COMP 302 Programming Languages and Paradigms
Assignment 5

Prakash Panangaden
McGill University: School of Computer Science

Due Date: 14^{th} **February 2020 3pm**

The assignment is due at 3pm. Extensions are not possible for any reason. There are two questions to be solved and one for your personal satisfaction.

[Question 1:**70 points**]
In this question you will implement one-variable polynomials as lists. Use the following type definitions and exception declaration

```
type term = Term of float * int
type poly = Poly of (float * int) list
exception EmptyList
```

A term like $3.5x^8$ is represented as $(3.5, 8)$. The exponent in a polynomial is **always** non-negative[1]. The exponent of a term is called its *degree*. A polynomial is a list of terms arranged so that the first term in the list has the highest degree and thereafter the terms are listed in decreasing order of the degree. We do not write terms that have 0.0 as a coefficient *except* in the special case where we have the zero polynomial. This is written as a polynomial with only one term `Poly [(0.0,0)]`. **An empty list is not a valid polynomial**. We also do not allow multiple terms of the same degree. These are just the rules that you *normally* use when writing polynomials. **Maintaining these restrictions is part of your programming task.** You can *assume* that the polynomials that you start with are represented correctly, and your outputs must respect these restrictions. You must make sure that you check whether your input is of the form `Poly []` which is an illegal input. In this case, raise the exception EmptyList. Apart from this you do not have to check that the input is of the right form. **Please do not include this case in your tests.**

Please implement the following functions:

```
val multiplyPolyByTerm : term * poly -> poly
val addTermToPoly : term * poly -> poly
val addPolys : poly * poly -> poly
val multPolys : poly * poly -> poly
```

The function `multiplyPolyByTerm` multiplies a term and a polynomial. The function `addTermToPoly`

[1]Polynomials with terms that have negative exponents are called Laurent polynomials and are a completely different kind of mathematical object.

adds a term to a polynomial. These two are then used in the next two functions which add and multiply polynomials respectively.

Code to start you off, including a couple of helpful auxiliary functions will be preloaded in the system..

```
# let t1 = Term (2.0,2);;
val t1 : term = Term (2., 2)
# let p1:poly = Poly [(3.0,5);(2.0,2);(7.0,1);(1.5,0)];;
val p1 : poly = Poly [(3., 5); (2., 2); (7., 1); (1.5, 0)]
# let p2 : poly = Poly [(3.0, 5); (4.0, 2); (7.0, 1); (1.5, 0)];;
val p2 : poly = Poly [(3., 5); (4., 2); (7., 1); (1.5, 0)]
# let p3 = multiplyPolyByTerm(t1,p1);;
val p3 : poly = Poly [(6., 7); (4., 4); (14., 3); (3., 2)]
# let p4 = addPolys(p1,p3);;
val p4 : poly =
  Poly [(6., 7); (3., 5); (4., 4); (14., 3); (5., 2); (7., 1); (1.5, 0)]
# let p5 = multPolys(p1,p3);;
val p5 : poly =
  Poly
   [(18., 12); (24., 9); (84., 8); (18., 7); (8., 6); (56., 5); (110., 4);
    (42., 3); (4.5, 2)]
```

[Question 2:**30 points**]

This exercise shows you how to do low-level pointer manipulation in OCaml 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 OCaml 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 cell2rlist (c:cell):rlist = ref (Some c)

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

You may find the `displayList` and `cell2rlist` functions useful. They will be preloaded for you.

Here are examples of the code in action:

```
# let l5 = cell2rlist c5;;
val l5 : rlist = ....
(* Messy display deleted. *)
# displayList l5;;
- : int list = [5; 3; 2; 1]
# displayList l5;;
- : int list = [5; 3; 2; 1]
# insert bigger 4 l5;;
- : unit = ()
# displayList l5;;
- : int list = [5; 4; 3; 2; 1]
# insert bigger 9 l5;;
- : unit = ()
# displayList l5;;
- : int list = [9; 5; 4; 3; 2; 1]
# insert bigger 0 l5;;
- : unit = ()
# displayList l5;;
- : int list = [9; 5; 4; 3; 2; 1; 0]
```

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 `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. **We will not ask for test cases for this question.**

[Question 3:**0 points**]
Can you come up with an algorithm that finds the largest and the next largest elements in an unsorted list or array using $n + \lceil \log_2 n \rceil - 2$ comparisons? Prove that this is optimal.