Lab Exam Software Specification and Testing 2014, October 23

```
module LabExamAnswers

where
import Data.List
```

There are six questions.

Question 1 Binary trees with information at the internal nodes can be defined in Haskell as follows (you have seen this definition in the paper exam this morning):

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

Implement a function `mapT` that does for binary trees what `map` does for lists. The type is

`mapT :: (a -> b) -> Btree a -> Btree b`

Include tests and/or specifications that demonstrate the correctness of your implementation.

Answer

```
mapT :: (a -> b) -> Btree a -> Btree b
mapT f Leaf = Leaf
mapT f (B x left right) = B (f x) (mapT f left) (mapT f right)
```

That this is correct can be seen by induction. The base case is correct, for mapping f to a leaf tree yields a leaf tree. If a tree consists of an item x, a left subtree t_1 and a right subtree t_2, and we can assume that the mapping works correctly for these subtrees, then the only thing we have to do in addition is replace x by fx.

Question 2 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 :: Btree 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 :: Btree 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 => Btree a -> Bool
treeProperty t = inOrder t == ...
```

Use this property to test your implementations.

Answer

```
inOrder :: Btree a -> [a]
inOrder Leaf = []
inOrder (B x left right) = inOrder left ++ [x] ++ inOrder right

inOrderRev :: Btree a -> [a]
inOrderRev Leaf = []
inOrderRev (B x left right) = inOrderRev right ++ [x] ++ inOrderRev left

treeProperty :: Eq a => Btree a -> Bool
treeProperty t = inOrder t == reverse (inOrderRev t)
```

How to test with this? Any tree should have this property!

Question 3 Recall the function `makeT` from the paper exam this morning, for transforming a list into a tree. Here it is again:

```
makeT :: Ord a => [a] -> Btree a
makeT = foldr insertT Leaf

insertT :: Ord a => a -> Btree a -> Btree a
insertT x Leaf = B x Leaf Leaf
insertT x (B y left right)
  | x < y     = B y (insertT x left) right
  | otherwise = B y left (insertT x right)
```

Define a sorting procedure for lists by means of in-order traversal of a tree created with `makeT`. Next, test your implementation well or prove its correctness.

Answer

```
srt :: Ord a => [a] -> [a]
srt = inOrder . makeT
```

We can see by induction that `makeT` creates an ordered tree. Inorder traversal of this tree creates an ordered list, for the traversal visits the internal nodes of the tree in their proper order (as the name 'inorder' implies).

Question 4 You are the leader of a team that develops tools for handling large dictionaries. A *dictionary* is a binary tree with pairs of type `(String,String)` at its nodes.

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

Call the first element of the pair the *lemma*, the second the *info*. Lemma/info pairs are specific versions of key/value pairs.

```
lemma, info :: (String,String) -> String
lemma (x,_) = x
info  (_,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 lemmas that are alphabetically before the lemma at the root node, and all items on the right subtree have lemmas that are

alphabetically after the lemma at the root node, and moreover the left and right subtrees are also ordered.

The first thing you need for success is a fast algorithm for dictionary lookup. Implement a function `lookUp :: String -> Dict -> [String]` that accomplishes this. An output `[]` indicates that the lemma is not defined in the dictionary, a non-empty list gives the info for a given lemma. Recall that the items in the dictionary tree have the form `(lemma,info)`. You can assume that your dictionary is ordered, and that each lemma occurs at most once in the dictionary.

Use inductive reasoning to show that your solution is correct.

Answer

This is a simple variation on binary search:

```
lookUp ::  String -> Dict -> [String]
lookUp x Leaf = []
lookUp x (B y left right) = if x == lemma y then [info y]
    else if x < lemma y then lookUp x left else lookUp x right
```

To show that this is correct, assume D is ordered. If D is empty (equal to Leaf), lookup should return [], and it does. If D is non-empty, and we assume that lookup in its left and right subtrees works correctly, then we get from the fact that D is ordered that the clause for lookup in `(B y left right)` is correct.

Question 5 Write code for inserting a new item at the correct position in an ordered dictionary. Generate an error if the lemma of the item already occurs in the dictionary, and an updated dictionary otherwise. The type should be

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

Next, write an automated test procedure for checking whether an ordered dictionary is still ordered after the insertion. Assume that you have a property `ordered :: Dict -> Bool` for this.

