

# Tutorium 02: Kombinatoren, Lazyness

---

David Kaufmann

09. November 2022

Tutorium Programmierparadigmen am KIT

## Verwirrung vom letzten Mal

Wir waren uns nicht sicher was wie stark bindet

```
module Length where
lengthL l = lengthAcc l 0 where
  lengthAcc [] a = a
  lengthAcc (h:t) a = lengthAcc t a + 1
```

`lengthAcc [1,2] 0`

`=> (lengthAcc [2] 0) + 1`

`=> ((lengthAcc [] 0) + 1) + 1`

- -
- Richtig, kleine Fehler
- Aufgabe nicht verstanden
- Grundansatz falsch
- Richtig!
- Richtiger Ansatz, aber unvollständig

# Übungsblatt 1

---

# Stabile Sortialgorithmen

Wir sortieren diese Liste anhand dem ersten Element der Tupel

- $[(1,1), (4,2), (2,1), (5,1), (4,1)]$

Das ist stabil

- $[(1,1), (2,1), (4,2), (4,1), (5,1)]$

Das nicht

- $[(1,1), (2,1), (4,1), (4,2), (5,1)]$

↪ Elemente mit dem gleichen Key müssen ihre Reihenfolge beibehalten

# **Vorlesungswiederholung**

---

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
  - Bsp.: `foldr f s [1,2,3]` berechnet `(f 1 (f 2 (f 3 s)))`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
  - Bsp.: `foldl f s [1,2,3]` berechnet `(f (f (f s 1) 2) 3)`
- Für beide gilt: `foldr operation initial list`

Weitere Kombinatoren:

- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `zip :: [a] -> [b] -> [(a, b)]`
- `and, or :: [Bool] -> Bool`

Idee: Statt Rekursion selbst zu formulieren verwenden wir fertige „Bausteine“, sogenannte „Kombinatoren“.



Schreibt ein Modul Tut02 mit:

- `import Prelude hiding (foldl, foldr, map, filter, scanl, zip, zipWith)`
- `map` — Einmal von Hand, einmal per Fold
- `filter` — Einmal von Hand, einmal per Fold
- `squares 1` — Liste der Quadrate der Elemente von `1`
- `zip as bs` — Erstellt Tupel der Elemente von `as` und `bs`
- `zipWith as bs` — Wendet komponentenweise `f` auf die Elemente von `as` und `bs` an
  - Bspw. `zipWith (+) [1, 1, 2, 3] [1, 2, 3, 5] == [2, 3, 5, 8]`

# Aufgaben

Schreibt ein Modul Tut02 mit:

- `import Prelude hiding (foldl, foldr, map, filter, scanl, zip, zipWith)`
- `map` — Einmal von Hand, einmal per Fold
- `filter` — Einmal von Hand, einmal per Fold
- `squares 1` — Liste der Quadrate der Elemente von 1
- `zip as bs` — Erstellt Tupel der Elemente von as und bs
- `zipWith as bs` — Wendet komponentenweise f auf die Elemente von as und bs an
  - Bspw. `zipWith (+) [1, 1, 2, 3] [1, 2, 3, 5] == [2, 3, 5, 8]`
- `foldl`
- `scanl f i 1` — Wie `foldl`, gibt aber eine Liste aller Akkumulatorwerte zurück
  - Bspw. `scanl (*) 1 [1, 3, 5] == [1, 3, 15]`

# Lazy Evaluation

```
$ ghci
GHCi, version 8.8.4: http://www.haskell.org/ghc/
Prelude> x = 42 `div` 0
Prelude> putStrLn $ show x
*** Exception: divide by zero
```

- Was heißt Lazy Evaluation?
- Wieso tritt erst bei der zweiten Eingabe ein Fehler auf?

```
$ ghci
GHCi, version 8.8.4: http://www.haskell.org/ghc/
Prelude> x = 42 `div` 0
Prelude> putStrLn $ show x
*** Exception: divide by zero
```

- Was heißt Lazy Evaluation?
- Wieso tritt erst bei der zweiten Eingabe ein Fehler auf?
- $\leadsto$  Berechnungen finden erst statt, wenn es *absolut* nötig ist

# Lazy Evaluation

[wiki.haskell.org/Lazy\\_Evaluation](http://wiki.haskell.org/Lazy_Evaluation):

*Lazy evaluation means that expressions are not evaluated when they are bound to variables, but their evaluation is **deferred** until their results are needed by other computations.*

- Auch: *call-by-name* im Gegensatz zu *call-by-value* in bspw. C
- Was bringt das?

# Lazy Evaluation

[wiki.haskell.org/Lazy\\_Evaluation](http://wiki.haskell.org/Lazy_Evaluation):

*Lazy evaluation means that expressions are not evaluated when they are bound to variables, but their evaluation is **deferred** until their results are needed by other computations.*

- Auch: *call-by-name* im Gegensatz zu *call-by-value* in bspw. C
- Was bringt das?
- Ermöglicht bspw. arbeiten mit unendlichen Listen
- Berechnungen, die nicht gebraucht werden, werden nicht ausgeführt

# Lazy Evaluation

- Arithmetische Operatoren ((+), (-), (\*)) werten Argumente **immer** aus
- Vergleichsoperatoren (<), (<=), (==) werten Argumente **immer** aus
- Boolesche Operatoren (( ), (||)) nutzen Short-Circuit-Auswertung
- Strukturelle Operatoren ((:), (,)) werten Argumente **nie** aus

# List Comprehension

Syntax:  $[e \mid q_1, \dots, q_m]$

- $e$  ist ein Ausdruck
- $q_1, \dots, q_m$  sind Generatoren ( $p \leftarrow xs$ ) oder Prädikate



# List Comprehension

Syntax:  $[e \mid q_1, \dots, q_m]$

- $e$  ist ein Ausdruck
- $q_1, \dots, q_m$  sind Generatoren ( $p \leftarrow xs$ ) oder Prädikate

Die Reihenfolge ist wichtig!

- $[(x,y) \mid x \leftarrow [1..10], y \leftarrow [1..x]]$  Funktioniert
- $[(x,y) \mid y \leftarrow [1..x], x \leftarrow [1..10]]$  Kompilierfehler
- $[(x,y) \mid x \leftarrow [1..], y \leftarrow [1..]]$  Immer  $(1,n)$

Erstelle eine Liste, die alle Tupel  $(x,y)$  mit  $x,y \in \mathbb{N}_+$  genau einmal enthält.

Erstelle eine Liste, die alle Tupel  $(x,y)$  mit  $x,y \in \mathbb{N}_+$  genau einmal enthält.

```
[(x,y) | n <- [1..], x <- [1..n-1], y <- [n-x]]
```

- *List comprehension, Laziness*
- $[f\ x \mid x \leftarrow xs, p\ x] \equiv \text{map } f (\text{filter } p\ xs)$   
Bspw.:  $[x * x \mid x \leftarrow [1..]] \Rightarrow [1,4,9,16,25,\dots]$
- *Tupel*
- $(,) :: a \rightarrow b \rightarrow (a, b)$  („Tupel-Konstruktor“)
- $\text{fst} :: (a, b) \rightarrow a$
- $\text{snd} :: (a, b) \rightarrow b$