COMP 302 Programming Languages and Paradigms Assignment 2

Prakash Panangaden
McGill University: School of Computer Science

Due Date: 14^{th} **Febuary 2017**

Please answer all questions: there are 5 in all plus two for spiritual growth. You must use the function names that we have given. We will put a file on the web site which you should use as a template.

Questions 6 and 7 are for your spiritual growth only. Please do not submit any answers.

[Question 1:**15 points**] This is a classic example of a higher-order function in action. Newton's method can be used to generate approximate solutions to the equation $f(x) = 0$ where f is a differentiable real-valued function of the real variable x. The idea can be summarized as follows:
Suppose that x_0 is some point which we suspect is near a solution. We can form the linear approximation l at x_0 and solve the linear equation for a new approximate solution. Let x_1 be the solution to the linear approximation $l(x) = f(x_0) + f'(x_0)(x - x_0) = 0$. In other words,

$$\begin{aligned} f(x_0) + f'(x_0)(x_1 - x_0) &= 0 \\ x_1 - x_0 &= -\frac{f(x_0)}{f'(x_0)} \\ x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

If our first guess x_0 was a good one, then the approximate solution x_1 should be an even better approximation to the solution of $f(x) = 0$. Once we have x_1, we can repeat the process to obtain x_2, etc. In fact, we can repeat this process indefinitely: if, after n steps, we have an approximate solution x_n, then the next step is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

This will produce approximate solutions to any degree of accuracy provided we have started with a good guess x_0. If we have reached a x_n such that $|f(x_n)| < t$, where t is some real number representing the tolerance, we will stop.

Implement a function called `newton` with the type shown below

```
val newton : f:(float -> float) * guess:float * tol:float * dx:float -> float
```

which when given a function f, a guess x_0, a tolerance tol and an interval dx, will compute a value x' such that $|f(x')| < tol$. You can test this on built-in real-valued functions like `sin` or other functions in the mathematics library. Please note that this method does not always work: one needs stronger conditions on f to guarantee convergence of the approximation scheme. Never test it on *tan*!

[Question 2: **35 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]. 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 which has only one term. **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. The only thing that I want you to check is whether your input is of the form `Poly []` which is an illegal input.

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
val evalTerm : v:float -> term -> float
val evalPoly : poly * v:float -> float
```

The function `multiplyPolyByTerm` multiplies a term and a polynomial. The function `addTermToPoly` adds a term to a polynomial. These two are then used in the next two

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

functions which add and multiply polynomials respectively. The function `evalTerm` evaluates a term at a given floating point input and analogously for `evalPoly`. Code to start you off, including a couple of helpful auxiliary functions are on the web site. For the last two functions you will find it useful to have a function that raises a `float` to an integer power.

```
val exp : b:float * e:int -> float
```

We have written this for you. Use it! Here are some examples of the code in action:

```
> let t1 = Term (2.0,2);;
val t1 : term = Term (2.0,2)
> p1;;
val it : poly = Poly [(3.0, 5); (2.0, 2); (7.0, 1); (1.5, 0)]
> let p2 = addTermToPoly(t1,p1);;
val p2 : poly = Poly [(3.0, 5); (4.0, 2); (7.0, 1); (1.5, 0)]
> let p3 = multiplyPolyByTerm(t1,p1);;
val p3 : poly = Poly [(6.0, 7); (4.0, 4); (14.0, 3); (3.0, 2)]
> let p4 = addPolys(p1,p3);;
val p4 : poly =
  Poly [(6.0, 7); (3.0, 5); (4.0, 4); (14.0, 3); (5.0, 2); (7.0, 1); (1.5, 0)]
> let p5 = multPolys(p1,p3);;
val p5 : poly =
  Poly
    [(18.0, 12); (24.0, 9); (84.0, 8); (18.0, 7); (8.0, 6); (56.0, 5);
     (110.0, 4); (42.0, 3); (4.5, 2)]
> evalPoly(p1,1.0);;
val it : float = 13.5
```

Finally, we have the polynomials as symbolic structures so we can use our naive rules for computing derivatives. Recall that the derivative of a term like cx^n with respect to x is $(n * c)x^{n-1}$ and the derivative is linear. Write an F# function `diffPoly` to compute the derivative of a polynominal with respect to x. Here is an example.

