Programmierung und Modellierung mit Haskell

Teil 2: Funktionsdefinition & Pattern-Matching

Steffen Jost

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

11. April 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 2: Funktionsdefinition & Pattern-Matching

1. Listen (Teil 2)
   - Ranges
   - List-Comprehension
   - Cons
2. Funktionsdefinitionen
   - Ausdrücke
   - Überschatten
   - Pattern-Matching
   - Guards
3. Zusammenfassung
   - Volle Funktionsdefinition
   - Beispiele

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]`
- Technische Hürden für Benutzung von GHC/GHCi

  Whitespace, Typklassen in Fehlermeldungen, etc.

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]`

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.

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

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:

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

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

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

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]`

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

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]
```

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
- Definitionen 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')]
```

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]
```

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.

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 Listenknoten, und wird deshalb oft auch **"Cons"-Operator** genannt

construct list node

Typvariablen

Welchen Typ hat `(:)`?

Typvariablen

Welchen Typ hat `(:)`?

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

Typvariablen

Welchen Typ hat `(:)`?

```
> :type (:)
(:) :: 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 `(:)`?

```
> :type (:)
(:) :: 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 `(:)`?

```
> :type (:)
(:) :: 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

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.

Klammerkonvention

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

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] ]
```

Die Definition der Konstanten darf auch ein zusammengesetzter Ausdruck sein, wie hier im letzten Beispiel gezeigt.

Top-level Konstanten werden maximal einmal ausgewertet.

Funktionsdefinitionen

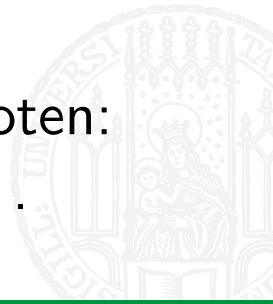
- Funktionsrumpf ist immer ein Ausdruck (z.B. ein Wert), innerhalb dieses Ausdrucks dürfen verwendet werden:
  - Funktionsparameter
  - *alle* gültigen Top-Level Definitionen

    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`,...



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
```

IF-THEN-ELSE

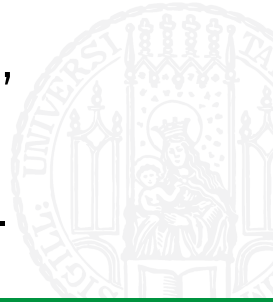
Ein wichtiges Konstrukt in vielen Programmiersprachen ist das **Konditional**; es erlaubt auf Bedingungen `::` `Bool` zu reagieren:

`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 zur jeweiligen Definition aus
- Lokale Definitionen werden höchstens einmal ausgewertet, auch wenn diese mehrfach im Ausdruck verwendet werden
- Lokale Definition können 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 stellen nachrangige lokale Definitionen gerne nach hinten. 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 ausschließlich für Top-Level Definitionen

Überschatten

```
schatten1  =  let x  = 1 in
              let y1 = x in
              let x  = 2 in
              let y2 = x in
              let x  = 3 in
              [x,y1,x,y2,x]
```

Was kommt heraus?

Überschatten

```
schatten1  =  let x  = 1 in
              let y1 = x in
              let x  = 2 in
              let y2 = x in
              let x  = 3 in
              [x,y1,x,y2,x]
```

Was kommt heraus? `[3,1,3,2,3]`

- Der Wert einer Variable/Bezeichners wird nie verändert.
  $\Rightarrow$ Referentielle Transparenz
- Die `let`-Definition führt einen *neuen* Bezeichner ein.
- Ein eventuell bereits vorhandener Bezeichner mit gleichem Namen wird **überschattet**, d.h. existiert weiterhin unverändert, aber ist nicht mehr ansprechbar.

