Project Functional Programming: Part 1
Academic Year 2012-2013

November 5, 2012

1 Introduction

Your final mark for the Functional Programming course will consists of 50% coming from the oral exam and 50% coming from project work. The project work consists of two programming assignments. The first assignment will represent 50% of your project mark and the second assignment will represent 50% of the mark. The first assignment has to be submitted before 3 December 2012 (8AM). The second assignment has to be submitted before 7 January 2013 (8AM). Submission will be done by sending your raw code files to Laurent Christophe before the deadline: `lachrist@ulb.ac.be`.

Most of the first assignment will be reused during the second assignment. We will give you feedback (during the lab sessions!) about the first assignment and you are welcome to improve the first assignment when submitting the second assignment. If you do so, an average will be made of both versions of the first assignment.

The projects are strictly personal and no plagiarism will be tolerated! You will have to defend your projects individually on a day that will be part of your regular exam schedule.

If you encounter any problems or if you have a precise question, you can always contact us at `lachrist@ulb.ac.be`. During winter holidays, we will do our best to give you feedback. However, this cannot be guaranteed.

2 The "Graph" type class

During this first assignement we focus on directed graphs. There are several ways to represent directed graphs, each one with its own strengths and weaknesses. Since we may want to easily add and modify concrete representations, we want to conceive them as multiple implementations of a common type class (judiciously named `Graph`). The `Graph` type class should be as generic as possible and should hide all the details of a concrete representation. Below, we give a minimal kernel of this type class. Feel free to extend it to simplify/improve your code, as long you do not break the abstraction layer.

```
-- g is the global type of the graph
-- n is the type of the node labels (must be unique)
-- e is the type of the edge labels
class Graph g n e | g -> n e where
  nodes :: g -> [n]                  -- enumeration of the nodes
  edge  :: g -> (n,n) -> Maybe e     -- returns an edge between two nodes (if any)
  empty :: g                         -- returns an empty graph
  insertNode :: g -> n -> g          -- add a node to the given graph
  insertEdge :: g -> (n,n) -> e -> g -- add an edge to the given graph
```

2.1 Language specification

By default, *ghc* only compiles Haskell 98 code. However, it allows Haskell programmers to use flags in order to enable more advanced language features. Those flags are specified at compile time or directly inside your source code files (which is the preferred method). So, given a list of flags we can either specifiy them at the `ghc` call site:

```
ghc File.hs -XFlag1 -XFlag2 ... -XFlagN
```

or inside a Haskell comment at the first line of your file:

```
{-# LANGUAGE Flag1, Flag2, ..., FlagN #-}
```

In order to compile the file containing the `Graph` typeclass you will need to add the following pragma at the top of your file:

```
{-# LANGUAGE MultiParamTypeClasses,
             FunctionalDependencies,
             FlexibleInstances,
             FlexibleContexts #-}
```

These flags correspond to the following features:

1. *Multiple Parameter Typeclasses*, `MultiParamTypeClasses`. By default, type classes accept only one parameter ; the `Graph` type class uses three parameters: `g`, `n` and `e`.

2. *Functional Dependencies*, `FunctionalDependencies`. The code `Graph g n e | g -> n e` introduces a functional dependency between `g` and `n`, `e` (e.g. `g` uniquely determine `n` and `e`).

3. *Flexible Instances*, `FlexibleInstances`. By default, inside a type class implementation, parameters can only be substituted by types of the form `(T a1 ... an)` where `a1` ... `an` are all type variables. So, for instance, you need to include this flag in order to provide an implementation of `Graph` where edges are of type `[Int]`.

   ```
   instance Graph <graph> <node> [Int] where ...
   ```

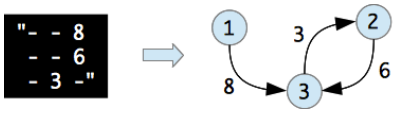
4. *Flexible Contexts*, `FlexibleContexts`. This flag enables the same freedom as `FlexibleInstances` but inside a context. For instance, this flag is required to write the type annotation shown below:

   ```
   f :: (Graph g n [Int]) => g -> n
   ```

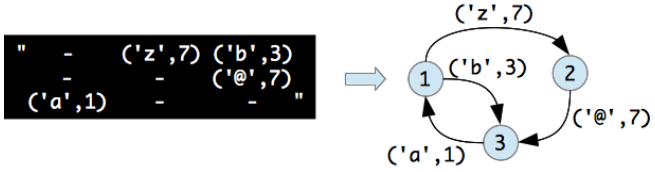
2.2 Programming Assignment:

1. Propose an implementation of the `Graph` type class based on the *adjacency matrix*[1] representation. Your representation should also implement the `Read` type class in such a way that it can be used to parse a string that represents a matrix. This representation assumes that node labels are simply an enumeration of integers. Here are two examples:

   (a) The string below can be read as a graph of three nodes whose edges and nodes are both of type `Int`:

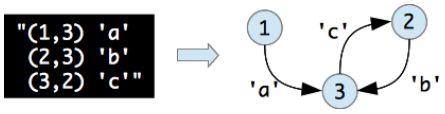
   (b) The string below can be read as agraph whose nodes are of type `Int` and whose edges are of type `(Int,Char)`:

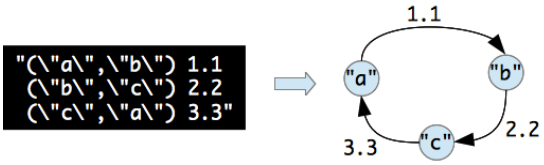
[1] http://en.wikipedia.org/wiki/Adjacency_matrix

2. Propose another implementation of the `Graph` type class that is based on a *binary search tree*[2]. Your representation should implement the `Read` type class again such that it can parse a string that represents a list whose elements have the form `(node1, node2) edge`. Here are two examples:

