# Functional Programming – Series 3

**Preparation.** This series of exercises is about *trees*. In order to show trees in a graphical way, you have to install the module *twentefp-eventloop-trees*, by the (shell) commands (see also the explanation on Blackboard, under `Tooling` → `Graphics for trees in Haskell`):

```
cabal update
cabal install twentefp-eventloop-trees
```

You can use the functionalities of this package by including the following line in your program:

**import** *FPPrac.Trees*

Now you have the type *RoseTree*, defined as

**data** *RoseTree* = *RoseNode String* [*RoseTree*]

at your disposal, as well as the pre-defined tree

*roseExampleTree*,

and the functions

*showRoseTree*, *showRoseTreeList*.

These functions will show the trees in a browser window, for that you have to download from Blackboard the file (under `Tooling` → `Graphics for trees in Haskell`:

`standard_webpage_18-3-17.zip`

Then unpack the `.zip` file, and open the file `standard_webpage.html` in a browser[1].

Evaluate the following expressions (check your browser for the output):

*roseExampleTree*

*showRoseTree roseExampleTree*

[1]NOTE: not all browsers work equally well because of different choices browsers make concerning all sorts of technical details. However, *Chrome* and *Internet Explorer* work well.

One further suggestion is to open only *one* tab in your browser, in order to let the communication between Haskell and the browser go smoothly. For more information on some problems with specific browsers, see the above mentioned manual.

**Remark.** In all exercises below you should demonstrate your functions graphically, using your own example trees.

**Exercise 1.** To graphically show trees of a type different from *RoseTree*, they have to be translated into *RoseTree*s first.

**a.** Given is the type of binary trees with *Int*s at the internal nodes and at the leaves:

$$\begin{aligned}\textbf{data}\ \mathit{Tree1a} = \ & \mathit{Leaf1a}\ \mathit{Int} \\ & \mid \mathit{Node1a}\ \mathit{Int}\ \mathit{Tree1a}\ \mathit{Tree1a}\end{aligned}$$

Define a function *pp1a* (for “pre-processor”) which translates a tree of type *Tree1a* into a tree of type *RoseTree*.

**b.** Define a type *Tree1b* for binary trees which contain 2-tuples of *Int*s at the internal nodes and at the leaves. Define a function *pp1b* which transforms trees of type *Tree1b* into trees of type *RoseTree*.

**c.** Ibid for a type *Tree1c* which contains *Int*s at the leaves and no information at the internal nodes (*hint*: use empty strings in your rose trees).

**d.** Finally, define the type *Tree1d* that has 2-tuples of *Int*s at the leaves, and no information at the internal nodes. It should be possible that a tree of this type has any number of subtrees in its internal nodes.

**Exercise 2.**

**a.** Define a function *treeAdd* that adds a number $x$ to every number in a tree of type *Tree1a*.

**b.** Define a function *treeSquare* that squares every number in a tree of type *Tree1a*.

**c.** Define a function

$$\mathit{mapTree} :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Tree1a} \to \mathit{Tree1a}$$

which applies a function $f$ of type $\mathit{Int} \to \mathit{Int}$ to every number in a tree of type *Tree1a*.

Define the functions of parts $a$ and $b$ using *mapTree*.

**d.** Define a function *addNode* which replaces every 2-tuple in a tree of type *Tree1a* by the sum of the numbers in each 2-tuple.

**e.** Define a variant of *mapTree* such that a function $f$ of type

$$(Int, Int) \rightarrow Int$$

can be applied to every 2-tuple in a tree of type *Tree1b*. Demonstrate your function with a few binary operations such as additions and multiplication (hint: use lambda-abstraction).

**Exercise 3.**

**a.** Define a function *binMirror* which mirrors a tree of type *Tree1a*. Check that mirroring twice returns the original tree again.

**b.** Write a variant of the mirror function which works for trees of type *Tree1d* such that also all tuples at the leaves will be swapped.

**Definition.** A binary tree with numbers is called *sorted* if for every node in the tree it holds that every number in the *left* subtree is smaller than or equal to the number in that node, and every number in the *right* subtree is larger.

**Exercise 4.** In this exercise we use trees with *Int*s at the internal nodes and nothing at the leaves. Define a type *Tree4* for such trees.

**a.** Write a function *insertTree* which inserts a number in a *sorted* tree of type *Tree4*. *Hint*: it is practical to insert a number at a leaf.

**b.** Write a function *makeTree* which produces a sorted tree from an unsorted list of numbers, by using the function *insertTree*.

Write your function in two ways: by recursion, and by *foldl* or *foldr*.

**c.** Write a function *makeList* which delivers the list of all numbers in a tree. If that tree is sorted, the list should maintain the sorting.

**d.** Combine functions above to sort a list.

**e.** The converse of **d**: combine the functions to sort a tree of type *Tree4*.

**Exercise 5.** Write a function *subtreeAt* which searches in a *sorted* tree of type *Tree4* the subtree at a node with a given number $n$.

If the number $n$ does not occur in the tree, your function should give an error message.

**Exercise 6.** Define a function *cutOffAt* which cuts off all branches in a tree of type *Tree1a* at a given depth, leaving shorter branches unchanged. As a result, an internal node may change into a leaf.

**Exercise 7.** A *path* in a binary tree is a string consisting of the characters 'l' and 'r', for "left" and "right" (respectively), indicating how to reach a node by starting at the root of the tree.

Define your functions in this exercise for type *Tree1a*.

**a.** Write a function *replace* which replaces the number at a node in some tree by a given number, if this node is indicated by a path.

**b.** Write a function *subTree* which returns the subtree indicated by a path. If the path is too long, an error message should be given.

**c – extra.** A leaf is a *neighbour* of another leaf if, when going through the leaves from left to right, there are no other leaves "in between".

Write two functions *leftNeighbour* and *rightNeighbour* which return the the path to the left and right neighbour (respectively) of a leaf, indicated by a path as well.

Show the result graphically by adding a specific number to the respective leaves.

**Exercise 8.** A binary tree is *balanced* if the difference in length of any two branches is at most one.

**a.** Write a function *isBalanced* which checks whether a tree of type *Tree4* is balanced.

**b.** Write a function *balance* which turns a tree of type *Tree4* into a balanced tree.

*Hint*: use lists as an intermediary step.

**c.** Check the result of *balance* from part **b** with the function *isBalanced* from part **a**.