Überschatten

Eine Überschattung kann auch nur vorübergehend stattfinden:

```
schatten2 = let x = 1          -- 1. Zeile
                y = let x = 2 -- 2. Zeile
                    in x      -- 3. Zeile
                z = x         -- 4. Zeile
            in [x,y,z]

schatten2 == [1,2,1]
```

Definition von `x` in Zeile 2 überschattet die Definition von `x` in Zeile 1 innerhalb des gesamten `let`-Ausdrucks (Zeile 2–3). Außerhalb dieses `let`-Ausdrucks gilt die Definition von `x` in Zeile 1. Diese wird z.B. dann wieder in Zeile 4 verwendet.

Keine Überschattung

Da ein Bezeichner für den gesamten Ausdruck gilt, darf man innerhalb eines einzigen `let`-Ausdruck den gleichen Bezeichner nur einmal definieren:

```
schatten3 = let x  = 1        -- 1. Zeile
                y1 = x        -- 2.
                x  = 2        -- 3.
                y2 = x        -- 4.
                x  = 3        -- 5.
            in [x,y1,x,y2,x] -- 6.
```

```
> :r
schatten.hs:1:17: error:
  Conflicting definitions for ‘x’
  Bound at: schatten.hs:1:17
            schatten.hs:3:17
            schatten.hs:5:17
  |
1 | schatten3 = let x  = 1       -- 1. Zeile
  |                 ^^^^^^^^^^^^^^^^^^^^^^^^...
Failed, no modules loaded.
```

Keine Überschattung

```
schatten4 = let x  = 1        -- 1. Zeile
                y  = x        -- 2.
            in                -- 3.
            let x  = x + 1    -- 4.
                z  = x        -- 5.
            in [y,x,z]        -- 6.
> schatten4
[1,
```

Fallstrick: Berechnung beendet sich nicht mehr!

In Zeile 4 wird *nicht* der in Zeile 1 definiere Wert verwendet, sondern der gerade definierte; da in Haskell alle `let`-Definitionen rekursiv sind. Rekursion behandeln wir gleich in Kapitel 3

Momentan gilt für uns also noch:
Links definierte Variablen im gleichen `let` nicht rechts verwenden!

Terme und Variablen

Terme enthalten **gebundene** und **ungebundene Variablen**.
Ungebundene Variablen nennt man auch **freie** / **offene Variablen**.

Beispiel:

$$\forall x \in \mathbb{N} \quad . \quad x + 1 \geq z$$

Im Beispiel ist die Variable x ist gebunden, z dagegen nicht.

Beispiel:

```
foo x = \y -> let z = u in x + y + z + v
```

Variablen `x`,`y` und `z` sind gebunden; Variablen `u`, `v` sind hier frei.

In Haskell werden Variablen immer durch **Pattern-Matching** gebunden. Pattern-Matching kommt an vielen Stellen vor: Funktionsdefinition, Lambda, Lokale Definition, ...

Terme und Variablen

Terme enthalten **gebundene** und **ungebundene Variablen**.
Ungebundene Variablen nennt man auch **freie** / **offene Variablen**.

Beispiel:

$$\forall x \in \mathbb{N} \quad . \quad x + 1 \geq z$$

Im Beispiel ist die Variable x ist gebunden, z dagegen nicht.

Beispiel:

```
foo x = \y -> let z = u in x + y + z + v
```

Variablen `x`,`y` und `z` sind gebunden; Variablen `u`, `v` sind hier frei.

In Haskell werden Variablen immer durch **Pattern-Matching** gebunden. Pattern-Matching kommt an vielen Stellen vor: Funktionsdefinition, Lambda, Lokale Definition, ...

Terme und Variablen

Terme enthalten **gebundene** und **ungebundene Variablen**.
Ungebundene Variablen nennt man auch **freie** / **offene Variablen**.

Beispiel:

$$\forall x \in \mathbb{N} \quad . \quad x + 1 \geq z$$

Im Beispiel ist die Variable x ist gebunden, z dagegen nicht.

Beispiel:

```
foo x = \y -> let z = u in x + y + z + v
```

Variablen `x`,`y` und `z` sind gebunden; Variablen `u`, `v` sind hier frei.

