# COMP304 Programming Languages (2016)
# Assignment 2 : More Haskell (Due midnight, Friday 12 August)

This assignment is intended to help you to understand some of the more advanced features of Haskell, and to illustrate some of the things Haskell is good for. You should try to make best use of the features that Haskell offers (especially algebraic data types, higher order functions, list comprehensions and infinite lists) to construct elegant and/or efficient solutions to these problems. You should consider different approaches that are available and discuss their relative merits.

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, may be be embedded as comments or presented in a separate document.

Note that functions must have the names and arguments as shown in the questions, since part of the assignment will be marked automatically.

1. **Binary trees** (20 marks)

   Using the following data definition (as given in lectures):

   ```
   data BinTree a = Empty | Node a (BinTree a) (BinTree a)
   ```

   write the following functions on binary trees:

   (a) `hasbt x t`: Test whether the tree `t` contains a node with label `x`.

   (b) `equalbt t1 t2`: Test whether trees `t1` and `t2` are identical; i.e. are both empty, or have the same label at the root and the same subtrees.

   (c) `reflectbt t`: Construct a mirror image of tree `t`.

   (d) `fringebt t`: Construct the fringe of tree `t`; i.e. a list containing the labels on the leaves of the tree, in the order they would be visited in a left-to-right depth-first traversal.

   (e) `fullbt t`: Check whether tree `t` is full; i.e. if every node has either 0 or 2 subtrees.

2. **Binary tree folds** (30 marks)

   We have looked at fold operations on lists, which capture the common structure underlying lots of functions that accumulate results while traversing lists. We can define similar kinds of fold operations for other data structures, such as trees. For example, counting the number of nodes in a tree and finding the height of a tree can both be understood as instances of a general tree traversal algorithm which recursively computes a result for each subtree then combines these with the information at the root to obtain the result for the whole tree — the only difference is in how the results for the subtrees are combined with the information at the root.

   We will consider binary trees with labelled nodes, as declared above.

(a) Write a binary tree fold function, `btfold`, which takes as arguments a function, a "unit" value and a binary tree. If the tree is empty, the fold function should return the unit value; otherwise, it should apply the fold recursively to both subtrees, then apply the given function to the values returned and the value at the root. For example, to count the number of nodes in a tree, we would call `btfold` with the function `\u v w -> 1 + v + w` and unit value `0`; to sum the labels on a tree (assuming they are numerical labels), we would call `btfold` with the function `\u v w -> u + v + w` and unit value `0`.

    Think carefully about the type of `btfold`, and of the function and unit values it takes as arguments, remembering that in some cases, the fold may need to return a value with a different type from the tree labels.

(b) For each of the functions in part (a) above, **either** use your binary tree fold function to give an alternative definition to the one you gave in part (a) (these should be called `hasbtf`, etc), **or** explain why the function can't be defined using your fold function. In the latter case, think about whether it could be defined using a different fold function, and if so give a such a definition.

    **Hint:** Make sure that you understand how list folds work before attempting this question!

3. **Binary Search Trees**

    (a) Use the above binary tree data type to implement a binary search tree (BST), with the following operations:

    (i) `empty :: BinTree`
    Return an empty BST.

    (ii) `insert :: a -> BinTree a -> BinTree a`
    Insert an item into a BST (no change if already there).

    (iii) `has :: a -> BinTree a -> Bool`
    Check whether a given item occurs in a BST.

    (iv) `delete :: a -> BinTree a -> BinTree a`
    Delete a given item from a BST (no change if not there).

    (v) `flatten :: BinTree a -> [a]`
    Returns a list of the items in a BST.

    (vi) `equals :: BinTree a -> BinTree a -> Bool` Determines whether two BSTs contain the same items.

    Note that we regard the BST as representing a set, so all values are unique, and inserting a value which is already there and deleting a value which not there leave the BST unchanged.

4. **Graph algorithms**

We can represent a weighted directed graph as a list of edges, where each edge is represented by a triple `(u,c,v)`, where `u` and `v` are the vertices at the ends of the edge and `c` is the weight/cost on that edge:

```
type Graph a = [(a,Int,a)]
```

Will assume that weights are positive integers, and allow a self-loop (an edge with the same start and end vertex) with a weight of zero in order to allow a vertex with no inward or outward edges to be represented.

(a) Write a function `reachable x y g` to determine whether there is a path from vertex `x` to vertex `y` in graph `g`, and another function `minCostPath x y g` to find a minimal cost path from `x` to `y` in `g`.

These functions should be both be implemented using an additional function which finds all of the paths from one vertex to another. Explain how lazy evaluation ensures that execution of `reachable` is less expensive than that of `minCostPath`.

(b) Write a function `cliques g` which splits a graph `g` into a list of subgraphs, such that each graph in a maximal connected component of `g` (i.e. all of the vertices in the component can be reached from all others, and the compent contains all of the vertices that can be reached from any of its vertices; and every edge of `g` occurs in one of the components of the result).