Programmierung und Modellierung mit Haskell

List-Comprehension und Pattern-Matching

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

10. April 2014

Inhalte letzte Vorlesung

- Funktionsbegriff $A \to B$
- Stelligkeit einer Funktion
- Infix-/Präfixnotation für Funktionsanwendung

```
(+) 1 2 `foo` 3
```

- Funktionen sind Werte in funktionalen Sprachen

- Basistypen `Bool, Int, Char, Double,...`
- Kartesische Produkte `(1,'a') :: (Int,Char)`
- Listen `['H','i','!'] :: [Char]`
- Typabkürzung mit `type` `type String = [Char]`

Listenkonstruktion

Erinnerung: Eine Liste ist eine geordnete Folge von Werten des gleichen Typs, mit beliebiger Länge, insbesondere auch Länge 0.

Möglichkeiten Listen zu konstruieren:

- Spezielle Syntax für vollständige Listen: `[1,2,3,4,5]`
- Aufzählungen `== [1..5]`
  - Man kann auch eine Schrittweite angeben:
    `[1,3..10] == [1,3,5,7,9]`
  - Die Haskell Syntax `[a,b..m]` ist eine Abkürzung für die Liste $[a, a+1(b-a), a+2(b-a), \ldots, a+n(b-a)]$, wobei n die größte natürliche Zahl mit $a+n(b-a) \leq m$ ist.
  - Funktioniert mit allen "aufzählbaren" Typen $\Rightarrow$ Typklassen
- List-Comprehension
- Infix Listenoperator `(:)`

Listenkonstruktion

Erinnerung: Eine Liste ist eine geordnete Folge von Werten des gleichen Typs, mit beliebiger Länge, insbesondere auch Länge 0.

Möglichkeiten Listen zu konstruieren:

- Spezielle Syntax für vollständige Listen: `[1,2,3,4,5]`
- Aufzählungen `== [1..5]`
  - Man kann auch eine Schrittweite angeben:
    `[1,3..10] == [1,3,5,7,9]`
  - Die Haskell Syntax `[a,b..m]` ist eine Abkürzung für die Liste $[a, a+1(b-a), a+2(b-a), \ldots, a+n(b-a)]$,
    wobei n die größte natürliche Zahl mit $a+n(b-a) \leq m$ ist.
  - Funktioniert mit allen "aufzählbaren" Typen $\Rightarrow$ Typklassen
- List-Comprehension
- Infix Listenoperator `(:)`

Listenkonstruktion

Erinnerung: Eine Liste ist eine geordnete Folge von Werten des gleichen Typs, mit beliebiger Länge, insbesondere auch Länge 0.

Möglichkeiten Listen zu konstruieren:

- Spezielle Syntax für vollständige Listen: `[1,2,3,4,5]`
- Aufzählungen `== [1..5]`
  - Man kann auch eine Schrittweite angeben:
    `[1,3..10] == [1,3,5,7,9]`
  - Die Haskell Syntax `[a,b..m]` ist eine Abkürzung für die Liste $[a, a+1(b-a), a+2(b-a), \ldots, a+n(b-a)]$,
    wobei n die größte natürliche Zahl mit $a+n(b-a) \leq m$ ist.
  - Funktioniert mit allen "aufzählbaren" Typen $\Rightarrow$ Typklassen
- List-Comprehension
- Infix Listenoperator `(:)`

Listenkonstruktion

Erinnerung: Eine Liste ist eine geordnete Folge von Werten des gleichen Typs, mit beliebiger Länge, insbesondere auch Länge 0.

Möglichkeiten Listen zu konstruieren:

- Spezielle Syntax für vollständige Listen: `[1,2,3,4,5]`
- Aufzählungen `== [1..5]`
  - Man kann auch eine Schrittweite angeben:
    `[1,3..10] == [1,3,5,7,9]`
  - Die Haskell Syntax `[a,b..m]` ist eine Abkürzung für die Liste $[a, a+1(b-a), a+2(b-a), \ldots, a+n(b-a)]$,
    wobei n die größte natürliche Zahl mit $a+n(b-a) \leq m$ ist.
  - Funktioniert mit allen "aufzählbaren" Typen $\Rightarrow$ Typklassen
- List-Comprehension
- Infix Listenoperator `(:)`

List-Comprehension

Mengen werden in der Mathematik oft intensional beschrieben:

$$\{ x^2 \mid x \in \{1, 2, \ldots, 10\} \text{ und } x \text{ ist ungerade} \} = \{1, 9, 25, 49, 81\}$$

wird gelesen als "Menge aller x^2, so dass gilt..."