```
val diffPoly : Poly -> Poly
> p1;;
val it : Poly = Poly [(3.0, 5); (2.0, 2); (7.0, 1); (1.5, 0)]
> diffPoly p1;;
val it : Poly = Poly [(15.0, 4); (4.0, 1); (7.0, 0)]
```

[Question 3: **20 points**] In this exercise you will work with expression trees very similar to the ones discussed in class and you will implement a little interpreter for them. The main difference is that you will implement your own lookup and insert functions to keep track of bindings and you will return **option** values. This could be done easily with the Map collection type but I want you to get the idea of how one handles situations where the item

you are looking for is not present. Thus we will use lists and raw pattern matching and explicit insertions of `Some` and `None`.

Define a function `lookup`, to lookup values in the binding list. If the name occurs more than once it must find the latest value inserted. We never remove values from the binding list. The binding list must be kept sorted by the name of the variable. You can use the `<` operator on strings but you need to give a type annotation to tell the system that you are using it on strings. Your lookup function should use options to deal with values that are not present.

Define a function `insert` to insert a new binding in the right place in the binding list.

Define a function `eval` which evaluates expressions and returns options.

Here are the types and function names.

```
type Exptree =
  | Const of int
  | Var of string
  | Add of Exptree * Exptree
  | Mul of Exptree * Exptree

type Bindings = (string * int) list
val lookup : name:string * env:Bindings -> int option
val insert : name:string * value:int * b:Bindings -> (string * int) list
val eval : exp:Exptree * env:Bindings -> int option
```

Here are some examples of eval in action:

```
eval(Add(Const 2, Const 3), []);;
val it : int option = Some 5
> eval(Add(Var "x", Const 3), [("x",2)]);;
val it : int option = Some 5
> eval(Add(Var "x", Const 3), [("y",2)]);;
val it : int option = None
>
```

Review **option** types before you start this question.

[Question 4: **15 points**] This question involves the Map collection library. Please read documentation on this collection from the web. We are going to model a couple of weeks of the English Premier League Football Season[2].

[2]The scores are fictitious.

Use the following type definitions:

```
type Team    = string
type Goals   = Goals of int
type Points  = Points of int
type Fixture = Team * Team
type Result  = (Team * Goals) * (Team * Goals)
type Table   = Map<Team,Points>
```

The league is just a list of team names. We will use the following list:

```
val league : string list =
  ["Chelsea"; "Spurs"; "Liverpool"; "ManCity"; "ManUnited"; "Arsenal";
   "Everton"; "Leicester"]
```

When a pair of teams play they each have a number of goals scored (which could be zero). We report this as a *result*, here is an example

```
(("Chelsea", Goals 2),("Spurs", Goals 1))
```

Note the use of the constructor function `Goals`. The team with more goals gets 3 points and the team with fewer goals gets 0 points. If they have the same number of goals (a draw) then each team gets 1 point. Points and goals are both (non-negative) integers so in order to distinguish them we declare them as different types with different tags (namely the constructor functions `Goals` and `Points`) so that our type checker can tell us if we are making type errors.

Write a function called `pointsMade` with the following type

```
val pointsMade :
  (Team * Goals) * (Team * Goals) -> (Team * Points) * (Team * Points)
```

which takes as input a result and outputs the points made by the teams. Here is an example:

```
val r5 : (string * Goals) * (string * Goals) =
  (("Chelsea", Goals 2), ("Spurs", Goals 1))
> pointsMade r5;;
val it : (Team * Points) * (Team * Points) =
  (("Chelsea", Points 3), ("Spurs", Points 0))
```

We have written the following function for you:

```
let initEntry (name:Team) = (name, Points 0)
let initializeTable l = Map.ofList (List.map initEntry l)
```

This creates a table which will record how many points each team has. Initially, no games have been played and every team has 0 points. The football matches are played (usually)

on the weekends so we have a list of results for each weekend. For the season we have a list of lists of results. Here is an example showing two weeks of results:

```
val s : Result list list =
  [[(("Chelsea", Goals 5), ("Arsenal", Goals 0));
    (("Spurs", Goals 3), ("ManCity", Goals 2));
    (("ManUnited", Goals 1), ("Liverpool", Goals 0));
    (("Everton", Goals 3), ("Leicester", Goals 5))];
   [(("Chelsea", Goals 2), ("Spurs", Goals 1));
    (("Liverpool", Goals 3), ("ManCity", Goals 2));
    (("ManUnited", Goals 1), ("Arsenal", Goals 4));
    (("Everton", Goals 1), ("Leicester", Goals 5))]]
```

Write functions with the name and type shown

```
val updateTable : t:Table * r:Result -> Table
val weekendUpdate : t:Table * rl:Result list -> Table
val seasonUpdate : t:Table * sll:Result list list -> Table
```

The first function updates the table using one result. The second one will use this to update the table based on a list of results from one weekend. The third function will update the table using the results from the entire season. I will illustrate with a very short two-week season as shown above.

