

# Tutorium 04: Entwurf in Haskell

---

Paul Brinkmeier

12. November 2019

Tutorium Programmierparadigmen am KIT

# Heutiges Programm

---

- Übungsblatt 3
- Software-Entwurf in Haskell

# Übungsblatt 3

---

```
module Fibs where
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- Schön kompakt
- `zipWith` ist endrekursiv  $\Rightarrow$  linearer Speicher
- `fibs !! n`  $\in O(n)$

```
module Fibs where
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- Schön kompakt
- `zipWith` ist endrekursiv  $\Rightarrow$  linearer Speicher
- `fibs !! n`  $\in O(n)$ 
  - (wenn Addition konstant)

## 2 — Collatz-Vermutung

```
module Collatz where

collatz = iterate next
  where next aN | aN `mod` 2 == 0 = aN `div` 2
               | otherwise      = 3 * aN + 1

num = length . takeWhile (/= 1) . collatz

maxNum a b = bestNum [(m, num m) | m <- [a..b]]
  where bestNum = foldl maxSecond (0, 0)
        maxSecond (a, b) (x, y)
          | b >= y    = (a, b)
          | otherwise = (x, y)
```

## 2 — Collatz-Vermutung

```
module CollatzAlt where

import Collatz (num)
import Data.Function (on)
import Data.List (maximumBy)

maxNum a b =
  maximumBy
    (compare 'on' snd)
    [(m, num m) | m <- [a..b]]
```

- „eleganter“
- In der Klausur aber eher nur Funktionen aus der Prelude verwenden



### 3 — Stream-Kombinatoren

```
module Merge where

import Sort (merge)
import Primes (primes)

primepowers n = mergeAll $ map primesexp [1..n]
  where mergeAll = foldl merge []
        primesexp i = map (^i) primes
```

- Für  $i$  in  $1..n$  unendliche Liste der Primzahlen hoch  $i$  erstellen
- Wegen Laziness: wird nur so weit ausgewertet wie nötig
- Dann: Alle miteinander vereinigen

```
*Merge> pp3 = primepowers 3
*Merge> take 10 pp3
[2,3,4,5,7,8,9,11,13,17]
*Merge> :sprint pp3
pp3 = 2 : 3 : 4 : 5 : 7 : 8 : 9 : 11 : 13 : 17 : _
```

- :sprint a gibt aktuelle Speicherrepräsentation für a aus
- \_ steht dabei für „noch nicht ausgewertet“
- $\rightsquigarrow$  praktisch für Debugging unendlicher Listen

# **Wiederholung: Algebraische Datentypen**

---

# Algebraische Datentypen

```
module DataExamples where

data Bool = True | False

data Category = Jackets | Pants | Shoes
  deriving Show

data Filter
  = InSale
  | IsCategory Category
  | PriceRange Float Float
```

- Keyword `data` definiert *neuen* Typ
- „enum auf Meth“
- Ersetzt oft Vererbung im Entwurfsprozess

# Software-Entwurf in Haskell

---

- Statt abstrakter Klasse + Unterklassen: ein Summentyp
  - OOP:

- Statt abstrakter Klasse + Unterklassen: ein Summentyp
  - OOP:
    - `abstract class Auto { ... }`
    - `class Verbrenner extends Auto { ... }`
    - `class Elektro extends Auto { ... }`
  - FP:

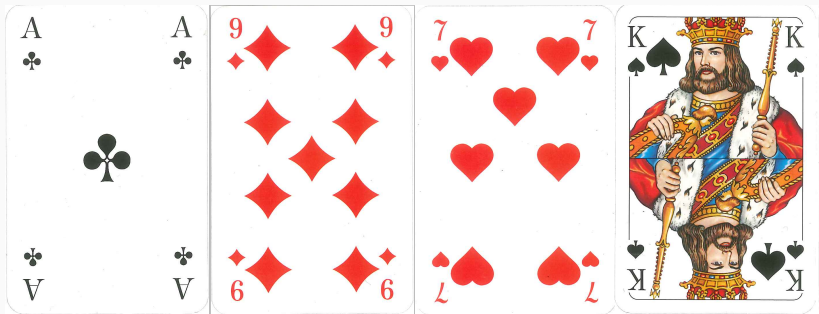
- Statt abstrakter Klasse + Unterklassen: ein Summentyp
  - OOP:
    - `abstract class Auto { ... }`
    - `class Verbrenner extends Auto { ... }`
    - `class Elektro extends Auto { ... }`
  - FP: `data Auto = Verbrenner ... | Elektro ...`
- Implementierung der verschiedenen Verhaltensweisen:
  - OOP:



- Statt abstrakter Klasse + Unterklassen: ein Summentyp
  - OOP:
    - `abstract class Auto { ... }`
    - `class Verbrenner extends Auto { ... }`
    - `class Elektro extends Auto { ... }`
  - FP: `data Auto = Verbrenner ... | Elektro ...`
- Implementierung der verschiedenen Verhaltensweisen:
  - OOP: `@Override void accelerate()` in Unterklassen
  - FP:

