Exercise Sheet Five : Trees and Graphs

The aim of this tutorial is to explore the use of structured data for representing trees and graphs in Haskell and the pros and cons of this approach as compared to more tabular approaches.

Attempt the exercises in your own time before Monday's tutorial and use that session for obtaining feedback and for asking questions related to the exercises.

Exercise One

The following data type and function from the standard prelude are used to implement comparisons between Order types.

```
data Ordering = LT | EQ | GT
compare :: Ord a => a -> a -> Ordering
```

So `compare` returns `LT` if its first argument is less than its second argument etc. For data type

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

use `compare` to define a function `occurs :: Ord a => a -> Tree a -> Bool` that determines whether a given element occurs within the given tree. You may assume that the tree is a binary search tree.

Exercise Two

Write a `foldTree` function that implements a fold function for the `Tree` data type above. Use it to implement `flatten :: Tree a -> [a]` that creates a list of values stored in the given tree.

Exercise Three

Write a `fold` function for the data type of expressions

```
data Expr = Val Int | Add Expr Expr | Sub Expr Expr
```

Think carefully about what type it should have. Use your `fold` function to define a function `eval :: Expr -> Int` that evaluates an expression and `size :: Expr -> Int` that calculates the size of the AST (in nodes) that represents the expression.

Exercise Four

Consider the data type

```
data Prop = Const Bool | Var Char | Not Prop
          | And Prop Prop | Imply Prop Prop
```

A variable occurrence in a `Prop` value P is said to be in negative position if it appears under an odd number of negations. A negation is a `Not` constructor or the first argument of the `Imply` constructor. So for example, X is in negative position in `Not (Var X)` but not in `Imply (Not (Var X) ) (Const True)` as it appears under two negations in the latter.

- We say that a proposition P is in negative form if all occurrences of all of its variables are negative.
- We say that a proposition P is in positive form if all occurrences of all of its variables are positive.
- We say that a proposition P is in mixed form if neither of the above hold.

Write a function `getForm` that accepts a proposition and determines which of the above cases holds.

Exercise Five

Define the data type `data Pair a b = P (a, b)`. Write code to make the type constructor `(Pair a)` into a Functor.

Similarly, define the data type `data Fun a b = F (a -> b)`. Write code to make the type constructor `(Fun a)` into a Functor.

If we were to define `Fun` as `data Fun a b = F (b -> a)` instead, would it be possible to make `(Fun a)` into a Functor? Do this or explain why not.

Exercise Six

Modify the Tree Zipper from the lecture notes so as to work for the following variant of the Tree data type: `LTree a = Leaf a | Node (LTree a) (LTree a)`.

Now, using this Zipper, write a function `inc2LR :: LTree Int -> LTree Int` that increments by 1 the value of the 2nd leftmost and 2nd rightmost leaves in the given tree.

Exercise Seven

Use the Data.Graph module to build a graph as follows:

- Nodes are numbered as Ints from 0 to 1000
- There is an edge from every even numbered node (<1000) N to node (N+1)
- There is an edge from every odd numbered node N to the node (N div 5)

Write a function `isReachable :: Int -> Int -> Bool` that decides whether there is a path between two given nodes in your graph defined above.

Exercise Eight

Construct the same graph as above but this time use cyclic dependencies.

Implement the same `isReachable` function for this graph.