# Tutorium 02: Mehr Haskell

Paul Brinkmeier

08. November 2022

Tutorium Programmierparadigmen am KIT

# Notenskala

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

# Heutiges Programm

# Programm

- Übungsblatt 1
- Wiederholung der Vorlesung
- Aufgaben zu Haskell

# Übungsblatt 1

# Basisfälle und Spezialfälle

```
pow3 b e
    | e < 0 = error "Negativer Exponent"
    | otherwise = pow3Acc 1 b e

pow3Acc acc b e
    | e == 0 = acc
    | even e = pow3Acc acc (b * b) (e `div` 2)
    | odd e  = pow3Acc (b*acc) (b * b) (e `div` 2)
```

- Spezial- und Fehlerfälle in den „Rumpf“
- Basisfälle für Iteration in die Endrekursion

  $\leadsto$ Vermeidung von Codedopplung

# Endrekursion

```
   pow3 2 8                    while (e != 0) {
=> pow3Acc   1   2 8             b = b * b;
=> pow3Acc   1   4 4             e = e / 2;
=> pow3Acc   1  16 2             if (e % 2 == 1) {
=> pow3Acc   1 256 1               acc = b * acc;
=> pow3Acc 256   _ 0 .           }
=> 256                      }
```

```
pow3Acc acc b e
    | e == 0 = acc
    | even e = pow3Acc acc (b * b) (e `div` 2)
    | odd e  = pow3Acc (b*acc) (b * b) (e `div` 2)
```

# Sortierung

$$\texttt{sort}\,[5, 2_1, 2_2, 2_3, 1] = [1, 2_1, 2_2, 2_3, 5]\,\checkmark$$
$$= [1, 2_3, 2_1, 2_2, 5]\,\text{✗}$$

- Bei Aufgabe 2 war implizit gefordert, dass die Sortierfunktionen stabil sind.
- Originalreihenfolge soll nicht geändert werden.
- $\Longrightarrow$ Bei `merge` nicht tauschen, >= vs. >, ...

# Wiederholung: Funktionen

# Cheatsheet: Listenkombinatoren

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `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“.

# Cheatsheet: Tupel und Konzepte

- *List comprehension*, *Laziness*
- `[f x | x <- xs, p x]` ≡ `map f (filter p xs)`
  Bspw.: `[x * x | x <- [1..]]` ⇒ `[1,4,9,16,25,...]`
- *Tupel*
- `(,) :: a -> b -> (a, b)` („Tupel-Konstruktor“)
- `fst :: (a, b) -> a`
- `snd :: (a, b) -> b`

# Cheatsheet: Typen

- `Char`, `Int`, `Integer`, ...
- `String`
- *Typvariablen* `a`, `b`/*Polymorphe Typen*:
  - `(a, b)`: Tupel
  - `[a]`: Listen
  - `a -> b`: Funktionen
  - Vgl. Java: `List<A>`, `Function<A, B>`
- *Typsynonyme*: `type String = [Char]`

# 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 l` — Liste der Quadrate der Elemente von `l`
- `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 l` — Liste der Quadrate der Elemente von `l`
- `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 l` — Wie `foldl`, gibt aber eine Liste aller Akkumulatorwerte zurück
  - Bspw. `scanl (*) 1 [1, 3, 5] == [1, 3, 15]`

# Lazy Evaluation

# 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?

# 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?
- $\rightsquigarrow$ Berechnungen finden erst statt, wenn es *absolut* nötig ist

# Lazy Evaluation

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:

> *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

# Hangman

# Hangman

Schreibt folgende Funktionen:

- `showHangman` — Zeigt aktuellen Spielstand als `String`
  - Definition: `showHangman word guesses = ...`
  - Typ: `showHangman :: String -> [Char] -> String`
- `updateHangman` — Bildet Usereingabe (als `String`) und alten Zustand auf neuen Zustand ab
  - Definition: `updateHangman inputLine guesses`
  - Beispiel: `updateHangman "haske-= "haske"`

# Hangman — CLI-Framework

```
module CLI where


runConsoleGame ::
  (s -> String) ->
  (String -> s -> s) ->
  s ->
  IO ()
```

- `s` ist der Typ des Spielzustands
- Anfänglicher Zustand: `[]` — leere Liste an Rateversuchen
- `showHangman :: [Char] -> String`

# Hangman — CLI-Framework

```
module CLI where


runConsoleGame ::
  (s -> String) ->
  (String -> s -> s) ->
  s ->
  IO ()
```

- `s` ist der Typ des Spielzustands
- Anfänglicher Zustand: `[]` — leere Liste an Rateversuchen
- `showHangman :: [Char] -> String`
- `updateHangman :: String -> [Char] -> [Char]`

# Hangman — CLI-Framework

```
module CLI where


runConsoleGame ::
  (s -> String) ->
  (String -> s -> s) ->
  s ->
  IO ()
```

- `s` ist der Typ des Spielzustands
- Anfänglicher Zustand: `[]` — leere Liste an Rateversuchen
- `showHangman :: [Char] -> String`
- `updateHangman :: String -> [Char] -> [Char]`
- `initHangman :: [Char]`

# Hangman — Beispiele

- `showHangman "test" "e"` $\Rightarrow$ `". e . . | e"`
- `showHangman "test" "sf"` $\Rightarrow$ `". . s . | s f"`
- `updateHangman "f" "abc"` $\Rightarrow$ `"fabc"`

# Übungsblatt 2

# `where` **vs.** `let`

```
f = let y = 21 in y where y = 50
```

Zu was wertet `f` aus?

# `where` **vs.** `let`

```
f = let y = 21 in y where y = 50
```

Zu was wertet `f` aus?