```
module Workshop6 where

import Data.Char
import Data.List
```

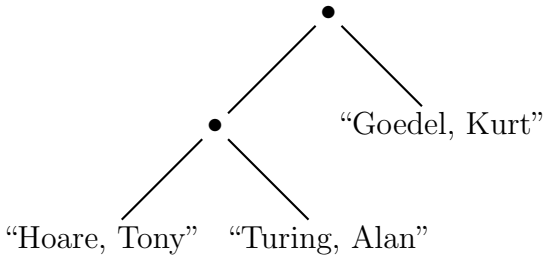
Workshop Testing and Formal Methods, Week 6

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:

```
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:

```
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?

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
```

Example of the use of `mapB`:

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

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?

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

```
depth :: Tree a -> Int
depth (T _ []) = 0
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?

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`.

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

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`.

—

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)
```

Question 10 Consider the following tree:

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

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