In Haskell werden Variablen immer durch **Pattern-Matching** gebunden. Pattern-Matching kommt an vielen Stellen vor: Funktionsdefinition, Lambda, Lokale Definition, ...

Terme und Variablen

Terme enthalten **gebundene** und **ungebundene Variablen**.
Ungebundene Variablen nennt man auch **freie** / **offene Variablen**.

Beispiel:

$$\forall x \in \mathbb{N} \quad . \quad x + 1 \geq z$$

Im Beispiel ist die Variable x ist gebunden, z dagegen nicht.

Beispiel:

```
foo x = \y -> let z = u in x + y + z + v
```

Variablen `x`,`y` und `z` sind gebunden; Variablen `u`, `v` sind hier frei.

In Haskell werden Variablen immer durch **Pattern-Matching** gebunden. Pattern-Matching kommt an vielen Stellen vor: Funktionsdefinition, Lambda, Lokale Definition, ...

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 der zuerst definierte Rumpf ausgewertet. Die Muster werden also *von oben nach unten* mit dem Argument verglichen.
- GHC warnt uns vor Definitionen mit unsinnigen Mustern.

Wildcards

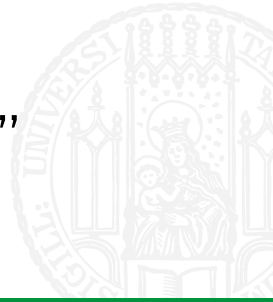
Das Muster "Variable" besteht jeden Vergleich:

```
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 (kopf : rumpf) =  False
```

Variable `kopf` hat Typ `a` und `rumpf` hat Typ `[a]`; beachten!

Listen-Muster

Mit Patterns können wir auch Listen auseinander nehmen:

```
null :: [a] -> Bool
null []           =  True
null (kopf : rumpf) =  False
```

Variable `kopf` hat Typ `a` und `rumpf` hat Typ `[a]`; beachten!

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, String)
sum []         = (0,    , "a: Leere Liste")
sum [x]        = (x     , "b: Genau 1 Element")
sum [x,y]      = (x+y   , "c: Genau 2 Elemente")
sum [x,y,z]    = (x+y+z, "d: Genau 3 Elemente")
sum (x:y:z:r) = (x+y+z, "e: Mindestens 3 oder mehr")
sum _          = (0     , "f: Beliebige Liste")

> sum [1,2,3]
(6,"d: Genau 3 Elemente")
```

Muster Überlappung

- Letztes Muster kann in diesem Beispiel nie erreicht werden, da immer ein vorheriges Muster zutreffen wird
- Auch Muster d & e überlappen, d.h. Reihenfolge ist wichtig

Listen-Muster

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

```
sum :: [Int] -> (Int, String)
sum []         = (0,    , "a: Leere Liste")
sum [x]        = (x     , "b: Genau 1 Element")
sum [x,y]      = (x+y   , "c: Genau 2 Elemente")
sum [x,y,z]    = (x+y+z, "d: Genau 3 Elemente")
sum (x:y:z:r) = (x+y+z, "e: Mindestens 3 oder mehr")
sum _          = (0     , "f: Beliebige Liste")
```

Beachte:

Variablen `x`,`y`,`z` matchen Elemente der Liste, d.h. haben den Typ `a`; aber Variable `r` hat den Typ `[a]`, da diese den gesamten Rumpf der Liste matched!

- Letztes Muster kann in diesem Beispiel nie erreicht werden, da immer ein vorheriges Muster zutreffen wird
- Auch Muster d & e überlappen, d.h. Reihenfolge ist wichtig

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 zu Recht vor solch unvollständigen Mustern.
Wenn man möchte, kann man auch die Funktion
`error :: String -> a` verwenden:

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

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 zu Recht vor solch unvollständigen Mustern.

Es gibt immer wieder Vorschläge, partielle Funktionen aus der `Prelude` zu verbannen und viele Alternativen, z.B. `ClassyPrelude`. Das ist sinnvoll, aber meist sind dort noch weitere Dinge geändert, was Anfänger sehr verwirren kann.

