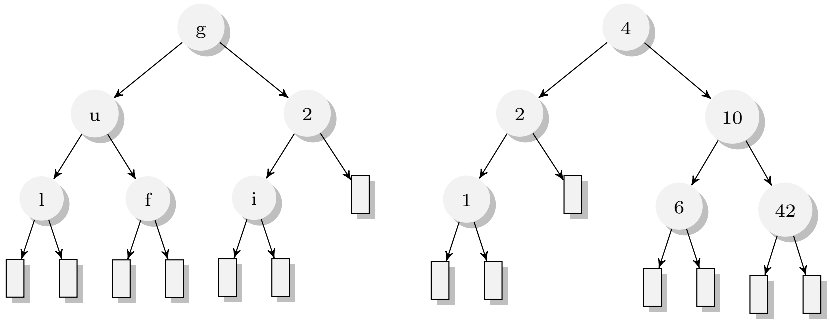
Notes:

- To solve the programming exercises you should use the Glasgow Haskell Compiler **GHC**, available for free at `http://www.haskell.org/ghc/`. You can use the command "`ghci`" to start an interactive interpreter shell.
- Please solve these exercises in **groups of four**!
- The solutions must be handed in **directly before (very latest: at the beginning of)** the exercise course on Tuesday, 03.05.2016, **16:15pm**, in lecture hall **AH 4**. Alternatively you can drop your solutions into a box which is located right next to Prof. Giesl's office (until 30 minutes before the exercise course starts).
- In addition, please send the solutions to programming exercises to lehre_lufgi2@cs.rwth-aachen.de.
- Please write the **names** and **immatriculation numbers** of all students on your solution. Also please staple the individual sheets!

Exercise 1 (Data Types): (2 + 1.5 + 2 + 2 = 7.5 points)

In this exercise we consider binary trees, where data is stored in inner nodes and leaf nodes are always empty. For instance consider the following two examples:



a) Give a definition for a data type `BinaryTree` for binary trees as described above. The data type should have the two constructors `Node`, for inner nodes, and `Leaf`, for leaf nodes.

Furthermore give a term `tree1` of type `BinaryTree Char` encoding the left example above and a term `tree2` of type `BinaryTree Int` encoding the right example.

b) Write a function `flattenTree` that transforms a binary tree into a list. The list should be constructed such that for every node N all elements in the left subtree come before the element stored in N and all elements from the right subtree come after the element stored in N.

For example `flattenTree tree1` should yield `"lufgi2"` and `flattenTree tree2` should return `[1, 2, 4, 6, 10, 42]`.

Also give the type declaration for the function `flattenTree`.

c) Write a function `elemTree` that checks if an object is contained in the tree.

For example `elemTree 'i' tree1` should return `True`, but `elemTree 3 tree2` should return `False`.

Also give a type declaration for `elemTree` with a sensible context for the type of the first parameter.

d) Write a function `isSorted` that checks if a given binary tree is a binary search tree, i.e., all elements in the left subtree of any node N are less than the element stored in N, and all elements in the right subtree of N are greater than the element stored in N.

For example `isSorted tree1` should return `False`, but `isSorted tree2` should return `True`.

Also give a type declaration for `isSorted` and a sensible context for all type variables.

Exercise 2 (Type classes): (1 + 1.5 + 2.5 + 3 = 8 points)

a) Consider the type `Nats` from the lecture that is defined as follows:

```
data Nats = Zero | Succ Nats deriving Show
```

Declare `Nats` as an instance of the type class `Eq` and implement the method `(==)` such that it computes equality between natural numbers.

b) Give a declaration for a type class `Ordered` as a subclass of `Eq` with the following methods:

- `lt :: a -> a -> Bool` (like "**l**ess **t**han" on numbers).
- `gt :: a -> a -> Bool` which is a mirrored version of `lt` (like "**g**reater **t**than" on numbers).

