Lab Exam Software Specification and Testing 2013, October 21

```
module LabExam

where
import Data.List
import Assert
```

There are six questions.

Question 1 Consider the following function:

```
f :: (Integer,Integer) -> (Integer,Integer)
f = until (odd.snd) (\ (m,n) -> (m+1,n `div` 2))
```

Write an assertive version of this by filling out the dots in the following definition. Next explain.

```
fA = assert1 (\ (s,r) (t,u) ->  ...  ==  ... ) f
```

Question 2 Binary trees can be defined in Haskell as:

```
data BinTree a = Nil | B a (BinTree a) (BinTree a) deriving (Eq,Show)
```

This is different from the version you saw this morning in the paper exam:

```
data Btree a = Leaf a | Node (Btree a) (Btree a) deriving (Eq,Show)
```

The difference is that `Btree` trees have information at their leaf nodes, but do not have information at their internal nodes, while with `BinTree` trees is is the other way around.

Implement a function `bintree2btree :: a -> BinTree a -> Btree a` that converts a `BinTree` to a `Btree` by throwing away internal node information and insertion of copies of the first argument at the leaves. So `bintree2bree x` should insert copies of `x` at the leaf nodes.

Implement a function `btree2bintree :: a -> Btree a -> BinTree a` that converts a `Btree` to a `BinTree` by throwing away the leaf information, and filling up the internal nodes with copies of the first argument. So `btree2bintree x` should insert copies of `x` at the internal nodes.

Next, show how you can test these two functions for correctness.

Question 3 In-order traversal of a binary tree visits the nodes of the tree by first doing an in-order traversal of the left subtree, then visiting the root node, and next doing an in-order traversal of the right subtree. Implement a function

```
inOrder :: BinTree a -> [a]
```

that collects the items found by in-order traversal of a binary tree in a list. Next define a second function for in-order traversal in right to left direction:

```
inOrderRev :: BinTree a -> [a]
```

Finally, fill in the dots to define a property that can be used to test the two functions by relating them to each other.

```
treeProperty :: Eq a => BinTree a -> Bool
treeProperty t = inOrder t == ...
```

Question 4 A *dictionary* is a binary tree with pairs of type `(String,String)` at its nodes.

```
type Dict = BinTree (String,String)
```

Call the first element of the pair the *key*, the second the *value*.

```
key, value :: (String,String) -> String
key (x,_) = x
value (_,y) = y
```

The idea is that the second string gives a translation or an explanation of the first string.

A dictionary is *ordered* if all items on the left subtree have keys that are alphabetically before the key at the root node, and all items on the right subtree have keys that are alphabetically after the key at the root node, and moreover the left and right subtrees are also ordered.

Write a test property `ordered :: Dict -> Bool` for this. (Hint: consider using the `inOrder` function from the previous exercise.)

Question 5 Implement a function `lookUp :: String -> Dict -> [String]` that looks up a key in an ordered dictionary. Make sure the lookup function exploits the order. An output `[]` indicates that the key is not defined in the dictionary, a non-empty list gives the value for a given key. Recall that the items in the dictionary tree have the form `(key,value)`. You can assume each key occurs at most once in the dictionary.

Question 6 Write code for inserting a new item at the correct position in an ordered dictionary. If the key of the item already occurs in the dictionary, replace the old information with the new information, otherwise just insert the new item. The type should be

```
insertKey :: (String,String) -> Dict -> Dict
```

Next, write an assertive version of this that checks whether an ordered dictionary is still ordered after the insertion. Use the property `ordered :: Dict -> Bool` that you defined earlier.