```
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

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

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

Wenn ein Pattern fehlschlägt (z.B. `(h:t)` auf leere Liste), wird bei Comprehensions einfach kein entsprechendes Element generiert.

In den ersten 3 Fällen führt dies jedoch zu einer Fehlermeldung! Hier also besser nur Muster verwenden, welche nie fehlschlagen können, z.B. Variablen, Wildcards, Tupel-Patterns, ...

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 _   = "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 _   = "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`
- Ein Wächter kann auch mehrere mit Komma getrennte Bedingungen haben: `| n>0, n/=7, n/=9  =  42`

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

Ähnlich wie bei List-Comprehensions ist auch noch der Rückpfeil für weitere Pattern-Matches und mehrere mit Komma getrennte Wächter erlaubt:

```
habGeldNachZahlung :: Int -> Int -> String
habGeldNachZahlung konto zahlung
  | -1 <- signum (konto - zahlung)
  , zahlung > 0
    = "Kannst Du Dir nicht leisten!"
  | -1 <- signum (konto - zahlung)
    = "Armer Schlucker!"
  |  1 <- signum (konto - zahlung)
  , zahlung > 0
    = "Bist echt reich!"
  | otherwise
    = "Alles im ok."
```

Pattern Guards

Ähnlich wie bei List-Comprehensions ist auch noch der Rückpfeil für weitere Pattern-Matches und mehrere mit Komma getrennte Wächter erlaubt:

```
habGeldNachZahlung :: Int -> Int -> String
habGeldNachZahlung konto zahlung
  | kontostatus < 0, zahlung > 0
    = "Kannst Du Dir nicht leisten!"
  | kontostatus < 0
    = "Armer Schlucker!"
  | kontostatus > 0, zahlung > 0
    = "Bist echt reich!"
  | otherwise
    = "Alles im ok."
  where
    kontostatus = signum (konto - zahlung)
```

eine `where`-Klausel reicht aber auch hier oft aus

Zusammenfassung

Im Kapitel 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 (:: 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 :: 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 :: 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 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

Typsignatur

```
foo :: a -> b -> c -> d
```

bedeutet: `foo` ist eine Funktion, welche Ihre drei Argumente mit Typ `a`, `b` und `c` auf ein Ergebnis des Typs `d` abbildet.

Es ist empfehlenswert, explizite Typsignaturen anzugeben:

1. Um die Gedanken beim Programmieren zu sortieren
2. Als Dokumentation des Codes
3. Für bessere Fehlermeldungen, da unnötig allgemeine Typen zu längeren Fehlermeldungen führen können
4. Spezialfälle Monomorphism-Restriction

Tipp

Falls man den Typ nicht versteht, dann einfach in GHCi mit `:type` den Typ bestimmen und in den Programmtext hineinkopieren.

Anonyme Funktionen

Wie auf Folie 1.44 erklärt, sind anonyme Funktionen einfach nur Funktionen ohne Namen. Es ist eine einfache, in der Mathematik gebräuchliche Notation, um über Funktionen zu reden.

Beispiel:

`(\x -> 3*x) 7` entspricht `let tri x = 3*x in tri 7`

In der ersten Variante schreiben wir in der Funktionsanwendung anstatt des Namens `tri` direkt die gesamte Funktionsdefinition hin!

Beispiel mit mehreren Argumenten und Pattern-Matching

Dieser Ausdruck `( \ (x,y) z -> x+y+z ) (1,2) 3`

entspricht also `let f (x,y) z = x+y+z in f (1,2) 3`

Für Pattern-Matching mit mehreren Fällen oder Wächtern muss man einen Case-Ausdruck verwenden: `\x -> case x of ...`

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
   round' z = round z
```

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
   round' z = round z
```

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)
```