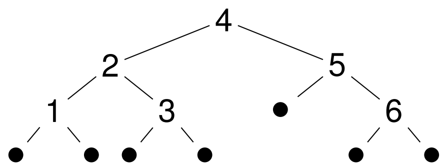
Tutorial 6A: Trees

In this tutorial we will look at trees, constructed as **recursive data types**. First we will look at the binary trees of the lectures, which store integers at the nodes (but not the leaves):

```
data IntTree = Empty | Node Int IntTree IntTree
  deriving Show

t :: IntTree
t = Node 4 (Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty))
           (Node 5 Empty (Node 6 Empty Empty))
```

We can draw the tree `t` as follows, using • for `Empty`. Note that each internal (i.e. non-leaf) node has three attributes: an integer, and two children, each itself a (sub)tree. Also, this is a computer-science tree: these grow from the ceiling down, as opposed to mathematical trees which grow from the ground up.



"`Deriving Show`" tells Haskell to make a default instance of the `Show` class for the `IntTree` type. It creates a literal representation of the data type: try it out with `*Main> t`.

Exercise 1: Complete the following functions.

a) `isEmpty`: determines whether a tree is `Empty` or not.

b) `rootValue`: returns the integer at the root of the tree, or zero for an empty tree.

c) `height`: returns the height of the tree. A leaf has height zero, and a node is one higher than its highest subtree. The function `max` will be helpful.

d) `member`: returns whether an integer occurs in the tree.

e) `paths`: **(Optional challenge)** return all the paths through the tree to where the integer was found, as lists of zeroes and ones, where zero means "left" and one means "right".

```
*Main> isEmpty t
False
*Main> rootValue t
4
```

```
*Main> height t
3
*Main> member 3 t
True
*Main> paths 3 t
[[0,1]]
*Main> paths 9 t
[]
*Main> paths 3 (Node 3 t t)
[[],[0,0,1],[1,0,1]]
```

For better readability, we can make our own `Show` instance, with our own `show` function for trees. We print a tree sideways, with the root to the left, and using indentation to indicate the parent-child relation. Browsing directories on Windows uses this, for instance. Comment out the line `deriving Show`, and un-comment the given `Show` instance. Try it out: a `+` indicates the root of a (sub)tree, connected to its parent with `|`.

```
*Main> t
    +-1
  +-2
  | +-3
+-4
  +-5
    +-6
```

Lambda-terms

In this part of the tutorial we will start implementing the λ-calculus. We will build a data type `Term` for terms, use it to build some example λ-terms, and give the functions `used` and `free` to find the used variables and the free variables in a term.

The **terms** of the λ-calculus are given by the following grammar.

$$N ::= x \mid \lambda x.N \mid N N$$

The set of **used variables** of a term M, $\mathsf{UV}(M)$, is defined by:

$$\begin{aligned}
\mathsf{UV}(x) &= \{x\} \\
\mathsf{UV}(\lambda x.M) &= \mathsf{UV}(M) \cup \{x\} \\
\mathsf{UV}(M N) &= \mathsf{UV}(M) \cup \mathsf{UV}(N)
\end{aligned}$$

The set of **free variables** of a term M, $\mathsf{FV}(M)$, is defined by:

$$\begin{aligned}
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(\lambda x.M) &= \mathsf{FV}(M) \setminus \{x\} \\
\mathsf{FV}(M N) &= \mathsf{FV}(M) \cup \mathsf{FV}(N)
\end{aligned}$$

Exercise 2:

a) Complete the type `Var` for **variables** as a synonym for `String`. In your file, it is given with the unit type `()` as a placeholder; replace this with your solution.

b) Complete the data type `Term`. Use the constructors `Variable`, `Lambda`, and `Apply`, and give each the right kinds of arguments according to the definition of λ-terms. Un-comment the declaration `deriving Show` and the example term `example` to test your solution. **Hint:** there is a solution in Lecture 3B. This uses `String` directly instead of `Var`, which you would still need to fix.

c) Un-comment the function `pretty` to display a λ-term nicely. Replace the declaration `deriving Show` for the data type `Term` with a `Show` instance for terms that uses `pretty`.

d) Build the terms $N_1 = \lambda x.x$, $N_2 = \lambda x.(\lambda y.x)\,z$ and $N_3 = (\lambda x.\lambda y.x\,y)\,(\lambda x.x)$ as `n1`, `n2`, and `n3`.

e) Complete the function `used` that collects the used variables in a term in an alphabetically ordered list without duplicates. Use the given function `merge`.

f) Complete the function `free` that collects the free variables in a term in an alphabetically ordered list without duplicates. Use the functions `merge` and `minus`.

```
*Main> example
Lambda "a" (Lambda "x" (Apply (Apply (Lambda "y" (Apply (Variable
 "a") (Variable "c"))) (Variable "x")) (Variable "b")))
*Main> pretty example
"\\a. \\x. (\\y. a c) x b"
*Main> example   -- after creating a Show instance using "pretty"
\a. \x. (\y. a c) x b
*Main> n1
\x. x
*Main> n2
\x. (\y. x) z
*Main> n3
```

```
(\x. \y. x y) (\x. x)
*Main> used n3
["x","y"]
*Main> used example
["a","b","c","x","y"]
*Main> free n2
["z"]
*Main> free example
["b","c"]
```