COMP304 Programming Languages (2017)
Assignment 2 : More Haskell
Due midnight, Monday 28 August

This assignment is intended to give you more experience in writing Haskell, and help you to understand some of the more advanced features of Haskell. You should try to make best use of the features that Haskell offers (especially algebraic data types, higher order functions, list comprehensions, modules and infinite lists) to construct elegant solutions to these problems, and use built-in functions from the standard prelude where appropriate (you may also use other library functions, as long as you explain them carefully, and this does not make the problem trivial).

You should think carefully about each function and how you might express it. Consider the different approaches that are available and discuss their relative merits. Think carefully before writing an explicit recursion — can you use a higher order function, such as a map or fold, instead? You should think about the efficiency of your solutions — whether they are as good as you expect them to be for the representation being used and whether they can be improved. You should also think about the effect of using lazy evaluation, and discuss its impact where it is significant.

You should submit Haskell source code for all of the programs, as literate Haskell scripts, using the online submission system. All Haskell code must be suitably commented to explain how it is intended to work, and should include signatures for all functions and code for test cases. Other discussion of what you have done and the results you obtained, should also be included. In particular, if your programs do not work correctly, you should provide test results demonstrating the incorrect behaviour, and explain what you think is the source of the problem and what might be needed to fix it.

Note that functions must have the names and arguments as shown in the questions, since part of the assignment will be marked automatically. Additional instructions regarding submission will be provided later — make sure you read them before submitting your assignment!

1. **Implementing sets**

   Suppose you want to implement a *set* type in Haskell. You recall from COMP103 that a set can be represented using (a) an unordered list, (b) an ordered list, or (c) a binary search tree.

   Implement a collection of set operations using each of these representations. You should include at least the following operations:

   (i) `makeSet :: [a] -> Set a` : Create a set with a given list of values, ignoring repeated elements.

   (ii) `has :: a -> Set a -> Bool` : Test whether a set contains a given element

   (iii) `card :: Set a -> Int` : Find the cardinality (number of elements) of a set.

   (iv) `add :: a -> Set a -> Set a` : Add an element to a set, leaving the set unchanged if it is already there.

   (v) `del :: a -> Set a -> Set a` : Delete an element from a set, leaving the set unchanged if it is not there.

(vi) `union :: Set a -> Set a -> Set a` : Form the union of two sets, i.e. the set of elements that occur in either (or both) of the sets.

(vii) `intersect :: Set a -> Set a -> Set a` : Form the intersection of two sets, i.e. the set of all elements that occur in both sets.

(viii) `equals :: a -> Set a -> Bool` : Determine whether two sets are equal, i.e. whether every element that occurs in either of the sets also occurs in the other. Can you use `(==)` instead of `equals` as the name for this function?

(ix) `subset :: a -> Set a -> Bool` : Determine whether one set is contained in another, i.e. whether every element that occurs in the first set also occurs in the second. Can you use `(<=)` instead of `subset` as the name for this function?

(x) `select :: (a -> Bool) -> Set a -> Set a` : Return the set of elements of a given set satisfying a given property.

Note that these signatures may need type constraints to be added.

Extra credit may be obtained by defining additional interesting operations.

`Set a` could be defined as a type synonym or as an algebraic data type — explain which you choose and why.

2. **Silly sort**

We can define a very simple, and possibly very inefficient, sort function in Haskell by noting that the result of sorting a list is a permutation of the list which is ascending order:

```
sort l = head [ s | s <- perms l, asc s ]
```

where `asc` determines whether a list is in ascending (or, more precisely, non-descending) order, and `perms` returns a list of permutations of a given list:

```
asc :: [a] -> Bool
perms :: [a] -> [[a]]
```

Define functions `asc` and `perms`, as described above, and perform some experiments to determine how the time and space used by `sort` is related to the size of the input.

You might find it helpful to start by assuming that the list has no repeated elements, and then extend your definition to handle lists with repeated elements.

Discuss how the use of lazy evaluation affects the time cost of this implementation, compared with the cost you would expect if eager evaluation was used.

3. **Graph algorithms**

An edge-labelled directed graph is defined to be a set of vertices, along with a set of edges, where an edge is a tripe, `(u,c,v)`, where `u` and v are the start and end vertices of the edge and `c` is the label on that edge. Such graphs are often called weighted graphs, when the labels are interpreted as weights or costs on edges, and can be represented in Haskell using the type:

```
type Graph a b = (Set a, Set (a,b,a))
```

where vertices are of type `a`, and weights are of type `b`. Note that it is implicit in this definition that there cannot be two edges between a given pair of vertices with the same label.

A path in edge-labelled directed graph is a list of edges, such that the end vertex of any edge is the same as the start vertex of the following edge:

```
type Path a b = [(a,b,a)]
```

Write the following functions on graphs:

(i) `makeGraph :: ([a], [(a,b,a)]) -> Graph a b` : Create a graph with given lists of vertices and edges — raise an error if any edge has a start or end vertex which is not in the given list of vertices, or if there are repeated vertices or edges.

(ii) `predecessors :: Graph a b -> a -> Set a` : Return the set of predecessors of a vertex (in a given graph – this will be left implicit in describing subsequent functions), i.e. the set of all vertices `u`, such that there is an edge from `u` to the given vertex.

(iii) `successors :: Graph a b -> a -> Set a` : Return the set of successors of a vertex, i.e. the set of all vertices `v`, such that there is an edge from the given vertex to `v`.

(iv) `isConnected :: Graph a b -> a -> Bool` : Check whether a graph is connected, relative to a given start vertex, i.e. if there is a directed path from a given vertex to every other vertex.

(v) `findPath :: Graph a b -> a -> a -> Maybe Path a b` : Find a path from one given vertex to another, if there is one. If there is more than one path from `u` to `v`, then `findPath g u v` may return any one of them.

(vi) `findPathLabel :: Graph a b -> a -> a -> Maybe [b]` : Find the label on a path from one given vertex to another, if there is one. The label on a path is the list of labels on the edges in the path. If there is more than one path from `u` to `v`, then `findPathLabel g u v` may return the label on any one of them.

(vii) `findMinCostPath :: Graph a b -> a -> a -> Maybe Path a b` : Find a path with minimal cost from one given vertex to another, if there is one. If there is more than one path from `u` to `v`, with minimal cost then `findMinCostPath g u v` may return any one of them.

(viii) `findPathWithLabel :: Graph a b -> a -> [b] -> Maybe Path a b` : Find a path starting from a given vertex with a given label. If there is more than one path starting from `u` with label `s`, then `findPathWithLabel g u s` may return any one of them.

In developing this function, you might find it helpful to start by assuming that for any vertex `u` and label `w`, there is at most one edge with start vertex `u` and label `w`. You can get part marks by providing a function that only works when this assumption holds.

Note that in many of these functions, you need to be careful to avoid going into an infinite loop when the graph contains cycles. To avoid this, you need to keep track of the vertices on the current path from the starting vertex and make sure you don't visit any of them again. You get path marks by providing functions that only work for non-cyclic graphs — but you should state that this is what you are doing.

For extra credit, after you have everything else working, you may attempt either of the following additional problem:

(a) `mst Graph a b -> Tree a b` : Construct a minimal cost spanning tree for a given graph, where `Tree a b` is a type consisting of trees with vertices of type `a` and labels of type `b` on its edges.

You may use either Prim's algorithm or Kruskal's algorithm, or any other you may know or invent, whichever you think will be easiest to implement in Haskell.