}

> reverse l0;;
val it : RList = {contents = null;}
```

The program is short (5 lines or fewer) 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 `reverse` 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. Perhaps your `reverse` 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 |
ChangePassword of string | Close
```

We have added two new transactions to the example done in class.

In class, 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 `makeProtectedAccount` with the arguments and types shown below.

```
let makeProtectedAccount(openingBalance: int, password: string) =
```

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 makeProtectedAccount :
  openingBalance: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.

The user can also close down the account which means that any *further* requests, even with the correct password, will just get the response `Account closed`. When the account is closed one should see the message `Account successfully closed`. Even after the account is closed it is still there and can answer requests, and will insist on the correct password, but the answer will always be `Account closed`. The user can also change password; from the type definition above it should be clear how this works.

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

```
val makeProtectedAccount :
  openingBalance:int * password:string -> (string * transaction -> unit)

> let Leah = makeProtectedAccount(1000, "BiologyRocks");;
val Leah : (string * transaction -> unit)
> let Talia = makeProtectedAccount(500, "MathIsAwesome");;
val Talia : (string * transaction -> unit)
> Talia("MathIsAwesome", Withdraw 50);;
The new balance is 450: .
val it : unit = ()
> Leah("PhysicsRocks", Withdraw 100);;
Incorrect password.
val it : unit = ()
> Leah("BiologyRocks",CheckBalance);;
The balance is 1000: .
val it : unit = ()
> Leah("BiologyRocks", Deposit 125);;
The new balance is 1125 .
val it : unit = ()
> Talia("MathIsAwesome", ChangePassword "CSisBetter");;
Password changed.
val it : unit = ()
> Talia("MathIsAwesome", Withdraw 50);;
Incorrect password.
val it : unit = ()
> Talia("CSisBetter", CheckBalance);;
The balance is 450: .
val it : unit = ()
```

```
> Leah("BiologyRocks", Close);;
Account successfully closed.
val it : unit = ()
> Leah("BiologyRocks", CheckBalance);;
Account closed.
val it : unit = ()
>
```

[**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. I have discussed queues in class briefly but you can read the details from documentation. 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. 0 points**] One can define stacks by means of equations as follows. There are three basic operations: `pop,top,push`. If s is a stack and x is an item we can specify the behaviour of a stack with the following equations.

$$top(push(x, s)) = x, \;\; pop(push(x, s)) = s.$$

Here `top` returns the top element without changing the stack and `pop` removes the top element and returns the new stack. There are error conditions when you try to apply these operations to an empty stack; I have not written them down. Here is the question: can you write down equations for queues? The answer is "no" but you should try to explain why and then *prove that it is impossible.*