COMP 302 Programming Languages and Paradigms Assignment 2

Prakash Panangaden
McGill University: School of Computer Science

Due Date: 15^{th} **Febuary 2016**

Please answer all questions: there are 5 in all plus one for spiritual growth. The solution to Q5 can be typed up and submitted through myCourses as with the others solutions. Please put the non-programming question in a separate file from the F# programs. It is never too early to learn LaTeX! but I am not requiring it. 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. We will also put a file showing some examples of the code in action. The deadline is 4:30 pm on the 15th of February.

Question 6 is for your spiritual growth only. Please do not submit an answer.

[Question 1:**20 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: **25 points**] In this question you will implement one-variable polynomials as lists. Use the following type definitions:

```
type term = float * int
type poly = term list
```

A term like $3.5x^8$ is represented as $(3.5, 8)$. 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. 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.

Please implement the following functions:

```
val mtp : t:term * p:poly -> poly
val atp : t:term * p:poly -> poly
val addpolys : p1:poly * p2:poly -> poly
val multpolys : p1:poly * p2:poly -> poly
val evalterm : v:float -> float * int -> float
val evalpoly : p:poly * v:float -> float
val diff : p:poly -> poly
```

The function `mtp` multiplies a term and a polynomial. The function `atp` adds a term to a polynomial. These two are then used in the next two functions which add and multiply polynomials respectively. The function `evalterm` evaluates a term at a given floating point input and analogously for `evalpoly`. The function `diff` computes the symbolic form of the

derivative of a polynomial. Code to start you off, including a couple of helpful auxiliary functions are on the web site.

[Question 3: **25 points**] In this question you will implement a simple map colouring problem using the *Set* collection in F#. A country is just a string. A *chart* is simply a map (in the normal sense of the word) of some countries. It is represented as set of pairs of countries. If (a, b) is in the set it means that the countries a and b share a border. This relation is symmetric but we will not put both (a, b) and (b, a) in the set. We want to colour the chart so that two countries that share a border are not given the same colour. We will not name the colours; we simply view a colour as a set of countries that share the same colour. A **colouring** is a set of colours; hence a set of sets of countries. The algorithm takes a chart, and an initially empty colouring and then tries to extend the colouring by adding countries from the chart. It works by naively checking if a country can be added to a given colour by making sure that it is not a neighbour of any of the countries already with that colour.

Here are the type definitions and names of functions:

```
type Country = string
type Chart = Set<Country * Country>
type Colour = Set<Country>
type Colouring = Set<Colour>
val areNeighbours :
  ct1:'a -> ct2:'a -> chart:Set<'a * 'a> -> bool when 'a : comparison
val canBeExtBy :
  col:Set<'a> -> ct:'a -> chart:Set<'a * 'a> -> bool when 'a : comparison
val extColouring :
  chart:Chart -> colours:Colouring -> country:Country -> Set<Set<Country>>
val countriesInChart : chart:Chart -> Set<Country>
val colourTheCountries : chart:Chart -> Colouring
```

Details of how to use the set collection are readily available on the web.

[Question 4: **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. I do not want you to use the built-in **Map** collection because it causes an exception to be raised when you do not find something in the list.

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 5: **10 points**] Prove by induction that the insertion sort program shown in class works correctly. You can assume that the insert program works correctly but you need to write *carefully* what it means for this program to be correct before you proceed with your main proof.

[Question 6: **0 points**] Can you come up with an example of a differentiable function for which Newton's method does not work?