```haskell
module Workshop6Answers where

import Data.List
import Data.Char
```

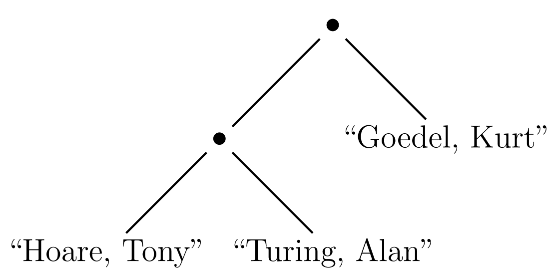
Workshop Testing and Formal Methods, Week 6, with Answers

The topic of this workshop is tree manipulation problems and tree algorithms. We start with a definition of trees with two branches and information at the leaves, so-called binary leaf trees.

Binary trees with information at the leaf nodes (binary leaf trees) are defined by:

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

Example binary tree, with information consisting of strings at the leaf nodes:



A Haskell version of the example:

```haskell
exampleTree :: Blt String
exampleTree = Node (Node (Leaf "Hoare, Tony")
                         (Leaf "Turing, Alan"))
                   (Leaf "Goedel, Kurt")
```

Question 1 Define a function `leafCount :: Blt a -> Int` that counts the number of leaf nodes in a binary tree.

How can you test `leafCount` for correctness? Or can you perhaps *prove* that it is correct?

Answer

```
leafCount ::  Blt a -> Int
leafCount (Leaf _) = 1
leafCount (Node left right) = leafCount left + leafCount right
```

An inductive argument shows that this is correct. A leaf tree has indeed one leaf, and if the leaf counts of the two subtrees of a non-leaf tree are given, summing them gives the correct count for the total number of leaves.

Question 2 Define a function `mapB` that does for binary trees what `map` does for lists. The type is:

```
mapB :: (a -> b) -> Blt a -> Blt b
mapB f = error "not implemented"
```

Example of the use of `mapB`:

```
*Workshop6> mapB (map toUpper) exampleTree
Node (Node (Leaf "HOARE, TONY") (Leaf "TURING, ALAN")) (Leaf "GOEDEL, KURT")
```

Answer

```
mapB :: (a -> b) -> Blt a -> Blt b
mapB f (Leaf x) = Leaf (f x)
mapB f (Node t1 t2) = Node (mapB f t1) (mapB f t2)
```

We move on to a definition of trees with an *arbitrary* number of branches, and information at the internal nodes. Here is a datatype for trees with *lists* of branches:

```
data Tree a = T a [Tree a] deriving (Eq,Ord,Show)
```

Here are some example trees:

```
example1 = T 1 [T 2 [], T 3 []]
example2 = T 0 [example1,example1,example1]
```

This gives:

```
*Workshop6> example1
T 1 [T 2 [],T 3 []]
*Workshop6> example2
T 0 [T 1 [T 2 [],T 3 []],T 1 [T 2 [],T 3 []],T 1 [T 2 [],T 3 []]]
```

Question 3 Define a function `count :: Tree a -> Int` that counts the number of nodes of a tree.

How can you test `count` for correctness? Or can you perhaps *prove* that it is correct?

Answer Here is the implementation:

```
count :: Tree a -> Int
count (T _ ts) = 1 + sum (map count ts)
```

This gives the correct result for the example trees above. But we can also prove that it is correct. The simplest possible tree has the form `T x []`. This tree has one node, so the count function gives the right result. Suppose the count function gives the right result for all trees in a non-empty tree list `ts`, then the whole tree has the sum of all these counts plus one as its count. That is precisely what the definition gives us.

Question 4 Consider the following function `depth :: Tree a -> Int` that gives the depth of a `Tree`.

```
depth :: Tree a -> Int
depth (T _ ts) = foldl max 0 (map depth ts) + 1
```

How can you test `depth` for correctness? Or can you perhaps *prove* that it is correct?

Answer The function gives the correct result on the example trees above. But again we can do better than test. The function is correct for trees without subtrees, for these have depth 0. If the function is correct for all trees in `ts`, then it is correct of trees of the form `T _ ts`, for these have depth equal to the maximum of the depth of the subtrees, plus one.

Here is an alternative definition:

```
depth1 :: Tree a -> Int
depth1 (T _ []) = 0
depth1 (T _ ts) = maximum (map depth1 ts) + 1
```

Note that the extra clause for the empty tree is necessary now, for `maximum []` generates an error.

Question 5 Define a function `mapT` that does for trees what `map` does for lists. The type is:

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

Example of the use of `mapT`:

```
*Workshop6> mapT succ example1
T 2 [T 3 [],T 4 []]
*Workshop6> mapT succ example2
T 1 [T 2 [T 3 [],T 4 []],T 2 [T 3 [],T 4 []],T 2 [T 3 [],T 4 []]]
```

Hint: in the definition you will need both `map` and `mapT`.

Answer

```
mapT :: (a -> b) -> Tree a -> Tree b
mapT f (T x ts) = T (f x) (map (mapT f) ts)
```

Question 6 How can you test `mapT` from the previous question for correctness? Or can you perhaps *prove* that it is correct?