Haskell bietet diese Notation ganz analog für Listen:

```
[ x^2 | x <- [1..10], odd x ]  ==  [1,9,25,49,81]
```

"Liste aller x^2,
wobei x aus der Liste $[1, \ldots, 10]$ gezogen wird und x ungerade ist"

Haskell hat auch eine Bibliothek für echte (ungeordnete) Mengen, aber Listen sind in Haskell grundlegender.

List-Comprehension

Mengen werden in der Mathematik oft intensional beschrieben:

$$\{ x^2 \mid x \in \{1, 2, \ldots, 10\} \text{ und } x \text{ ist ungerade} \} = \{1, 9, 25, 49, 81\}$$

wird gelesen als "Menge aller x^2, so dass gilt..."

Haskell bietet diese Notation ganz analog für Listen:

```
[ x^2 | x <- [1..10], odd x ]  ==  [1,9,25,49,81]
```

"Liste aller x^2,
wobei x aus der Liste $[1, \ldots, 10]$ gezogen wird und x ungerade ist"

Haskell hat auch eine Bibliothek für echte (ungeordnete) Mengen, aber Listen sind in Haskell grundlegender.

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

Abkürzung: `let` erlaubt Abkürzungen zur Wiederverwendung

```
> [ z | x <- [1..10], let z = x^2, z>50]
[64,81,100]
```

- Beliebig viele Generatoren, Filter und Abkürzungen
- Definition können "weiter rechts" verwendet werden
- Verschachtelung von List-Comprehensions erlaubt

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

Abkürzung: `let` erlaubt Abkürzungen zur Wiederverwendung

```
> [ z | x <- [1..10], let z = x^2, z>50]
[64,81,100]
```

- Beliebig viele Generatoren, Filter und Abkürzungen
- Definition können "weiter rechts" verwendet werden
- Verschachtelung von List-Comprehensions erlaubt

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

Abkürzung: `let` erlaubt Abkürzungen zur Wiederverwendung

```
> [ z | x <- [1..10], let z = x^2, z>50]
[64,81,100]
```

- Beliebig viele Generatoren, Filter und Abkürzungen
- Definition können "weiter rechts" verwendet werden
- Verschachtelung von List-Comprehensions erlaubt

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

Abkürzung: **`let`** erlaubt Abkürzungen zur Wiederverwendung

```
> [ z | x <- [1..10], let z = x^2, z>50]
[64,81,100]
```

- Beliebig viele Generatoren, Filter und Abkürzungen
- Definition können "weiter rechts" verwendet werden
- Verschachtelung von List-Comprehensions erlaubt

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

Abkürzung: `let` erlaubt Abkürzungen zur Wiederverwendung

```
> [ z | x <- [1..10], let z = x^2, z>50]
[64,81,100]
```

- Beliebig viele Generatoren, Filter und Abkürzungen
- Definition können "weiter rechts" verwendet werden
- Verschachtelung von List-Comprehensions erlaubt

Beispiele

Beliebig viele Generatoren, Filter und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden:

```
> [ (wert,name) | wert <-[1..3], name <- ['a'..'b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Die Reihenfolge der Generatoren bestimmt die Reihenfolge der Werte in der Ergebnisliste.

```
> [ (wert,name) | name <- ['a'..'b'], wert <-[1..3]]
[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]
```

Beispiele

Beliebig viele Generatoren, Filter und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden:

```
> [ (wert,name) | wert <-[1..3], name <- ['a'..'b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Die Reihenfolge der Generatoren bestimmt die Reihenfolge der Werte in der Ergebnisliste.

```
> [ (wert,name) | name <- ['a'..'b'], wert <-[1..3]]
[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]
```

Listenkonstruktor Cons

Jede nicht-leere Liste besteht aus einem **Kopf** und einem **Rumpf**.

engl.: **Head** and **Tail**

Einer gegeben Liste kann man mit dem Infixoperator `(:)` ein neuer Kopf gegeben werden, der vorherige Kopf wird zum zweiten Element der Liste:

```
> 0:[1,2,3]
[0,1,2,3]
```

```
> 'a':('b':['c'])
"abc"
```

Tatsächlich ist `[1,2,3]` nur andere Schreibweise für `1:2:3:[]`, beide Ausdrücke sind äquivalent. `(:)` ist rechtsassoziativ.

