

Mehr funktionale Muster

Funktionale Programmierung

Prof. Dr. Oliver Braun

Letzte Änderung: 09.10.2018 06:54

- ähnlich zu **switch-case** in C-ähnlichen Sprachen
- aber allgegenwärtiges Feature von Haskell
- für **alle** Datenkonstruktoren möglich

Beispiele

```
isItTwo :: Integer → Bool
```

```
isItTwo 2 = True
```

```
isItTwo _ = False
```

```
data StudyCourse = IB | IC | IF
```

```
studyCourseName :: StudyCourse → String
```

```
studyCourseName IB = "Wirtschaftsinformatik"
```

```
studyCourseName IC = "Scientific Computing"
```

```
studyCourseName IF = "Informatik"
```

```
data StudyCourseSemester = IB Int | IC Int | IF Int
```

```
studyCourseName :: StudyCourseSemester → String
```

```
studyCourseName (IB i) = "Wirtschaftsinformatik, "  
                        ++ show i ++ ". Semester"
```

```
...
```

Pattern Matching Tuples

- statt

```
f :: (a, b) → (c, d) → ((b, d), (a, c))  
f x y = ((snd x, snd y), (fst x, fst y))
```

- mit Pattern Matching

```
f (a, b) (c, d) = ((b, d), (a, c))
```

- oder z.B.

```
fst3 :: (a, b, c) → a  
fst3 (x, _, _) = x  
third3 :: (a, b, c) → c  
third3 (_, _, x) = x
```

Pattern Matching im Ausdruck — **case**

- Pattern Matching ist nicht nur bei Parametern möglich
- Beispiel

```
data StudyCourse = IB | IC | IF
studyCourseName :: StudyCourse → String
studyCourseName course = "Bachelor "
  ++ case course of
    IB → "Wirtschaftsinformatik"
    IC → "Scientific Computing"
    IF → "Informatik"
```

- Fälle (im Beispiel: IB, IC und IF) müssen exakt untereinander ausgerichtet werden

- engl. *Higher Order Functions*
- Funktionen, die Funktionen als Parameter oder als Ergebnis haben
- Beispiel

```
flip :: (a → b → c) → b → a → c
```

```
Prelude> flip (-) 1 10
```

```
9
```

```
subtract5 = flip (-) 5
```

Nützliche HOFs für Listen

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
...
```

- Beispiele:

```
Prelude> map show [1..10]
["1","2","3","4","5","6","7","8","9","10"]
Prelude> filter odd [1..10]
[1,3,5,7,9]
Prelude> map (\x -> if x<3 then "zu klein"
                else show x) [1..5]
["zu klein","zu klein","3","4","5"]
```

- mehrere Ausdrücke für eine Funktionsdefinition
- erster Guard der `True` ergibt
- Beispiele

```
myAbs :: Integer → Integer
```

```
myAbs x
```

```
  | x < 0 = (-x)
```

```
  | otherwise = x
```

```
isRight :: (Num a, Eq a) ⇒ a → a → a → String
```

```
isRight a b c
```

```
  | a2 + b2 == c2 = "RIGHT ON"
```

```
  | otherwise = "not right"
```


- Funktionen können direkt kombiniert werden, analog zur Mathematik

$$(f \circ g)(x) \equiv f(g(x))$$

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- Beispiel:

```
negate . sum $ [1, 2, 3, 4, 5]
```

statt

```
negate (sum [1, 2, 3, 4, 5])
```

Punktfrei mit dem Punkt

- bisher haben wir Funktionen so definiert, dass wir das Ergebnis für einen **Punkt** angegeben haben, z.B.

```
f x = negate $ sum x
```

- stattdessen können wir auch schreiben

```
f = \x → (negate . sum) x
```

- Erinnerung: η -Konversion im λ -Kalkül: $\lambda x.f x \equiv f$
- damit können wir definieren:

```
f = negate . sum
```

- das ist **punktfrei**, weil der Punkt x nicht in der Funktionsgleichung auftaucht

- es gibt die beiden folgenden Funktionen um Werte auf der Konsole auszugeben

```
putStrLn :: String → IO ()  
print   :: Show a ⇒ a → IO ()
```

- `print` wandelt einen Wert zuerst mit der Funktion

```
show :: Show a ⇒ a → String
```

in einen `String` um

- `print` ist definiert durch

```
print = putStrLn . show
```

- Haskell hat keine Kontrollstrukturen wie Schleifen
 - da ohne Zustandsänderung natürlich nicht möglich!
- Mittel der Wahl in Haskell: **Rekursion**
- Rekursion hat üblicherweise
 - einen Rekursionsschritt und
 - einen Abbruchfall

- mathematisch definiert als

$$n! = \begin{cases} 1 & , \text{für } n = 0 \\ n * (n-1)! & , \text{sonst } (n > 0) \end{cases}$$

- in Haskell

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

- Datentyp `Maybe a` (wie `Optional` in Java oder `optional` in C++)

```
data Maybe a = Nothing | Just a
```

- Beispiel

```
safeDiv :: Integer → Integer → Maybe Integer
```

```
safeDiv x 0 = Nothing
```

```
safeDiv x y = Just $ x `div` y
```

```
Prelude> 5 `div` 0
```

```
*** Exception: divide by zero
```

```
Prelude> 5 `safeDiv` 0
```

```
Nothing
```

```
Prelude> 5 `safeDiv` 2
```

```
Just 2
```

Wenn **Nothing** nicht genug ist

- Datentyp `Either a b`

```
data Either a b = Left a | Right b
```

- `Right` (rechts, aber auch *richtig*) wird normalerweise für den Erfolg verwendet
- `Left` für den Fehlerfall

Beispiel mit **Either**

```
safeDiv :: Integer → Integer → Either String Integer
safeDiv x 0 = Left "divide by zero"
safeDiv x y = Right $ x `div` y
```

```
Prelude> 5 `safeDiv` 0
Left "divide by zero"
Prelude> 5 `safeDiv` 2
Right 2
```


Und wie geht es weiter im Fehlerfall?

- entweder mit Patter Matching

```
f ... = case ... of  
  Nothing ⇒ ...  
  Just ... ⇒ ...  
g ... = case ... of  
  Left ... ⇒ ...  
  Right ... ⇒ ...
```

- oder (besser) mit Funktionen aus `Data.Maybe` und `Data.Either`

```
maybe :: b → (a → b) → Maybe a → b  
either  :: (a → c) → (b → c) → Either a b → c
```

und viele mehr

- in die Doku schauen, lohnt sich!!!