VL06: Haskell (Funktionen höherer Ordnung, Currying)

IFM 5.3 Spezielle Methoden der Programmierung

Carsten Gips, FH Bielefeld

18.05.2015

Wiederholung

Wiederholung

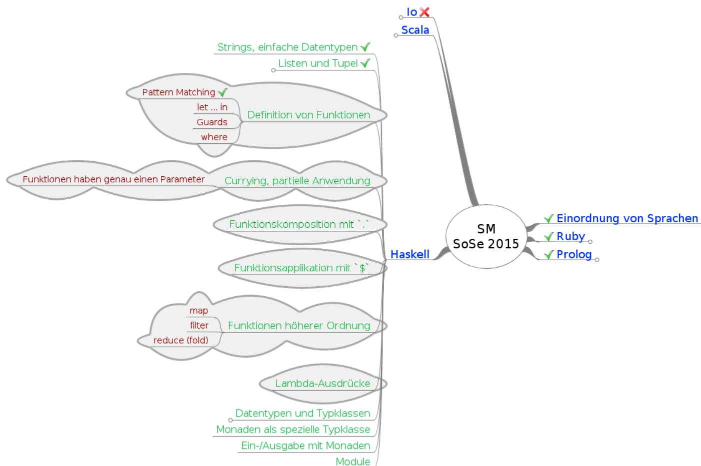
- Wie können Sie die ersten n Elemente einer Liste auswählen?
- Wie können Sie die ersten n Elemente einer Liste entfernen?
- Was sind “List Comprehensions”?
- Was ist der Unterschied zwischen den beiden folgenden Aufrufen?

```
zip [1 .. 5] ["one", "two", "three"]
[(x, y) | x <- [1 .. 5], y <- ["one", "two", "three"]]
```

- Was bedeutet “Pattern Matching” im Zusammenhang mit der Funktionsdefinition?

Motivation

Themen für heute



- Funktionen mit Guards und `where` definieren
- Currying und partiell angewendete Funktionen
- Funktionen höherer Ordnung: Faltungen, Map, Filter, Lambda-Ausdrücke

Weitere Formen der Funktionsdefinition

Guards, guards!

```haskell
bmi gewicht groesse
    | gewicht / groesse^2 <= 18.5    = "Spacko"
    | gewicht / groesse^2 <= 25.0    = "Normalo"
    | gewicht / groesse^2 <= 30.0    = "Untergross"
    | otherwise                      = "Autsch"
```

- Pattern Matching: Parameter passt zu Form (Destruktion)
- Guards: Lesbare Form eines `if`-Statements
  $\Rightarrow$ Wichtig: Letzter Fall immer mit `otherwise`

ghci :l bmi.hs

Verbesserung von `bmi`

Was stört Sie an dieser Version von `bmi`?

```
1 bmicalc gewicht groesse
2     | gewicht / groesse^2 <= 18.5 = "Spacko"
3     | gewicht / groesse^2 <= 25.0 = "Normalo"
4     | gewicht / groesse^2 <= 30.0 = "Untergross"
5     | otherwise                   = "Garfield"
```

Where!?

```
bmicalc gewicht groesse
    | bmi <= 18.5 = "Spacko"
    | bmi <= 25.0 = "Normalo"
    | bmi <= 30.0 = "Untergross"
    | otherwise   = "Garfield"
    where bmi = gewicht / groesse^2
```

⇒ Zusammenfassung der Berechnung in lokaler Variable

Where!? (cnt.)

```haskell
bmicalc gewicht groesse
    | bmi <= skinny = "Spacko"
    | bmi <= normal = "Normalo"
    | bmi <= fat    = "Untergross"
    | otherwise     = "Garfield"
    where bmi = gewicht / groesse^2
          skinny = 18.5
          normal = 25.0
          fat    = 30.0
```

⇒ Einrückung wichtig!

List Comprehensions (Listenverarbeitung)

$$S = \{2 * x \mid x \in N, x \leq 10\}$$

List Comprehensions (Listenverarbeitung)

$$S = \{2 * x \mid x \in N, x \leq 10\}$$

```
1 [x*2 | x <- [1..10]]
```

List Comprehensions (Listenverarbeitung)

$$S = \{2 * x \mid x \in N, x \leq 10\}$$

```
1  [x*2 | x <- [1..10]]
```

```
1  Prelude> [x*2 | x <- [1..10], x*2>5]
2
3  Prelude> let xs = ["A", "B", "C"]
4  Prelude> [a ++ "-" ++ b | a <- xs, b <- xs]
5  Prelude> [a ++ "-" ++ b | a <- xs, b <- xs, a < b]
6
7  Prelude> [(a,b) | a <- [1..3], b <- [1..a]]
```

ghci

Beispiel `zipWith`-Funktion

- Eingabe: Funktion f, zwei Listen
- Rückgabe: Liste
- Arbeitsweise: Fügt die Listen zusammen, indem für die korrespondierenden Elemente jeweils die Funktion f aufgerufen wird

Signatur?

