ProgrammierParadigmen

Tobias Kahlert

Musterlösung

- Es gibt eine Musterlösung
- Ausgabe immer eine Woche nach Abgabe

Datum	Thema	Unterlagen
11.10.2016	Erste Schritte mit Haskell	[Übungsblatt 0]
20.10.2016	Rekursive Funktionen und Listen	[Übungsblatt 1][Zusatzblatt 1] [Beispiellösung Blatt 1][Beispiellösung Zusatzblatt 1]
27.10.2016	Bindung, Kombinatoren, Pattern	[Übungsblatt 2][Zusatzblatt 2]
3.11.2016	Laziness, Streams	[Übungsblatt 3][Zusatzblatt 3]

http://pp.info.uni-karlsruhe.de/lehre/WS201718/paradigmen/uebung/

Blatt 1

- Vergesst das Modul am Anfang nicht
- Testet Randbedingungen ab (aber nicht mit if!)!
- Benutzt möglich wenig if … then … else …
- Pattern Matching
- Guards
- Benutzt mehr where oder let … in
- Benutzt mehr Hilfsfunktionen/variablen
- Benutzt `` für binäre Operationen (Beispiel gleich)

where

```
count y xs = countAkk xs 0
    where
        countAkk [] acc = acc
        countAkk (x:xs) acc = countAkk xs (acc + f)
            where f = if y == x then 1 else 0
```

where

```
count y xs = countAkk xs 0
    where
        countAkk [] acc = acc
        countAkk (x:xs) acc = countAkk xs (acc + f)
            where f = if y == x then 1 else 0
```

where

```
count y xs = countAkk xs 0
    where
        countAkk [] acc = acc
        countAkk (x:xs) acc = countAkk xs (acc + f)
            where f = if y == x then 1 else 0
```

Binäre Operatoren

```
mul a b
  | a < 0 = error „a < 0“
  | b < 0 = error „b < 0“
  | otherwise = a * b

> mul 4 6

> 4 `mul` 6
```

Aufgabe 1

```
f y = \z -> x + 7 * z - y
x = 1
g x = x + (let y = x * 2; x = 5 * 5
                    in (let x = f x 2 in x + y))
h = let z = 2 in g x + (\z -> -z) z
    where z = 3
```

Tupel Dekonstruktion

```
returnTuple :: a -> (Integer, Integer)

func a = fst (returnTuple a) + snd (returnTuple + a)
```

Tupel Dekonstruktion

```
returnTuple :: a -> (Integer, Integer)

func a = fst (returnTuple a) + snd (returnTuple + a)

func a = fst tuple + snd tuple
    where
        tuple = returnTuple a
```

Tupel Dekonstruktion

```
returnTuple :: a -> (Integer, Integer)

func a = fst (returnTuple a) + snd (returnTuple + a)

func a = fst tuple + snd tuple
    where
        tuple = returnTuple a

func a = x + y
    where
        (x, y) = returnTuple a
```

Boolsche Ausdrücke

```
if X then True else False

if X then False else True

if X then Y else False

func
 | X = Y
 | otherwise = False
```

Boolsche Ausdrücke

```
if X then True else False                    X

if X then False else True

if X then Y else False

func
 | X = Y
 | otherwise = False
```

Boolsche Ausdrücke

```
if X then True else False                    X

if X then False else True                    not X

if X then Y else False

func
 | X = Y
 | otherwise = False
```

Boolsche Ausdrücke

```
if X then True else False          X

if X then False else True          not X

if X then Y else False             X && Y

func
 | X = Y
 | otherwise = False
```

Boolsche Ausdrücke

`if X then True else False`	`X`
`if X then False else True`	`not X`
`if X then Y else False`	`X && Y`

`func`
`| X = Y`
`| otherwise = False`
	`X && Y`

zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith (+) [1, 2, 3] [1, 2, 3]
=> [2, 4, 6]
```

zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith (+) [1, 2, 3] [1, 2, 3]
⇒ [2, 4, 6]

zipWith (+) [1, 2, 3] [1, 2, 3, 4]
=> ???
```

