PROGRAMMIERUNG

ÜBUNG 4: FUNKTIONEN HÖHERER ORDNUNG

Eric Kunze
eric.kunze@tu-dresden.de

, 02. Mai 2022

letzte Änderung:
27.04.2022, 13:01

Übungsblatt 4 — Aufgabe 1

Algebraische Datentypen

AUFGABE 1 – TEIL (A)

Anzahl der Blätter

```haskell
data RoseTree = Node Int [RoseTree]
countLeaves :: RoseTree -> Int
```

AUFGABE 1 – TEIL (A)

Anzahl der Blätter

```haskell
data RoseTree = Node Int [RoseTree]
countLeaves :: RoseTree -> Int
```

```haskell
countLeaves :: RoseTree -> Int
countLeaves (Node _ [] )     = 1
countLeaves (Node _ [t])     = countLeaves t
countLeaves (Node x (t:ts))
  = countLeaves t + countLeaves (Node x ts)
```

AUFGABE 1 – TEIL (B)

gerade Anzahl an Kindern

```haskell
data RoseTree = Node Int [RoseTree]
evenNodes :: RoseTree -> Bool
```

AUFGABE 1 – TEIL (B)

gerade Anzahl an Kindern

```haskell
data RoseTree = Node Int [RoseTree]
evenNodes :: RoseTree -> Bool
```

```haskell
evenNodes :: RoseTree -> Bool
evenNodes (Node _  []       ) = True
evenNodes (Node x  [t]      ) = False
evenNodes (Node x (t1:t2:ts))
  = evenNodes (Node x ts) && evenNodes t1 &&
    evenNodes t2
```

AUFGABE 1 – TEIL (B)

gerade Anzahl an Kindern

```haskell
data RoseTree = Node Int [RoseTree]
evenNodes :: RoseTree -> Bool
```

```haskell
evenNodes' :: RoseTree -> Bool
evenNodes' (Node _ []) = True
evenNodes' (Node _ ts)
  = mod (length ts) 2 == 0 && evenNodes'' ts
  where
    evenNodes'' :: [RoseTree] -> Bool
    evenNodes'' []     = True
    evenNodes'' (t:ts)
      = evenNodes' t && evenNodes'' ts
```

Übungsblatt 4 — Aufgaben 2 & 3

Funktionen höherer Ordnung

FUNKTIONEN

Wir kennen bereits einige Möglichkeiten Funktionen zu notieren. Hier seien einige weitere erwähnt.

anonyme Funktionen. Funktionen ohne konkreten Namen

z.B. `(\x -> x+1)` ist die Addition mit `1`

```
1 (\x -> x+1) 4 = 5
```

FUNKTIONEN

Wir kennen bereits einige Möglichkeiten Funktionen zu notieren. Hier seien einige weitere erwähnt.

anonyme Funktionen. Funktionen ohne konkreten Namen
z.B. `(\x -> x+1)` ist die Addition mit `1`

```
(\x -> x+1) 4 = 5
```

Operator $\leftrightarrow$ Funktion Aus Operatoren (wie z.B. +) kann man eine Funktion machen und vice versa.

- Operator $\rightarrow$ Funktion: Klammern

```
(+) :: Int -> Int -> Int
(+) x y =  x + y
```

- Funktion $\rightarrow$ Operator: Backticks ‘...‘

```
5 ‘mod‘ 2 = 1
```

FUNKTIONSKOMPOSITION

Analog zur mathematischen Notation $f = g \circ h$ für $f(x) = g(h(x))$ versteht auch Haskell das Kompositionsprinzip mit dem Operator `.`
z.B.

```
sqAdd :: Int -> Int
sqAdd = (^2) . (+ 5)
```

statt `sqAdd x = (x + 5)^2` für das Quadrat des fünften Nachfolgers

PARTIELLE APPLIKATION

Funktionen müssen nicht immer mit allen Argumenten versorgt werden. Lässt man (hintere) Argumente weg, so spricht man von Unterversorgung. Die Modulo Funktion hat eigentlich zwei Argumente. Lassen wir das zweite Argument weg, so liefert dies uns eine neue Funktion, die noch ein Argument entgegennimmt und sodann die Restberechnung ausführt.

```
1 mod :: Int -> Int -> Int
2 mod m n = ...
3
4 mod 10 :: Int -> Int
5 (mod 10) n = mod 10 n
```