`(:)` konstruiert also einen neuen Listknoten, und wird deshalb oft auch **"Cons"-Operator** genannt

construct list node

Listenkonstruktor Cons

Jede nicht-leere Liste besteht aus einem **Kopf** und einem **Rumpf**.

engl.: **Head** and **Tail**

Einer gegeben Liste kann man mit dem Infixoperator `(:)` ein neuer Kopf gegeben werden, der vorherige Kopf wird zum zweiten Element der Liste:

```
> 0:[1,2,3]
[0,1,2,3]
```

```
> 'a':('b':['c'])
"abc"
```

Tatsächlich ist `[1,2,3]` nur andere Schreibweise für `1:2:3:[]`, beide Ausdrücke sind äquivalent. `(:)` ist rechtsassoziativ.

`(:)` konstruiert also einen neuen Listknoten, und wird deshalb oft auch **"Cons"-Operator** genannt

construct list node

Listenkonstruktor Cons

Jede nicht-leere Liste besteht aus einem **Kopf** und einem **Rumpf**.

engl.: **Head** and **Tail**

Einer gegeben Liste kann man mit dem Infixoperator `(:)` ein neuer Kopf gegeben werden, der vorherige Kopf wird zum zweiten Element der Liste:

```
> 0:[1,2,3]
[0,1,2,3]
```

```
> 'a':('b':['c'])
"abc"
```

Tatsächlich ist `[1,2,3]` nur andere Schreibweise für `1:2:3:[]`, beide Ausdrücke sind äquivalent. `(:)` ist rechtsassoziativ.

`(:)` konstruiert also einen neuen Listknoten, und wird deshalb oft auch **"Cons"-Operator** genannt

construct list node

Typvariablen

Welchen Typ hat `(:)`?

Typvariablen

Welchen Typ hat `(:)`?

```
> :t (:)
(:) :: a -> [a] -> [a]
```

Typvariablen

Welchen Typ hat `(:)`?

```
> :t (:)
(:) :: a -> [a] -> [a]
```

Typen werden in Haskell immer groß geschrieben. Also ist `a` kein Typ, sondern eine Typvariable. Typvariablen stehen für **einen** beliebigen Typen. Typvariablen werden immer klein geschrieben.

Typvariablen

Welchen Typ hat `(:)`?

```
> :t (:)
(:) :: a -> [a] -> [a]
```

Typen werden in Haskell immer groß geschrieben. Also ist `a` kein Typ, sondern eine Typvariable. Typvariablen stehen für **einen** beliebigen Typen. Typvariablen werden immer klein geschrieben.

Der Cons-Operator funktioniert also mit Listen beliebigen Typs:

```
> :t (:) 3                                  -- Präfix Notation
(:) 3 :: [Integer] -> [Integer]

> :t ('a' :)                                -- Infix  Notation
('a' :) :: [Char] -> [Char]
```

Typvariablen

Welchen Typ hat `(:)`?

```
> :t (:)
(:) :: a -> [a] -> [a]
```

Typen werden in Haskell immer groß geschrieben. Also ist `a` kein Typ, sondern eine Typvariable. Typvariablen stehen für **einen** beliebigen Typen. Typvariablen werden immer klein geschrieben.

Der Cons-Operator funktioniert also mit Listen beliebigen Typs:

```
> :t (:) 3                                  -- Präfix Notation
(:) 3 :: [Integer] -> [Integer]

> :t ('a' :)                                -- Infix  Notation
('a' :) :: [Char] -> [Char]
```

Der Ausdruck `'a':[1]` ergibt aber einen Typfehler, da verschiedene Typen für `a` gleichzeitig notwendig wären!

$\Rightarrow$ Polymorphie

Funktionstypen (Wdh.)