   (a) The string below can be read as a graph whose nodes are of type `Int` and whose edges are of type `Char`:

   (b) The string below can be read as a graph whose nodes are of type `String` and whose edges are of type `Float`:

3 Programming Assignment: Graph Algorithms

In this section, we ask you to implement two common algorithms on graphs:

1. Implement the *Dijkstra*[3] algorithm on a graph to obtain a list of nodes that represents the shortest path between two nodes (if any). The type signature of your function should be:

   ```
   dijkstra :: (Num e, Graph g n e) => g -> n -> n -> Maybe [n]
   ```

   (depending on the implementation the context part of the signature may have to be extended)

2. Implement the *Force-directed layout*[4] algorithm to position the nodes of a graph such that the graph is "nice" to see. Your functions should takes a graph, a random number generator[5] and return a mapping from the nodes to cartesian coordinates. The type signature of your functions should be (again, context is implementation-dependent):

   ```
   layout :: (Num e, Graph g n e, RandomGen r) => g -> r -> [(n,Float,Float)]
   ```

4 Show Time

4.1 PsTricks and LatexDraw

PsTricks[6] is a set LaTeX -macros that allow the inclusion of PostScript drawings directly inside TeX or LaTeX code. In this project you will have to generate *PsTricks* code to render and check the result of the implemented graph algorithms. The basic commands that you should use are:

- *Lines*: given two points (x_1, y_1) and (x_2, y_2), draws an arrow of color `MyColor` from the first point to the second point:

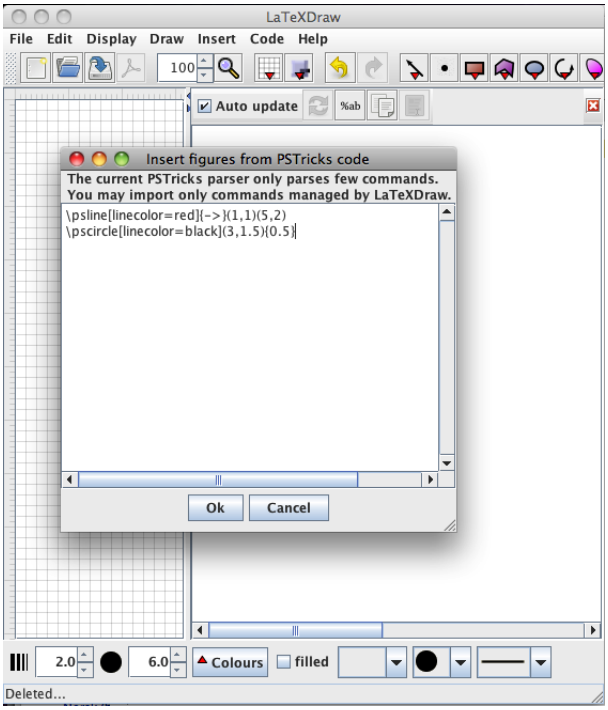
  ```
  \psline[linecolor=MyColor]{->}(x1,y1)(x2,y2)
  ```

- *Circles*: given a point (x, y), draws a circle of color `MyColor` with a radius r:

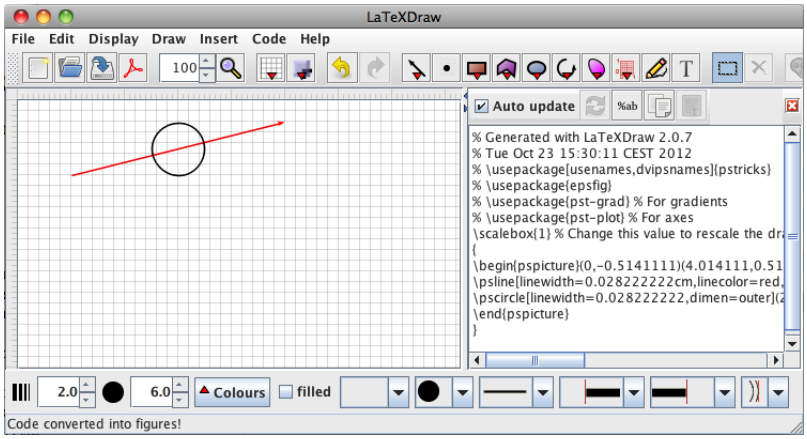
[2] http://en.wikipedia.org/wiki/Binary_search_tree
[3] http://en.wikipedia.org/wiki/Dijkstra's_algorithm
[4] http://en.wikipedia.org/wiki/Force-based_algorithms_(graph_drawing)
[5] This generator is needed to initially place the nodes. We suggest you to use the module `System.Random`, http://tinyurl.com/yldnaej
[6] http://archive.cs.uu.nl/mirror/CTAN/graphics/pstricks/base/doc/pstricks-doc.pdf

```
\pscircle[linecolor=MyColor](x,y){r}
```

Radii and cartesian coordinates can be any numeral, `MyColor` can be a user-defined color or predefined color like `black` and `red`. It is recommended to use *LatexDraw*[7] to test generated *PsTricks* codes. *LatexDraw* is a freeware program written in *Java* that is easy to install on all platforms. You can insert *PsTricks* code from the "insert" menu:



The resulting drawing from the previous dialog is:



4.2 Programming Assignment:

Write a function that, given a graph, a seed value and two nodes, produces valid *PsTricks* code.

```
toPsP :: (Num e, Graph g n e, RandomGen r) => g -> r -> n -> n -> Seed -> String
```

The interpretation of the *PsTricks* code should represent the graph `g` where:

1. The nodes have been placed according to the force-directed layout.
2. The shortest path between the given nodes computed by the Dijkstra method has been highlighted using a different color.

[7] `http://latexdraw.sourceforge.net/`