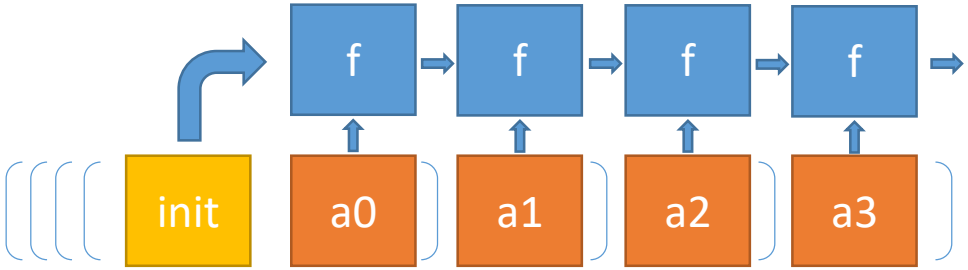
zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith (+) [1, 2, 3] [1, 2, 3]
⇒ [2, 4, 6]

zipWith (+) [1, 2, 3] [1, 2, 3, 4]
⇒ [2, 4, 6]

zipWith (+) [1, 2, 3] [1..]
=> [2, 4, 6]
```

foldl & foldr

foldl :: (a -> b -> a) -> a -> [b] -> a

f init list

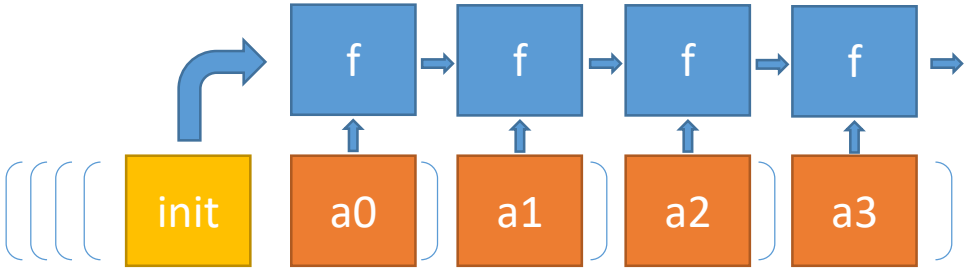


foldl 0 (+) [1, 2, 3]
-> (((0 + 1) + 2) + 3)
-> 6

foldl & foldr

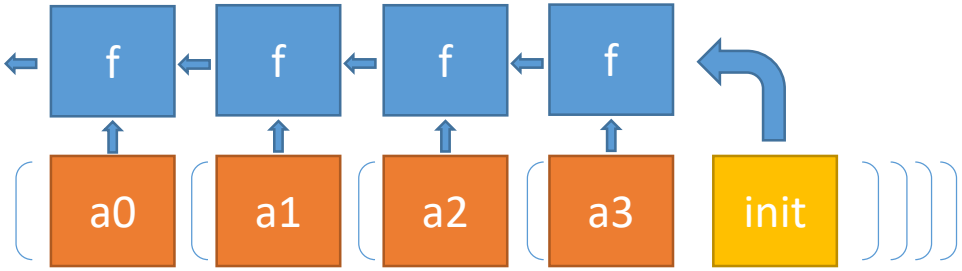
foldl :: (a -> b -> a) -> a -> [b] -> a

f init list



foldl 0 (+) [1, 2, 3]
-> (((0 + 1) + 2) + 3)
-> 6

foldr :: (b -> a -> a) -> a -> [b] -> a



foldr 0 (+) [1, 2, 3]
-> (1 + (2 + (3 + 0)))
-> 6

⇔

```
foldr (+) 0 X
⇔ sum X

sum $ map f X
  where
     f a = if a <= 7 then 1 else 0

⇔ length $ filter (<=7) X
```

Blatt 2

- Bindungen
- Listenkombinatoren
- Hisch-Index
- (Lauflängenkodierung)

Eigene Operatoren/Precedences

Prec-edence	Left associative operators	Non-associative operators	Right associative operators
9	!!		.
8			^, ^^, **
7	*, /, `div`,		
	`mod`, `rem`, `quot`		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=,	
		`elem`, `notElem`	
3			&&
2			||
1	>>, >>=		
0			$, $!, `seq`

https://www.haskell.org/onlinereport/decls.html#sect4.4.2

Haskell

https://xkcd.com/1312/