

Prof. Dr. Jürgen Giesl
Peter Schneider-Kamp
Stephan Swiderski

Exercises *Functional Programming* – Sheet 3

Solutions will be collected until Wednesday, May 2, 2007 in the exercise course.

Exercises can be solved both in English and in German.

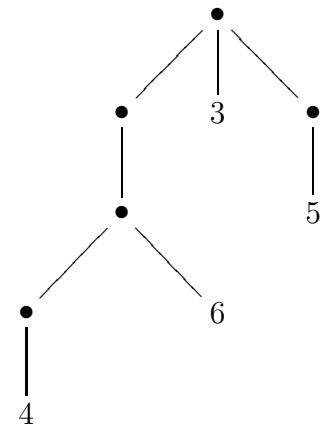
Exercise 1 (2+2+1 points)

The following data structure represents trees, where only leaves are labelled:

```
data Tree a = Node [Tree a] | Leaf a
```

Consider the tree t of integers. The representation of t as an object of type `Tree Int` in Haskell would be:

```
Node [Node [Node [Node [Leaf 4],  
Leaf 6]],Leaf 3,Node [Leaf 5]]
```



Implement the following functions in Haskell.

- The function `mapTree` of type `mapTree :: (a -> b) -> Tree a -> Tree b` applies a function to all values in the nodes of a tree. For example, `mapTree (*2) t` should return the following tree:

```
Node [Node [Node [Node [Leaf 8],Leaf 12]],Leaf 6,Node [Leaf 10]]
```
- The function `foldTree` of type `([b] -> b) -> (a -> b) -> Tree a -> b` works as follows: `foldTree n 1 t` replaces all occurrences of the constructor `Node` in the tree t by `n` and it replaces all occurrences of the constructor `Leaf` in t by `1`. So for the tree t above, `foldTree sum (\x -> x) t` should return 18. Here, `Node` is replaced by `sum` and `Leaf` is replaced by `(\x -> x)`.
- Use the `foldTree` function from (b) to implement the `maxTree` function which returns the largest (w.r.t. `>`) label of the tree. Apart from the function declaration, also give the type declaration for `maxTree`. So for the tree t above, `maxTree t` should return 6. This function is only defined for trees which are fully labelled (i.e. there are no terms of the form `Node []` inside a given tree).

Exercise 2 (1+2+2 points)

Implement the following functions in Haskell and also give their type declarations.

- (a) Write the functions $*|$ and $|-|$ from exercise sheet 1, by using the pre-defined higher-order functions `zipWith` and `map`.

Hints:

- `zipWith` combines two list with a given combinator:
`zipWith f [a1, ..., an] [b1, ..., bn]` returns `[f a1 b1, ..., f an bn]`
- $*|$ is the multiplication of a scalar with a vector:
`x *| [y1, ..., yn]` returns `[x * y1, ..., x * yn]`
- $|-|$ is the component-wise subtraction of vectors:
`[x1, ..., xn] |-| [y1, y2, ..., yn]` returns `[x1 - y1, ..., xn - yn]`

- (b) Given a list of functions `[f1, f2, ..., fk]` and a value `x`, the function `combine` returns the function `f1(f2... (fk x) ...)`.

For example, `combine [(*2), \x -> div x 3] 12` should return 8. Implement `combine` using pre-defined higher-order functions (e.g., `foldr`) when appropriate, but you may not use recursion in your declarations.

- (c) Given a list of functions `[f1, f2, ..., fk]` and a list of values `[x1, ..., xk]`, the function `multiApply` returns the list `[f1 x1, f2 x2, ..., fk xk]`.

For example, `multiApply [(*2), \x -> div x 3] [2, 18]` should return `[4, 6]`. Implement `multiApply` using pre-defined higher-order functions (e.g., `zipWith`) when appropriate, but you may not use recursion in your declarations.

Exercise 3 (1+1 points)

Implement the following function only with the help of list comprehensions, i.e., if you have to implement a function `f` then you should use only two declarations of the following form (where `x, y` are variables):

```
f :: ...
```

```
f ... = [(x,y) | ...]
```

- (a) Given a natural number `n > 0`, the function `productPairs` computes the list of all tuples `(x, y)` of natural numbers where their product is `n`.

For example, `productPairs 4` should return exactly `[(1,4), (2,2), (4,1)]`.

- (b) Given two lists `xs` and `ys`, the function `crossProduct` computes a list `zs` such that `(x, y) ∈ zs` if and only if `x ∈ xs` and `y ∈ ys`.

For example, `crossProduct [1,2,3] [3,4]` should return exactly the list `[(1,3), (2,3), (3,3), (1,4), (2,4), (3,4)]`.