[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.

Solution

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

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)

let cellToRList (c:Cell):RList = ref (Some c)

let bigger(x:int, y:int) = (x > y)

let rec insert comp (item: int) (list: RList) =
    match !list with
    | None -> list := Some { data = item ; next = ref None}
    | Some { data = d } when comp (item, d) ->
        let newCell = Some { data = item; next = ref (!list) } in list := newCell
    | Some { next = tail } ->
        insert comp item tail

What is dangerous in this question is writing next = list in case 2 of the match above. What happens in this case is that a cycle is created. However, the code for insert will terminate and when you try to display it you will see the loop. Deduct 10 points for this error.
[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:

```haskell
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.

```haskell
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.

**Solution**

```haskell
let make_protected_account(opening_balance: int,password: string) =
  let balance = ref opening_balance
  let passwd = password
  fun (passkey:string, t : transaction) ->
      if not (passkey = passwd)
      then
        printfn "Incorrect password."
      else
        match t with
        | Withdraw(x) ->
          if (x < !balance)
            then
              balance := !balance - x;
              printfn "The new balance is %i: " !balance
            else
              printfn "Insufficient funds."
        | Deposit(x) ->
          balance := !balance + x;
          printfn "The new balance is %i" !balance
        | CheckBalance ->
          printfn "The balance is %i:" !balance
```
It is crucial that the password is captured. This can be done explicitly as I have shown, but it will happen even if one does not do this explicitly, because the frame created by the top level call is still active and will be trapped in the closure.

[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:

```java
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.

Solution

```c
open System.Collections.Generic;

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

let bfIter f ltr =
    let todo = Queue<ListTree<'a>> ()
    todo.Enqueue ltr
    while (todo.Count <> 0) do
        let (Node(x,tl)) = todo.Dequeue()
        List.iter (todo.Enqueue) tl
        f x
```

Short and sweet!

[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.

```c
let result =
    let x = 2
    let y = 1
    let f =
        let x = y
        fun u -> (u + x)
    let y = 6
    f(y)
```
Solution First the picture after the function $f$ has been properly bound.

Note that the binding $(x, 2)$ does not point to result which has not been completed yet. In fact this will not be completed until the very end when everything else has been removed.

Now the next drawing shows the inner binding of $(y, 6)$ with the evaluation of $f(y)$ about to begin.
This picture shows the binding of the parameter to the argument and the evaluation of the function body about to start.
Finally we get the binding of \texttt{result}. Everything else has been removed.

It is important that they show or explain the stages of this computation. I do not want to be too stringent about showing every single stage but they must indicate that this is a situation that changes with the stage of the computation. Take off 3 points if they had the pointer from \((x, 2)\) going to \texttt{result} instead of to the global environment. Take off 3 points for not showing or saying that everything is removed at the end. Take 5 points off for getting the closure pointer wrong. Do not take points off for not mentioning the global environment.