Der Typ einer Funktion ist ein zusammengesetzter Funktionstyp, der immer aus genau zwei Typen mit einem Pfeil dazwischen besteht. Jede Funktion hat genau ein Argument und ein Ergebnis.

Funktionstypen sind implizit rechtsgeklammert, d.h. man darf die Klammern manchmal weglassen:

`Int -> Int -> Int` wird gelesen als **`Int -> (Int -> Int)`**

Entsprechend ist die Funktionsanwendung implizit linksgeklammert:

`bar 1 8` wird gelesen als `(bar 1) 8`

Das bedeutet: `(bar 1)` ist eine Funktion des Typs **`Int -> Int`**!
Funktionen sind also normale Werte in einer funktionalen Sprache!

Funktionsdefinitionen (Wdh.)

Beispiel: Funktion `succ` bildet eine Zahl auf Ihren Nachfolger ab

```
succ :: Int -> Int
succ x = x + 1
```

Schema:

1. **Typsignatur** angeben — optional, aber empfehlenswert
2. Funktionsname, gefolgt von **Funktionsparametern**
3. nach einem `=` ein Ausdruck, der **Funktionsrumpf**

Bei Funktionsanwendung wird gemäß der definierenden Funktionsgleichung von links nach rechts ersetzt:

$$\texttt{succ 7} \quad \leadsto \quad \texttt{7 + 1} \quad \leadsto \quad \texttt{8}$$

Funktionsparameter werden vorher durch die entsprechenden **Argumente** der Funktionsanwendung ersetzt, bzw. **substituiert**

Konstanten

Die einfachste Definition ist eine Funktion ohne Argumente, also eine Konstante:

```
myName :: String
myName = "Steffen"


pi = 3.1415


squareNumbers :: [Int]
squareNumbers = [ x * x | x <- [1..9999] ]
```

Top-level Konstanten werden maximal einmal ausgewertet

Funktionsdefinitionen

- Funktionsrumpf ist immer ein Ausdruck (z.B. ein Wert), innerhalb dieses Ausdrucks dürfen verwendet werden:
  - Funktionsparameter
  - *alle* gültigen Top-Level Definitionen

    Reihenfolge der Definitionen innerhalb Datei ist unerheblich; Typdeklaration schreibt man üblicherweise zuerst

- Funktionsnamen müssen immer mit einem Kleinbuchstaben beginnen, danach folgt eine beliebige Anzahl an Zeichen:
  - Buchstaben, klein und groß
  - Zahlen
  - Apostroph `'`
  - Unterstrich `_`

  Beispiel: `thisIsAn_Odd_Fun'Name`

  Allerdings sind einige Schlüsselwörter als Bezeichner verboten: z.B. `type`, `if`, `then`, `else`, `let`, `in`, `where`, `case`,...

IF-THEN-ELSE

Ein wichtiges Konstrukt in vielen Programmiersprachen ist das **Konditional**; es erlaubt auf Bedingung `:: Bool` zu reagieren:

```
if <bedingung> then <das_eine> else <das_andere>
```

Zuerst muss die Bedingung zu `True` oder `False` ausgewertet werden, dann können wir den gesamten Ausdruck zu dem entsprechenden Zweig auswerten:

```
> if True then 43 else 69
43
> if False then 43 else 69
69
```

In einer funktionalen Sprache wie Haskell ist dies ein Ausdruck, kein Befehl! Der Else-Zweig ist nicht optional. If-then-else in Haskell entspricht also dem "? :"-Ausdruck in C oder Java.

Beispiele

Damit könnten wir schon interessante Funktionen definieren:

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

```
signum :: Int -> Int
signum n = if n < 0
           then -1
           else
             if n == 0
               then 0
               else 1
```

Beispiele

Damit könnten wir schon interessante Funktionen definieren:

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

```
signum :: Int -> Int
signum n = if n < 0
           then -1
           else
             if n == 0
               then 0
               else 1
```

Lokale Definitionen

Ein weiterer wichtiger zusammengesetzter Ausdruck ist die **lokale Definition**:

```
betragSumme :: Int -> Int -> Int
betragSumme x y =
  let x_abs = abs x in
  let y_abs = abs y in
  x_abs  +  y_abs
```