Answer The calls `mapT succ example1` and `mapT succ example2` yield the expected outputs. You can devise more tests.

But again, a correctness proof by induction is possible, and this is more appropriate than a test. The base case is the case of a tree of the form `T x []`. Here the result should be `T (f x) []`, and that is what the function gives us. For the recursive case, assume `mapT f` yields the correct result of all trees in `ts`. Then `map` collects these results, and finally `(f x)` is put at the top node. This is also correct.

Question 7 Write a function `collect` that collects the information in a tree of type `Tree a` in a list of type `[a]`. The type specification is

```
collect :: Tree a -> [a]
```

Here is an example call:

```
*Workshop6> collect example2
[0,1,2,3,1,2,3,1,2,3]
```

Answer

```
collect :: Tree a -> [a]
collect (T x ts) = x : concatMap collect ts
```

A fold operation on trees can be defined by

```
foldT :: (a -> [b] -> b) -> Tree a -> b
foldT f (T x ts) = f x (map (foldT f) ts)
```

Question 8 Redefine `count`, `depth`, `collect` and `mapT f` in terms of `foldT`.

Answer

```
count' :: Tree a -> Int
count' = foldT (\ _ ns -> sum ns + 1)
```

```
depth' :: Tree a -> Int
depth' = foldT (\ _ ds -> if null ds then 0 else maximum ds + 1)
```

```
collect' :: Tree a -> [a]
collect' = foldT (\ x lists ->  x : concat lists)
```

```
mapT' :: (a -> b) -> Tree a -> Tree b
mapT' f = foldT (\ x ts ->  T (f x) ts)
```

—

If you have a step function of type `node -> [node]` and a seed, you can grow a tree, as follows.

```
grow :: (node -> [node]) -> node -> Tree node
grow step seed = T seed (map (grow step) (step seed))
```

Example:

```
*Workshop6> grow  (\x -> if x < 2 then [x+1, x+1] else []) 0
T 0 [T 1 [T 2 [],T 2 []],T 1 [T 2 [],T 2 []]]
```

Question 9 Consider the following output:

```
*Workshop6> count (grow  (\x -> if x < 2 then [x+1, x+1] else []) 0)
7
```

Can you predict the value of the following:

```
count (grow  (\x -> if x < 6 then [x+1, x+1] else []) 0)
```

Answer

```
*Workshop6Answers> count (grow  (\x -> if x < 6 then [x+1, x+1] else []) 0)
127
```

Remark: the tree grown by `(\x -> if x < 6 then [x+1, x+1] else []) 0` is a balanced binary branching tree of depth 6.

As we have seen in a previous workshop, a balanced binary tree of depth n has $2^{n+1} - 1$ nodes. For a balanced binary tree of depth 6 this gives $2^7 - 1 = 127$ nodes.

—

Question 10 The above example trees were all finite. We can also grow infinite trees. Here is an example:

```
infTree :: Tree Integer
infTree = grow (\ n -> [n+1,n+1]) 0
```

In order to use these, we need a function for cutting off the tree at a given depth; the analogue of `take` for lists.

```
takeT :: Int -> Tree a -> Tree a
takeT = error ‘‘not yet implemented"
```

Implement this function.

Answer

```
takeT :: Int -> Tree a -> Tree a
takeT 0 (T x _)  = T x []
takeT n (T x ts) = T x (map (takeT (n-1)) ts)
```

This gives:

```
*Workshop5Answers> takeT 4 infTree
T 0 [T 1 [T 2 [T 3 [T 4 [],T 4 []],T 3 [T 4 [],T 4 []]],T 2 [T 3
 [T 4 [],T 4 []],T 3 [T 4 [],T 4 []]]],T 1 [T 2 [T 3 [T 4 [],T 4 []],
  T 3 [T 4 [],T 4 []]],T 2 [T 3 [T 4 [],T 4 []],T 3 [T 4 [],T 4 []]]]]
```

Question 11 Consider the following tree:

```
tree n = grow (f n) (1,1)

f n = \ (x,y) -> if x+y <= n then [(x+y,y),(x,x+y)] else []
```

Can you show that the number pairs (x, y) that occur in `tree n` are precisely the pairs in the set

$$\{(x, y) \mid 1 \leq x \leq n, 1 \leq y \leq n \text{ with } x, y \text{ co-prime}\}.$$

Hint: try to see the connection with Euclid's gcd algorithm. Euclid's gcd algorithm terminates with (1,1) for precisely the co-prime pairs of positive natural numbers.

Answer Every path through a tree, from leaf to root, is a sequence of computation steps according to Euclid's GCD algorithm: replace the larger of two numbers by their difference, and keep the smaller number. The fact that the sequence ends in the root $(1, 1)$ shows that the GCD of any number pair on the path equals 1, i.e., the two numbers are co-prime. Conversely, any pair of co-prime numbers can be thought of as the result of 'running Euclid's GCD algorithm backwards', and that is precisely what the tree expansion gives us.