Give a default implementation for `gt`, the function `lt` has to be implemented in the instances of the type class `Ordered`.

c) Declare the built-in type `Integer`[1] and `Nats` from a) as instances of the type class `Ordered`. Implement `lt` and `gt` as the "less than" and "greater than" relation on numbers respectively for both instances. Only provide an implementation if the default implementation is not sufficient.

For example `lt -3 2`, `lt Zero (Succ (Succ Zero))` and `gt (Succ Zero) Zero` is `True`, but `lt 0 0`, `lt (Succ Zero) Zero` and `gt -5 3` is `False`.

d) Implement a function `sortAsc` that sorts lists of type `[a]` in ascending order, provided that `a` is an instance of the type class `Ordered`. So if `lt x y` then `x` comes before `y` in the sorted list.

Give a type declaration with a sensible context in addition to the implementation.

For example `sortAsc [3, -5, 24, 5, 3, 14]` should yield `[-5, 3, 3, 5, 14, 24]` and `sortAsc [Succ (Succ Zero), Zero, Succ Zero]` should yield `[Zero, Succ Zero, Succ( Succ Zero)]`.

You may of course write auxiliary functions and you may use built-in functions *except* built-in sorting functions.

Hints:

- You may use the built-in function `filter :: (a -> Bool) -> [a] -> [a]` in your sorting function.

Exercise 3 (Using Higher-Order Functions): (1.5 + 1 = 2.5 points)

In this exercise you may *not* use any predefined functions on lists except `map`, `foldr`, and `filter`. Also, you may *not* use explicit recursion.

a) Implement a function `length' :: [a] -> Int` that computes the length of a list.

b) Implement a function `countLetters :: [Char] -> Int` that counts the number of letters in a string. For example the number of letters in `"Exercise Sheet 2: Exercise 3"` is 21.

Hints:

- The function `isLetter :: Char -> Bool` returns `True` for all characters that are considered to be a letter. To use it, write `import Data.Char` as the first line of your source file.

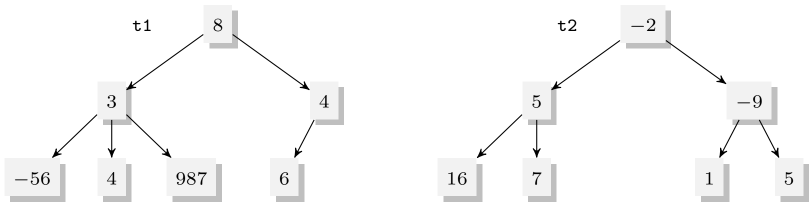


Figure 1: Two trees of type `MultTree Int`

Exercise 4 (Defining Higher-Order Functions): (3 points)

Consider the following data type which represents non-empty trees whose nodes may have arbitrary many children:

```
data MultTree a = MultNode a [MultTree a] deriving Show
```

Figure 1 shows two examples for such a trees.

a) Write a function `zipWithMult :: (a -> b -> c) -> MultTree a -> MultTree b -> MultTree c` that behaves similar to the function `zipWith` for lists, i.e., it combines the two input trees using the given function of type `a -> b -> c`. Nodes that have no corresponding node in the other tree are dropped. In this way, a new `MultTree c` is constructed.

For example, the application of the function `zipWithMult (+)` to the trees shown in Figure 1 results in the tree shown in Figure 1

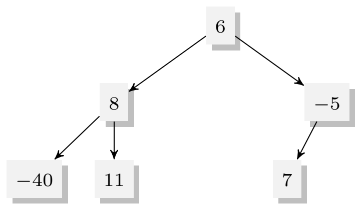


Figure 2: Result of `zipWithMult (+) t1 t2`

Hints:

- The predefined function `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` combines two lists. The result of `zipWith (*) [1, 2, 3] [4, 5]` is `[1 * 4, 2 * 5]`. Its implementation behaves similar to:

```
zipWith _ [] _           = []
zipWith _ _ []           = []
zipWith f (x:xs) (y:ys) = (f x y):(zipWith f xs ys)
```

[1] `Integer` is a type for arbitrary large intger numbers, whereas `Int` represents integers of fixed range (implementation defined, at least 30 bit).