- Frische Bezeichner `x_abs` und `y_abs` nur innerhalb des Let-Ausdrucks benutzbar; werten zu jeweiligen Definition aus
- Lokale Definitionen werden höchstens einmal ausgewertet, auch wenn diese mehrfach im Ausdruck verwendet werden
- Lokale Definition kann auch Funktionen definieren

Layout

Einrückung gemäß Layout-Regel spart Tipparbeit und erhöht die Lesbarkeit: ⇒ Whitespace sensitiv!

```
betragSumme :: Int -> Int -> Int
betragSumme x y =
  let x_abs = abs x
      y_abs = abs y
  in x_abs + y_abs
```

Mehrere lokale Definitionen benötigen nur einen `let`-Ausdruck:

- **Spalte weiter rechts:** vorheriger Zeile geht weiter
- **gleicher Spalte:** nächste lokale Definition
- **Spalte weiter links:** Definition beendet

Erstes Zeichen nach `let` legt die Spalte fest

Vorteil: alle Definitionen dürfen sich dann gegenseitig verwenden

Where-Klausel

Mathematiker schreiben nachrangige lokale Definition gerne hinten dran. Haskell erlaubt dies auch:

```
betragSumme :: Int -> Int -> Int
betragSumme x y = x_abs + y_abs
  where
    x_abs = abs x
    y_abs = abs y
```

- Auch hier ist wieder die *Layout-Regel* zu beachten
- `where` kann alles, was ein `let`-Ausdruck auch kann
- `where` ist kein Ausdruck, sondern eine spezielle Syntax für Top-Level Definitionen

Pattern-Matching

Musterabgleich bzw. **Pattern-Matching** ist eine elegante Möglichkeit, Funktionen abschnittsweise zu definieren:

```
count :: Int -> String
count 0 = "Null"
count 1 = "Eins"
count 2 = "Zwei"
count x = "Viele"
```

- Anstatt einer Variablen geben wir auf der linken Seite der Funktionsdefinition einfach einen Wert an.
- Wir können mehrere Definitionen für eine Funktion angeben. Treffen mehrere Muster zu, wird zum zuerst definierten Rumpf ausgewertet. Die Muster werden also **von oben nach unten** mit dem Argument verglichen.
- GHC warnt uns vor Definitionen mit unsinnigen Mustern.

Wildcards

Das Muster "Variable" besteht jeden Vergleich:

```
count' :: Int -> String
count' 0 = "Null"
count' x = "Viele"
```

Man kann das Muster _ verwenden, wenn der Wert egal ist.
Man erkennt so besser, welche Argumente verwendet werden:

```
findZero :: Int -> Int -> Int -> Int -> String
findZero 0 _ _ _  = "Erstes"
findZero _ 0 _ _  = "Zweites"
findZero _ _ 0 _  = "Drittes"
findZero _ _ _ 0  = "Viertes"
findZero _ _ _ _  = "Keines"
```

- Vermeidet auch hilfreiche Warnung "defined but not used"
- Benannte Wildcards, z.B. `_kosten` auch möglich

Tupel-Muster

Mit Patterns können wir auch Tupel auseinander nehmen:

```
type Vector = (Double,Double)

addVectors:: Vector -> Vector -> Vector
addVectors (x1,y1) (x2,y2) = (x1+x2, y1+y2)

addV5 :: Vector -> Vector
addV5 v = addVectors v (5,5)

fst3 :: (a,b,c) -> a
fst3 (x,_,_) = x

snd3 :: (a,b,c) -> b
snd3 (_,y,_) = y
```

Listen-Muster

Mit Patterns können wir auch Listen auseinander nehmen:

```
null        :: [a] -> Bool
null []             =  True
null (_head : _tail) =  False
```

Wir können die leere Liste matchen, eine nicht-leere Liste, oder auch Listen mit genau *n*-Elementen:

```
count        :: [a] -> String
count []       =  "Null"
count [_]      =  "Eins"
count [_,_]    =  "Zwei"
count [x,y,z]  =  "Drei"
count  _       =  "Viele"
```

Listen-Muster

Wir können natürlich anstatt Wildcards auch Variablen verwenden, wenn wir die Elemente der Liste verwenden möchten:

```
sum :: [Int] -> Int
sum []         = 0
sum [x]        = x
sum [x,y]      = x+y
sum (x:y:z:_) = x+y+z
```

```
> sum [1,2,3]
6
```