```
> let t0 = initializeTable league;;
val t0 : Map<Team,Points> =
  map
    [("Arsenal", Points 0); ("Chelsea", Points 0); ("Everton", Points 0);
     ("Leicester", Points 0); ("Liverpool", Points 0); ("ManCity", Points 0);
     ("ManUnited", Points 0); ("Spurs", Points 0)]
> let t2 = seasonUpdate(t0,s);;
val t2 : Table =
  map
    [("Arsenal", Points 3); ("Chelsea", Points 6); ("Everton", Points 0);
     ("Leicester", Points 6); ("Liverpool", Points 3); ("ManCity", Points 0);
     ("ManUnited", Points 3); ("Spurs", Points 3)]
```

I intend you to use each function to define the following one.

The above does not have the teams shown in order of their points. In order to show the standings we will use insertion sort (no point writing a fancy sort for such small examples). I have written most of this code for you except for the comparison function called `less` which will be a parameter to the sorter.

```
let less((s1,n1):Team * Points, (s2,n2):Team * Points) = ....
```

```
let rec myinsert item lst =
  match lst with
  | [] -> [item]
  | x::xs -> if less(item,x) then x::(myinsert item xs) else item::lst

let rec isort lst =
  match lst with
  | [] -> []
  | x::xs -> myinsert x (isort xs)

let showStandings (t:Table) = isort (Map.toList t)
```

Here is what the result looks like:

```
> showStandings t2;;
val it : (Team * Points) list =
  [("Chelsea", Points 6); ("Leicester", Points 6); ("Arsenal", Points 3);
   ("Liverpool", Points 3); ("ManUnited", Points 3); ("Spurs", Points 3);
   ("Everton", Points 0); ("ManCity", Points 0)]
```

[Question 5: **15 points**] In this question we will implement some (very) simple graph algorithms: we will be given a road map and we will look for paths between cities. We are given the map as a list of pairs. The first item in each pair is the name of a city and the second item is a set of cities; these are the cities *directly* connected to the first city.

We use the following type definitions and raw data

```
type Destination = City of string
type RoadMap = Roads of Map<Destination,Set<Destination>>

val roadData : (string * string list) list =
  [("Andulo", ["Bibala"; "Cacolo"; "Dondo"]);
   ("Bibala", ["Andulo"; "Dondo"; "Galo"]); ("Cacolo", ["Andulo"; "Dondo"]);
   ("Dondo", ["Andulo"; "Bibala"; "Cacolo"; "Ekunha"; "Funda"]);
   ("Ekunha", ["Dondo"; "Funda"]);
   ("Funda", ["Dondo"; "Ekunha"; "Galo"; "Kuito"]);
   ("Galo", ["Bibala"; "Funda"; "Huambo"; "Jamba"]); ("Huambo", ["Galo"]);
   ("Jamba", ["Galo"]); ("Kuito", ["Ekunha"; "Funda"])]
```

The first task is to convert this raw data into a proper road map of type `RoadMap`.

```
val makeRoadMap : data:(string * string list) list -> RoadMap
```

Write the code for the function `makeRoadMap`.

Finally, write a function that takes the number of steps as a parameter and finds which cities are *up to that number of steps* away.

```
val upToManySteps :
  RoadMap -> n:int -> startCity:Destination -> Set<Destination>
```

Here are some examples:

```
> upToManySteps theMap 3 (City "Kuito");;
val it : Set<Destination> =
  set
    [City "Andulo"; City "Bibala"; City "Cacolo"; City "Dondo"; City "Ekunha";
     City "Funda"; City "Galo"; City "Huambo"; City "Jamba"; ...]
>  upToManySteps theMap 2 (City "Kuito");;
val it : Set<Destination> =
  set [City "Dondo"; City "Ekunha"; City "Funda"; City "Galo"; City
"Kuito"]
> upToManySteps theMap 2 (City "Jamba");;
val it : Set<Destination> =
  set [City "Bibala"; City "Funda"; City "Huambo"; City "Jamba"]
> upToManySteps theMap 2 (City "Andulo");;
val it : Set<Destination> =
  set
    [City "Andulo"; City "Bibala"; City "Cacolo"; City "Dondo"; City "Ekunha";
     City "Funda"; City "Galo"]
```

[Question 6: **0 points**] Annoying question; it will teach you patience and attention to detail. Write a program to *display* a polynomial on the screen in the manner that we are used to. This means that if the coefficient is 1.0 we don't write it, if the exponent is 0 we omit it, the terms are ordered by decreasing degree and zero terms are not displayed except in the special case of the zero polynomial.

[Question 7: **0 points**] This one is for ambitious students interested in the theory of algorithms. It is hard; I was not able to do it, but in the past 26 years that I have taught at McGill 3 people have done it.

Come up with the best (i.e. fewest comparisons) algorithm that you can to find the largest and the *second largest* numbers in a set of n numbers. There is an algorithm that can do it with $n+\log_2 n-2$ comparisons. The real challenge: prove that this is the best possible.