You can use the following *invariant wrapper*:

```
invar :: (a -> Bool) -> (a -> a) -> a -> a
invar p = assert (\ x y -> not (p x) || p y)

assert :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert p f x = let x' = f x in
   if p x x' then x' else error "assert"
```

Answer

```
insertLemma :: (String,String) -> Dict -> Dict
insertLemma (u,v) Leaf = B (u,v) Leaf Leaf
insertLemma (u,v) (B (w,x) left right) =
  if u == w then error "lemma already present"
  else if u < w then
    B (w,x) (insertLemma (u,v) left) right
  else
    B (w,x) left (insertLemma (u,v) right)
```

An automated test:

```
insertLemma' p = invar ordered (insertLemma p)
```

This test will always succeed. For the key question in checking whether `insertLemma` is correct is: does it always transform an ordered dictionary into an ordered dictionary? There are three cases to consider: (1) attempt to insert a lemma that is already present. In this case the dictionary is not changed, so the order is trivially preserved. (2) Insert a lemma that is alphabetically before the lemma at the root of the tree. In this case the lemma is inserted in the left subtree, and our induction hypothesis yields that the order of that subtree gets preserved by the insertion. So the whole tree remains ordered. (3) Insert a lemma that is alphabetically after the lemma at the root of the tree. In this case the lemma is inserted in the right subtree, and again the induction hypothesis yields that the order of the subtree, and hence of the whole tree, is preserved.

Question 6 You are still leading the team mentioned in a previous exercise. You realize that for your integrity tests on dictionaries it is crucial that you have a good implementation of the check for whether a dictionary is ordered. You have asked your two senior engineers Dr. Sharp and Mr. Curry to implement such checks.

Dr. Sharp comes up with the following proposal.

```
ordered1 :: Dict -> Bool
ordered1 Leaf = True
ordered1 (B _ Leaf Leaf) = True
ordered1 (B x Leaf (B x1 l r)) =
  lemma x < lemma x1 && ordered1  (B x1 l r)
ordered1 (B x (B x1 l r) Leaf) =
  ordered1 (B x1 l r) && lemma x1 < lemma x
ordered1 (B x (B x1 l1 r1) (B x2 l2 r2)) =
  lemma x1 < lemma x && lemma x < lemma x2 &&
  ordered1 (B x1 l1 r1) && ordered1 (B x2 l2 r2)
```

Mr. Curry comes up with something quite different:

```
ordered2 :: Dict -> Bool
ordered2 dict = ordered' [] [] dict where
  ordered' :: [String] -> [String] -> Dict -> Bool
  ordered' _ _ Leaf = True
  ordered' xs ys (B u left right) = let z = lemma u in
      (xs == [] || z > head xs) &&
      (ys == [] || z < head ys) &&
       ordered' xs [z] left && ordered' [z] ys right
```

Your task is to find out which implementation is correct, and to make a decision about which code to adopt for your dictionary software. But maybe they are both wrong, or both correct. Write a short note to your senior engineers in which you assess their proposals, and motivate your decision about which method to adopt.

Answer

Dear Dr. Sharp, dear Mr. Curry,

Thank you for your proposals. I decided to use an easy check on both of them, by comparing your proposals with my own simple-minded solution. I am sure you both will agree that a dictionary is ordered if and only if an in-order traversal of the dictionary tree yields a list where the lemmas appear in alphabetical order. So here is my check:

```
ordered :: Dict -> Bool
ordered tree = ordered' (inOrder tree) where
  ordered' [] = True
  ordered' [x] = True
  ordered' (x:y:zs) = lemma x < lemma y && ordered' (y:zs)
```

I hope you do agree that the following tests on your code are fair:

```
sharpTest :: Dict -> Bool
sharpTest = \ d -> ordered1 d == ordered d

curryTest :: Dict -> Bool
curryTest = \ d -> ordered2 d == ordered d
```

With this test, I found the following simple counterexample to Dr. Sharp's code:

```
example = B ("c","..")
            (B ("b","..") (B ("a","..") Leaf Leaf)
                          (B ("d","..") Leaf Leaf))
            Leaf
```

This exposes a problem in Dr. Sharp's code that is absent from Mr. Curry's implementation. Mr. Curry's implementation keeps track of the upper and lower bounds for the lemmas in a subtree. This information is necessary to implement the part of the definition that says "the lemmas of the items on *all* nodes in the left subtree are below the lemma of the item at the current node" (and similarly for the right subtree).

I don't think the flaw in Dr. Sharp's code is easy to repair, for I noticed that the code does not keep track of the upper and lower bounds for the lemmas in a subtree.

I did not find counterexamples to the implementation of Mr. Curry. Since his code is more efficient than the code of my own reference test, I have decided that we will use Mr. Curry's implementation in our production code.

With best regards,

The Boss