5. Übung zur Vorlesung "Fortgeschrittene funktionale Programmierung"

Laziness und Faltung

Boris Dudelsack

4. November 2016

Aufgabe 1: Ströme

```haskell
data Stream a = Cons a (Stream a)

streamToList :: Stream a → [a]
streamToList (Cons x xs) = x : streamToList xs

repeatStream :: a → Stream a
repeatStream x = Cons x (repeatStream x)

mapStream :: (a → b) → Stream a → Stream b
mapStream f (Cons x xs) = Cons (f x) (mapStream f xs)

iterateStream :: (a → a) → a → Stream a
iterateStream f x = Cons x (iterateStream f (f x))

nats :: Stream Integer
nats = iterateStream (+1) 0

interleaveStream :: Stream a → Stream a → Stream a
interleaveStream (Cons x xs) ys = Cons x (interleaveStream ys xs)

ruler :: Stream Integer
ruler = interleaveStream (repeatStream 0) (mapStream (+1) ruler)
```

Aufgabe 2: Baum-Durchläufe

```
minTree :: BinTree Int → Int
minTree Empty        = maxBound
minTree (Node x y z) = min x (min (minTree y) (minTree z))

replace :: BinTree a → b → BinTree b
replace Empty _        = Empty
replace (Node x y z) v = Node v (replace y v) (replace z v)

replaceMinRec :: BinTree Int → a → (BinTree a, Int)
replaceMinRec t v = (t’, m)
  where
    m = minTree t
    t’ = replace t v

-- wie benutze ich hier replaceMinRec?

replaceMin :: BinTree Int → BinTree Int
replaceMin t = t’
  where
    m = minTree t
    t’ = replace t m
```

Aufgabe 3: Faltungen

```
map' :: (a → b) → [a] → [b]
map' f = foldr (\x xs → f x : xs) []

map'' :: (a → b) → [a] → [b]
map'' f = foldl (\xs x → f x : xs) []

filter' :: (a → Bool) → [a] → [a]
filter' p = foldr (\x xs → if p x then x : xs else xs) []

filter'' :: (a → Bool) → [a] → [a]
filter'' p = foldl (\xs x → if p x then x : xs else xs) []

reverse' :: [a] → [a]
reverse' = foldr (\x xs → xs ++ [x]) []

reverse'' :: [a] → [a]
reverse'' = foldl (flip (:)) []

concat' :: [a] → [a] → [a]
concat' = foldr (\x xs → xs ++ [x])

concat'' :: [a] → [a] → [a]
concat'' = foldl (flip (:))
```