- Statt abstrakter Klasse + Unterklassen: ein Summentyp
  - OOP:
    - `abstract class Auto { ... }`
    - `class Verbrenner extends Auto { ... }`
    - `class Elektro extends Auto { ... }`
  - FP: `data Auto = Verbrenner ... | Elektro ...`
- Implementierung der verschiedenen Verhaltensweisen:
  - OOP: `@Override void accelerate()` in Unterklassen
  - FP: `accelerate :: Auto -> Auto`, Pattern-Matching auf Varianten

## Aufgaben: Spielkarten



```
module PlayingCards where
```

```
data Card = Card Rank Suit
```

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

```
data Rank = Seven | Eight | Nine | Ten
```

```
          | Jack | Queen | King | Ace
```

## Aufgaben: Spielkarten

```
module PlayingCards where

data Card = Card Rank Suit
data Suit = Spades | Hearts | Diamonds | Clubs
data Rank = Seven | Eight | Nine | Ten
           | Jack | Queen | King | Ace
```

- [//github.com/pbrinkmeier/pp-tut](https://github.com/pbrinkmeier/pp-tut)
- Implementiert Eq und Ord für Card, Suit und Rank:
  - instance Eq Suit where  
    Spades == Spades = True  
    ...
  - Tipp für Ord: Abbilden auf Int, dann vergleichen

## Aufgaben: Spielkarten

```
module PlayingCards where

data Card = Card Rank Suit
data Suit = Spades | Hearts | Diamonds | Clubs
data Rank = Seven | Eight | Nine | Ten
           | Jack | Queen | King | Ace
```

- [//github.com/pbrinkmeier/pp-tut](https://github.com/pbrinkmeier/pp-tut)
- Implementiert Eq und Ord für Card, Suit und Rank:
  - instance Eq Suit where  
    Spades == Spades = True  
    ...
  - Tipp für Ord: Abbilden auf Int, dann vergleichen
- Implementiert Show für Card, Suit und Rank:
  - show \$ Card Ace Spades == "ace of spades"
  - show \$ Card Nine Clubs == "9 of clubs"

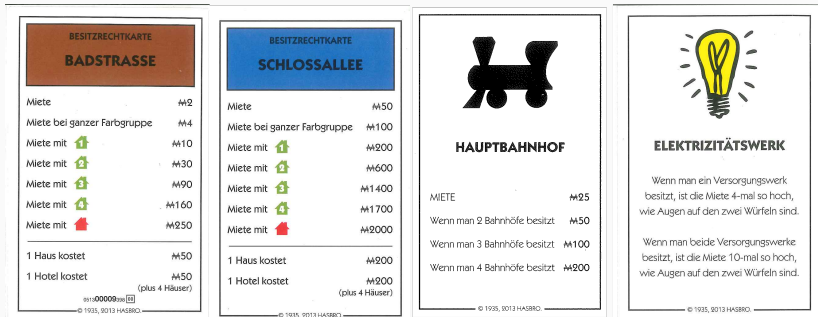
```
module PlayingCards where

data Card = Card Rank Suit deriving Eq
data Suit = Spades | Hearts | Diamonds | Clubs
           deriving (Eq, Ord)
data Rank = Seven | Eight | Nine | Ten
           | Jack | Queen | King | Ace
           deriving (Eq, Ord)

instance Ord Card where
    (Card r1 s1) <= (Card r2 s2) = r1 <= r2 || s1 <= s2
```

- Selbstimplementierung ist unnötige Schreiarbeit  $\rightsquigarrow$  kann i.d.R. vom Compiler übernommen werden
- deriving funktioniert nur für Eq, Ord, Enum, Enum, Bounded, Show und Read

# Aufgaben: Monopoly-Karten



module MonopolyCards where

data Card

```
= Street {- ... -}  
| Station {- ... -}  
| Utility {- ... -}
```

## Aufgaben: Monopoly-Karten

```
module MonopolyCards where

data Card
  = Street {- ... -}
  | Station {- ... -}
  | Utility {- ... -}
```

- Sagt dem Compiler, dass Card vergleich- und sortierbar ist
- Implementiert Show Card:
  - `show badStrasse == "Str (braun) Badstrasse: Haus = 50M"`
  - `show hauptbahnhof == "Sta Hauptbahnhof"`
  - `show electricCompany == "Utl Elektrizitätswerk"`



## Aufgaben: Monopoly-Karten

```
module MonopolyRent where

import MonopolyCards

getRent :: Card -> [Card] -> Int -> Int
getRent _ _ _ = 0
```

- Schreibt eine Funktion, die die Miete eines Felds ausrechnet
- Die Funktion nimmt:
  - Die Karte, auf der man gelandet ist
  - Die Liste der Karten des Besitzers (für bspw. Bahnhöfe)
  - Die Summe der Augen, die gewürfelt wurden (für die Werke)
- Die Funktion soll die Miete als Int zurückgeben