

# Tutorium 02: Mehr Haskell

---

Paul Brinkmeier

28. Oktober 2019

Tutorium Programmierparadigmen am KIT

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

# Heutiges Programm

---

- Übungsblatt 1
- Wiederholung der Vorlesung
- Hangman in Haskell

# Übungsblatt 1

---

## 1.1 — pow1

```
module Arithmetik1 where
```

```
pow1 base exp
```

```
  | exp == 0    = 1
```

```
  | otherwise = base * (pow1 base (exp - 1))
```

```
module Arithmetik2 where

pow2 base exp
  | exp == 0 =
    1
  | exp `mod` 2 == 0 =
    pow2 (base * base) (exp `div` 2)
  | otherwise =
    base * (pow2 base (exp - 1))
```

## 1.3 — pow3

```
module Arithmetik3 where

pow3 base exp = pow3Acc base exp 1
  where
    pow3Acc base exp acc
      | exp == 0 =
        acc
      | exp `mod` 2 == 0 =
        pow3Acc (base * base) (exp `div` 2) acc
      | otherwise =
        pow3Acc base (exp - 1) (base * acc)
```



## 1.4 — root

```
module Arithmetik4 where

import Arithmetik3 (pow3)

root exp r
  | exp <= 0  = error "Exponent negativ"
  | r < 0     = error "Wurzel komplex"
  | otherwise = searchRoot 0 (r + 1)
where
  searchRoot lower upper
    | upper - lower == 1 = lower
    | r < avg 'pow3' exp = searchRoot lower avg
    | otherwise          = searchRoot avg upper
  where
    avg = (lower + upper) 'div' 2
```

## 1.5 — isPrime

```
module Arithmetik5 where

import Arithmetik4 (root)

isPrime n = not (any (divides n) [2..root 2 n])
  where
    divides p q = p `mod` q == 0
```

## 2 — insert, insertSort

```
module Sort1 where

insert x [] = [x]
insert x (s : sp)
  | s > x    = x : s : sp
  | otherwise = s : insert x sp

insertSort []      = []
insertSort (s : sp) = insert s (insertSort sp)
```

### 3 — merge, mergeSort

```
module Sort2 where

merge listA [] = listA
merge [] listB = listB
merge (a : as) (b : bs)
  | a < b      = (a : merge as (b : bs))
  | otherwise = (b : merge (a : as) bs)

mergeSort []    = []
mergeSort [a]   = [a]
mergeSort list = merge (mergeSort a) (mergeSort b)
  where
    a = take (length list `div` 2) list
    b = drop (length list `div` 2) list
```

## **Wiederholung: Eingebaute Funktionen**

---

- Für `[ a ]`:
  - `map :: (a -> b) -> [ a ] -> [ b ]`
  - `filter :: (a -> Bool) -> [ a ] -> [ a ]`
  - `all :: (a -> Bool) -> [ a ] -> Bool`
  - `any :: (a -> Bool) -> [ a ] -> Bool`
  - `foldl :: (b -> a -> b) -> b -> [ a ] -> b`
- Für Funktionen:
  - `(.) :: (a -> b) -> (b -> c) -> (a -> c)`
  - `($) :: (a -> b) -> a -> b`
  - `flip :: (a -> b -> c) -> (b -> a -> c)`

Schreibt ein Modul Tut02 mit:

- `import Prelude ()` — Verhindert Laden der Standardbibliothek
- `map`
- `filter`
- `squares 1` — Liste der Quadrate der Elemente von 1
- `odd`, `even` — Prüft ob eine Zahl (un-)gerade ist
- `odds`, `evens` — Liste aller (un-)geraden Zahlen  $\geq 0$
- `foldl`
- `scanl f 1` — 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 8.6.5: 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 8.6.5: 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](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 arbeiten mit unendlichen Listen
- Berechnungen, die nicht gebraucht werden, werden nicht ausgeführt

# Hangman

---

- [//pbrinkmeier.de/Hangman.hs](http://pbrinkmeier.de/Hangman.hs)
- `showHangman` — Zeigt aktuellen Spielstand als `String`
- `updateHangman` — Bildet Usereingabe (als `String`) und alten Zustand auf neuen Zustand ab
- `initHangman` — Anfangszustand, leere Liste

```
module CLI where

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

- s ist der Typ des Spielzustands
- Anfänglicher Zustand: [] — leere Liste an Rateversuchen
- Parameter 1: showHangman
- Parameter 2: updateHangman
- Parameter 3: initHangman

## Hangman — Beispiele

- `showHangman "Test"['e'] == ". e . . | e"`
- `showHangman "Test"['s', 'f'] == ". . s . | s f"`
- `updateHangman "f"['a'] == ['f', 'a']`