Unvollständige Muster

```
head :: [a] -> a
head (h:_) = h
```

```
tail :: [a] -> [a]
tail (_:t) = t
```

Achtung: Die Muster von `head` und `tail` sind unvollständig.
Ein Aufruf kann dann einen Ausnahmefehler verursachen:

```
> head []
*** Exception: Prelude.head: empty list
```

Das bedeutet, das `head` und `tail` partielle Funktionen definieren.
GHC warnt uns zurecht vor solch unvollständigen Mustern.
Wenn es sich nicht vermeiden läßt, dann solle man wenigstens die Funktion `error :: String -> a` verwenden:

```
myHead :: [a] -> a
myHead (h:_) = h
myHead []    = error "myHead of empty list undefined"
```

Muster verschachteln

Verschiedene Muster dürfen kombiniert und verschachtelt werden:

```
sumHeads :: [(Int,Int)] -> [(Int,Int)]
sumHeads ((x1,y1):(x2,y2):rest) = (x1+x2,y1+y2):rest

> sumHeads [(1,2),(3,4),(5,6)]
[(4,6),(5,6)]
```

Teile verschachtelter Muster können mit as-Patterns benannt werden. Hinter einem Bezeichner schreibt man ein @-Symbol vor einem eingeklammerten Untermuster:

```
firstLetter :: String -> String
firstLetter xs@(x:_) = xs ++ " begins with " ++ [x]

> firstLetter "Haskell"
"Haskell begins with H"
```

wobei Infixfunktion (++) zwei Listen miteinander verkettet

Muster verschachteln

Verschiedene Muster dürfen kombiniert und verschachtelt werden:

```
sumHeads :: [(Int,Int)] -> [(Int,Int)]
sumHeads ((x1,y1):(x2,y2):rest) = (x1+x2,y1+y2):rest

> sumHeads [(1,2),(3,4),(5,6)]
[(4,6),(5,6)]
```

Teile verschachtelter Muster können mit `as`-Patterns benannt werden. Hinter einem Bezeichner schreibt man ein `@`-Symbol vor einem eingeklammerten Untermuster:

```
firstLetter :: String -> String
firstLetter xs@(x:_) = xs ++ " begins with " ++ [x]

> firstLetter "Haskell"
"Haskell begins with H"
```

wobei Infixfunktion (++) zwei Listen miteinander verkettet

Pattern-Matching überall

Pattern-Matching ist nicht nur in Funktionsdefinition erlaubt, sondern an allen möglichen Stellen, z.B. auf linker Seite von

- Generatoren in List-Comprehensions:

  ```
  > [ x | (x,_,7) <- [(1,2,7),(2,4,6),(3,8,7),(4,0,7)]]
  [1,3,4]
  ```

  Wenn der Pattern-Match fehlschlägt gibt es kein Listenelement dafür; es gibt keine Fehlermeldung.

- anonymen Funktionen: `\(x,_) -> x`
- `let`-Definitionen: `let (_,y) =`...
- Definitionen in `where`-Klauseln

Dabei sollte man darauf achten, dass das Pattern-Matching nicht fehlschlagen kann. Tupel können z.B. immer erfolgreich matchen, aber Listen-Pattern können fehlschlagen.

Case Ausdruck

Es gibt auch die Möglichkeit, ein Musterabgleich innerhalb eines beliebigen Ausdrucks durchzuführen:

```
case <Ausdruck> of
  <Muster> -> <Ausdruck>
  <Muster> -> <Ausdruck>
  <Muster> -> <Ausdruck>
```

Auch hier gilt wieder die *Layout-Regel*: In der Spalte, in der das erste Muster nach dem Schlüsselwort `of` beginnt, müssen auch alle anderen Muster beginnen.
Ein Ergebnis-Ausdruck kann sich so über mehrere Zeilen erstrecken, so lange alles weit genug eingerückt wurde.
Ein terminierendes Schlüsselwort gibt es daher nicht.