Beispiel `zipWith`-Funktion

- Eingabe: Funktion f, zwei Listen
- Rückgabe: Liste
- Arbeitsweise: Fügt die Listen zusammen, indem für die korrespondierenden Elemente jeweils die Funktion f aufgerufen wird

Signatur?

```
1 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Definition?

Beispiel `zipWith`-Funktion

- Eingabe: Funktion f, zwei Listen
- Rückgabe: Liste
- Arbeitsweise: Fügt die Listen zusammen, indem für die korrespondierenden Elemente jeweils die Funktion f aufgerufen wird

Signatur?

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Definition?

```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

ghci

Berechnung der Aktivierung von Neuron a_j

Aktivierung von Neuron a_j:

$$a_j = \begin{cases} 1 & \text{falls} \quad w_{0,j} + a_1 w_{1,j} + a_2 w_{2,j} + ... + a_n w_{n,j} \geq 0 \\ 0 & \text{falls} \quad w_{0,j} + a_1 w_{1,j} + a_2 w_{2,j} + ... + a_n w_{n,j} < 0 \end{cases}$$

Zusammenfassung

Was haben Sie heute gehört?

- Funktionsdefinition mit Guards, mit Where-Bindings und Let-in
- Funktionen haben in Haskell genau **einen** Parameter
- **Currying**: Partielle Anwendung einer Funktion auf einen Parameter liefert Funktion mit restlichen Parametern
- Funktionskomposition mit `.`
- Funktionen höherer Ordnung: Funktionen als Parameter oder Rückgabe
  ⇒ Beispiele: `map`, `filter`, `foldl` (“reduce”)
- Anonyme Funktionen mit Lambda-Ausdrücken

Nächste Woche: Haskell (Typen und Typklassen)

Literatur zum Weiterlesen

- Miran Lipovaca: “Learn You a Haskell for Great Good!”, learnyouahaskell.com
- O’Sullivan, Stewart, Goerzen: “Real World Haskell”, book.realworldhaskell.org
- Bruce A. Tate: “Seven Languages in Seven Weeks”, Pragmatic Bookshelf Inc., 2010
  - Haskell: Kapitel 8
- Block, Neumann: “Haskell Intensivkurs”, Springer, 2011

Lernziele – Nach dieser Vorlesung sollten Sie ...

Verstehen (K2)
- Funktionen haben in Haskell einen Parameter
- Prinzip des Currying (schrittweise partielle Applikation)
- Signatur von Funktionen höherer Ordnung
- Currying oft lesbarer als Lambda-Ausdrücke

Anwenden (K3)
- Funktionsdefinition mit Guards und Pattern Matching
- Nutzung von Where-Bindings und Let-in
- Komposition von Funktionen mit `.`
- Umgang mit `map`, `filter`, `foldl`, `zip`, ...
- Nutzung von Lambda-Ausdrücken

Diese Fragen sollten Sie beantworten können ...

- Worin besteht der Unterschied zwischen Guards und Pattern Matching?
- Worin besteht der Unterschied zwischen Where-Bindings und Let-in-Ausdrücken?
- Erklären Sie Currying an einem Beispiel. Worin liegt die praktische Bedeutung von partieller Applikation?
- Was sind Lambda-Ausdrücke?

Diese Fragen sollten Sie beantworten können ...

- Erklären Sie `foldl` an einem Beispiel.
- Was bedeuten die folgenden Code-Schnipsel?

```
map (+ 1) [1, 2, 3]
filter odd [1, 2, 3, 4, 5]
foldl1 (+) 0 [1 .. 3]
```

Diese Fragen sollten Sie beantworten können ...

- Schreiben Sie im folgenden Code-Schnipsel `fibNth` mit Hilfe von Funktionskomposition um:

```
lazyFib x y = x:(lazyFib y (x + y))
fib = lazyFib 1 1
fibNth x = head (drop (x - 1) (take (x) fib))
```

- Definieren Sie eine Funktion `fib5th` durch partielle Applikation.