PARTIELLE APPLIKATION

Funktionen müssen nicht immer mit allen Argumenten versorgt werden. Lässt man (hintere) Argumente weg, so spricht man von Unterversorgung. Die Modulo Funktion hat eigentlich zwei Argumente. Lassen wir das zweite Argument weg, so liefert dies uns eine neue Funktion, die noch ein Argument entgegennimmt und sodann die Restberechnung ausführt.

```
1 mod :: Int -> Int -> Int
2 mod m n = ...
3
4 mod 10 :: Int -> Int
5 (mod 10) n = mod 10 n
```

```
1 (> 3) :: Int -> Bool
2 (> 3) x = x > 3
```

FUNKTIONEN HÖHERER ORDNUNG — MAP

Funktionen können als Argumente von Funktionen auftreten. Wir lernen drei Basics kennen:

FUNKTIONEN HÖHERER ORDNUNG — MAP

Funktionen können als Argumente von Funktionen auftreten. Wir lernen drei Basics kennen:

Die Funktion map

- `map` ermöglicht es eine Funktion `f` auf alle Elemente einer Liste anzuwenden

```
map :: (Int -> Int) -> [Int] -> [Int]
map f [] = []
map f (x:xs) = f x : map f xs
```

- *Beispiel.*

```
map square [1,2,7,12,3,20] = [1,4,49,144,9,400]
```

FUNKTIONEN HÖHERER ORDNUNG — FILTER

Die Funktion filter

- `filter p xs` liefert eine Liste, die genau die Elemente von `xs` enthält, welche das Prädikat `p` erfüllen

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 filter p xs = [ x | x <- xs, p x]
```

- *Beispiel.*

```
1 filter odd [1,2,7,12,3,20] = [1,7,3]
```

FUNKTIONEN HÖHERER ORDNUNG — FOLDR

Die Funktion foldr

- `foldr f z xs` faltet eine Liste `xs` und verknüpft jeweils durch die Funktion `f`; gestartet wird mit `z` und dem rechtesten Element

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f z []     = z
3 foldr f z (x:xs) = f x (foldr f z xs)
```

- *Beispiel.*

```
1 foldr (+) 3 [1,2,3,4,5] = 18
2 length xs = foldr (+) 0 (map (\x -> 1) xs)
```

FUNKTIONEN HÖHERER ORDNUNG – ÜBERSICHT

- `map` wendet Funktion auf alle Listenelemente an

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

- `filter` wählt Listenelemente anhand einer Funktion aus

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x]
```

- `foldr` faltet eine Liste mit Verknüpfungsfunktion (von rechts beginnend)

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```haskell
f :: [Int] -> Int
```

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```haskell
f :: [Int] -> Int
```

```haskell
f xs
  = foldr (+) 0 (map (^2) (filter (`mod` 2) == 0) xs))
```

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```haskell
f :: [Int] -> Int
```

```haskell
f xs
  = foldr (+) 0 (map (^2) (filter (`mod` 2) == 0) xs))
```

```haskell
f' xs = foldr (*) 1 (map (^2) (filter even xs))
```

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```haskell
f :: [Int] -> Int
```

```haskell
f xs
  = foldr (+) 0 (map (^2) (filter (`mod` 2) == 0) xs))
```

```haskell
f' xs = foldr (*) 1 (map (^2) (filter even xs))
```

```haskell
f'' = foldr (*) 1 . map (^2) . filter even
```

```haskell
f'''
  = foldr (*) 1 . map (^2) . filter ((== 0) . (`mod`
      2))
```

AUFGABE 3

Faltung einer Liste von *links*

```
foldleft ::  (Int -> Int -> Int) -> Int -> [Int] ->
    Int
```

Faltung einer Liste von *links*

```
foldleft ::  (Int -> Int -> Int) -> Int -> [Int] ->
    Int
```

```
foldleft :: (Int -> Int -> Int) -> Int -> [Int] -> Int
foldleft f x []     = x
foldleft f x (y:ys) = foldleft f (f x y) ys
```

ENDE

Fragen?