Beispiel

Freunde anderer Sprachen würden vermutlich schreiben:

```
describeList :: [a] -> String
describeList xs =
  "The list is " ++ case xs of
    []  -> "empty."
    [x] -> "a singleton list."
    xs  -> "a longer list."
```

Ohne case-Ausdruck ist es aber vielleicht lesbarer:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ describe xs
  where
    describe []  = "empty."
    describe [x] = "a singleton list."
    describe xs  = "a longer list."
```

Beispiel

Freunde anderer Sprachen würden vermutlich schreiben:

```
describeList :: [a] -> String
describeList xs =
  "The list is " ++ case xs of
    []  -> "empty."
    [x] -> "a singleton list."
    xs  -> "a longer list."
```

Ohne `case`-Ausdruck ist es aber vielleicht lesbarer:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ describe xs
  where
    describe []  = "empty."
    describe [x] = "a singleton list."
    describe xs  = "a longer list."
```

Wächter

Ein Musterabgleich kann zusätzlich mit **Wächtern** oder **Guards** verfeinert werden. Wächter sind Bedingungen, also ein Ausdrücke des Typs `Bool`, welche Variablen des Patterns verwenden dürfen:

```
signum :: Int -> Int
signum n
  | n < 0     = -1
  | n > 0     =  1
  | otherwise =  0
```

- Es wird der erste Zweig gewählt, welcher zu `True` auswertet
- Schlagen alle Wächter fehl, wird das nächste Pattern geprüft
- `otherwise` ist *kein Schlüsselwort*, sondern eine Konstante der Standardbibliothek und gleich `True`

Beispiel

Vergleich zwischen den möglichen Notationen:

```
signum :: Int -> Int
signum n
  | n < 0     = -1
  | n > 0     = 1
  | otherwise = 0
```

```
signum :: Int -> Int
signum n = if n < 0
           then -1
           else
             if n > 0
               then 1
               else 0
```

Beispiel

Vergleich zwischen den möglichen Notationen:

```
signum :: Int -> Int
signum n  | n < 0     = -1
          | n > 0     =  1
          | otherwise =  0
```

```
signum :: Int -> Int
signum n  = if      n < 0 then -1
            else if n > 0 then  1
            else               0
```

Die Wächter-Notation ist mit etwas Übung sicherlich lesbarer...

Pattern Guards

Seit Haskell 2010 können mehrere durch Komma getrennte Wächter angegeben werden, wie bei List-Comprehensions:

```
welcome :: String -> String -> String
welcome vorname nachname
| vorname  == "Steffen",
  nachname == "Jost"
  = "Greetings, Dr Jost!"
| nachname == "Jost"
  = "Welcome!
| otherwise
  = "Go away!"
```

Alle Guards eines Zweiges müssen erfüllt werden.

Ähnlich wie bei List-Comprehensions ist auch noch der Rückpfeil `<-` für weitere Pattern-Matches innerhalb des Wächters erlaubt.

Pattern Guards

Seit Haskell 2010 können mehrere durch Komma getrennte Wächter angegeben werden, wie bei List-Comprehensions:

```
welcome :: String -> String -> String
welcome vorname nachname
| vorname  == "Steffen",
  nachname == "Jost"
  = "Greetings, Dr Jost!"
| nachname == "Jost"
  = "Welcome!
| otherwise
  = "Go away!"
```

Alle Guards eines Zweiges müssen erfüllt werden.

Ähnlich wie bei List-Comprehensions ist auch noch der Rückpfeil `<-` für weitere Pattern-Matches innerhalb des Wächters erlaubt.

Kommentare

Auch wenn Haskell generell gut lesbar ist, sollte man sein Programm immer sinnvoll kommentieren:

Einzeiliger Kommentar:

Haskell ignoriert bis zum Ende einer Zeile alles, was nach einem doppelten Minus kommt. Gut, für kurze Bemerkungen.

```
id :: String -> String -- Identity function,
id x = x               -- does nothing really.
```

Mehrzeiliger Kommentar:

Für größeren Text eignen sich mehrzeilige Kommentare. Diese beginnen mit `{-` und werden mit `-}` beendet.

```
{- We define some useful constants
   for high-precision computations here. -}
pi = 3.0
e  = 2.7
```

Zusammenfassung

Heute behandelte Haskell Ausdrücke:

- Cons-Constructor `(:) :: a -> [a] -> [a]`
- Listen Aufzählungen `[1,3..99]`
- List-Comprehensions `[x|x<-[1..9], even x]`
- Konditional `if .. then .. else ..`
- Lokale Definitionen `let .. in ..`
- Pattern Matching `case .. of ..`

. . . sowie verschiedene Notationen zur Funktionsdefinition:

Funktionsdefinition in Haskell

```
foo :: typ1 -> typ2 -> typ3 -> ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter z. B. Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> typ2 -> typ3 -> ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter (z. Bsp. Bool)
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> typ2 -> typ3 -> ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter (z. Bsp. Bool)
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter (z. Bsp. Bool)
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n = expr2
foo pat31 ... pat3n = expr3
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
    | grd311, ..., grd31k = expr31
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
    | grd311, ..., grd31k = expr31
  where idA = exprA
        idB = exprB
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

Beispiele

```
show_signed :: Integer -> String
show_signed 0                = " 0"
show_signed i | i>=0         = "+" ++ (show i)
              | otherwise    =        (show i)


printPercent :: Double -> String
printPercent  x  = lzero ++ (show p2) ++ "%"
 where
   p2 :: Double
   p2 = (fromIntegral (round' (1000.0*x))) / 10.0

   lzero = if p2 < 10.0 then "0" else ""

   round' :: Double -> Int    -- Fixing type avoids
   round' z = round z         -- avoids a warning here
```

Beispiele

```
show_signed :: Integer -> String
show_signed 0                = " 0"
show_signed i | i>=0         = "+" ++ (show i)
              | otherwise    =        (show i)


printPercent :: Double -> String
printPercent  x  = lzero ++ (show p2) ++ "%"
 where
   p2 :: Double
   p2 = (fromIntegral (round' (1000.0*x))) / 10.0

   lzero = if p2 < 10.0 then "0" else ""

   round' :: Double -> Int    -- Fixing type avoids
   round' z = round z         -- avoids a warning here
```

Beispiel – Alles zusammen

```
concatReplicate :: Int -> [a] -> [a]
concatReplicate _     [] = []
concatReplicate n (x:xs)
    | n <= 0    = []
    | otherwise = concatReplicateAux n x xs
  where
    concatReplicateAux 0 _ [] = []
    concatReplicateAux 0 _ (h:t)
      = concatReplicateAux  n h t
    concatReplicateAux c h t
      = h : concatReplicateAux (c-1) h t
```

Beispiel – Alles zusammen

```
concatReplicate :: Int -> [a] -> [a]
concatReplicate _     [] = []
concatReplicate n (x:xs)
    | n <= 0    = []
    | otherwise = concatReplicateAux n x xs
  where
    concatReplicateAux 0 _ [] = []
    concatReplicateAux 0 _ (h:t)
      = concatReplicateAux  n h t
    concatReplicateAux c h t
      = h : concatReplicateAux (c-1) h t
```

Rekursion siehe Vorlesungsteil 3

Beispiel – Alles zusammen

```
concatReplicate :: Int -> [a] -> [a]
concatReplicate _     [] = []
concatReplicate n (x:xs)
    | n <= 0    = []
    | otherwise = concatReplicateAux n x xs
  where
    concatReplicateAux 0 _ [] = []
    concatReplicateAux 0 _ (h:t)
      = concatReplicateAux  n h t
    concatReplicateAux c h t
      = h : concatReplicateAux (c-1) h t
```

Rekursion siehe Vorlesungsteil 3

Alternative Definition (siehe Vorlesungsteil 10):

```
concatReplicate n = concatMap (replicate n)
```

FP Haskell Center

Integrierte Entwicklungsumgebung (IDE): `www.fpcomplete.com`
Erlaubt keinen Zugriff auf Interpreter. Stattdessen:

```
module Main where

main :: IO ()
main = do
    print result1
    print result2
--------------------------------
-- Normaler funktionaler Code:

result1 :: Integer
result1 = 40 + 2

result2 =  'H':"allo!"
```

DO-Notation und IO wird in Vorlesung 13 behandelt

Ziel dieser Webseite: Haskell vermitteln
Ziel unserer Vorlesung: Funktionale Programmierung