Einführung in die Funktionale Programmierung mit Haskell

List-Comprehension, Patterns, Guards

Steffen Jost

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

24. April 2013

Letzte Vorlesung

Der Vollständigkeit wegen hier eine Liste von Diskussionspunkten, auf welche in der letzten Vorlesung zum Teil aufgrund von Fragen vorgegriffen wurde:

- Typabkürzung mit `newtype`
- Kommentare
- List Ranges `[0..1000]`
- Verschiedene Möglichkeiten der Funktionsdefinition
- Konditional `if-then-else`

Diese Punkte werden wir heute noch einmal ausführlicher betrachten und vervollständigen.

NEWTYPE

Foliensatz 02, “Typabkürzung” S.16:
Mit dem Schlüsselwort `type` können Typakürzung definiert werden, um die Lesbarkeit von Programmen zu erhöhen.

- In der vorangegangenen Vorlesung wurde von einem Teilnehmer richtig bemerkt, dass Typabkürzungen mit `type` beliebig vertauschbar sind, was zwar praktisch, aber manchmal ungewollt sein kann, z.B. wenn man eine Anzahl Äpfel und Anzahl Birnen als `Int` repräsentiert, aber versehentlich Vertauschungen ausschliessen möchte.
- Wir betrachteten deshalb als Antwort auf dieses Problem `newtype`, mit dem dies verhindert werden kann.
- `newtype` ist ein optimierter Spezialfall einer Datentypdeklaration.
  Im Kapitel Datentypen werden wir das genauer betrachten.

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 constant
   values for our high-precision
   computations here.                -}
pi = 3.0
e  = 2.7
```

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]`
  - 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]`
  - 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]`
  - 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]`
  - 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:

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

"Die Liste aller x^2, so dass 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 grundlegender in Haskell.

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:

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

"Die Liste aller x^2, so dass 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 grundlegender in Haskell.

List-Comprehension

$$[\, x^2 \mid x <- [1..10],\ odd\ x \,]$$

Rumpf: bestimmt wie ein Listenelement berechnet wird

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

Bedingung: entscheidet, ob der Wert in der erzeugten 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, Bedingung und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden!

List-Comprehension

$$[\, x^2 \mid x <- [1..10],\ odd\ x \,]$$

Rumpf: bestimmt wie ein Listenelement berechnet wird

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

Bedingung: entscheidet, ob der Wert in der erzeugten 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, Bedingung und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden!

List-Comprehension

$$[\, x^2 \mid x <- [1..10],\ \mathit{odd}\ x \,]$$

Rumpf: bestimmt wie ein Listenelement berechnet wird

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

Bedingung: entscheidet, ob der Wert in der erzeugten 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, Bedingung und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden!

List-Comprehension

$$[\, x^2 \mid x <- [1..10],\ odd\ x \,]$$

Rumpf: bestimmt wie ein Listenelement berechnet wird

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

Bedingung: entscheidet, ob der Wert in der erzeugten 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, Bedingung und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden!

Beispiele

Beliebig viele Generatoren, Bedingung 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.

```
> [ (x,y) | x <-[1,3..9], y <- [8,7..x], x+y < 8]
[(1,6),(1,5),(1,4),(1,3),(1,2),(1,1),(3,4),(3,3)]
```

Vorangehende Definition können "weiter rechts" verwendet werden.

```
> [ (x,y,z) | x <-[1,3..9], y <- [8,7..4],
              x < y, let z = x * y - x, z > x + y ]
[(3,8,21),(3,7,18),(3,6,15),(3,5,12),(3,4,9),
 (5,8,35),(5,7,30),(5,6,25),(7,8,49)]
```

Beispiele

Beliebig viele Generatoren, Bedingung 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.

```
> [ (x,y) | x <-[1,3..9], y <- [8,7..x], x+y < 8]
[(1,6),(1,5),(1,4),(1,3),(1,2),(1,1),(3,4),(3,3)]
```

Vorangehende Definition können “weiter rechts” verwendet werden.

```
> [ (x,y,z) | x <-[1,3..9], y <- [8,7..4],
              x < y, let z = x * y - x, z > x + y ]
[(3,8,21),(3,7,18),(3,6,15),(3,5,12),(3,4,9),
 (5,8,35),(5,7,30),(5,6,25),(7,8,49)]
```

Beispiele

Beliebig viele Generatoren, Bedingung 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.

```
> [ (x,y) | x <-[1,3..9], y <- [8,7..x], x+y < 8]
[(1,6),(1,5),(1,4),(1,3),(1,2),(1,1),(3,4),(3,3)]
```

Vorangehende Definition können "weiter rechts" verwendet werden.

```
> [ (x,y,z) | x <-[1,3..9], y <- [8,7..4],
              x < y, let z = x * y - x, z > x + y ]
[(3,8,21),(3,7,18),(3,6,15),(3,5,12),(3,4,9),
 (5,8,35),(5,7,30),(5,6,25),(7,8,49)]
```

Beispiele

Es darf natürlich verschachtelt werden...

```
>  [ [x,x-1..0] | x <-[1,3..5]]
[[1,0],[3,2,1,0],[5,4,3,2,1,0]]

> [ q | q <- [ x^2 | x <- [1..20], even x],
        q > 10, q < 200]
[16,36,64,100,144,196]
```

...aber das kann schnell unübersichtlich werden!

```
> [ c | c <- "Donaudampfschiffer",
        not (c `elem` ['a','e','i','o','u'])]
"Dndmpfschffr"
```

Konstanten und Funktionen können es übersichtlicher machen:

```
> [ c | c <- "Donaudampfschiffer", istKonsonant c]
"Dndmpfschffr"
```

Beispiele

Es darf natürlich verschachtelt werden...

```
>  [ [x,x-1..0] | x <-[1,3..5]]
[[1,0],[3,2,1,0],[5,4,3,2,1,0]]

> [ q | q <- [ x^2 | x <- [1..20], even x],
        q > 10, q < 200]
[16,36,64,100,144,196]
```

...aber das kann schnell unübersichtlich werden!

```
> [ c | c <- "Donaudampfschiffer",
        not (c `elem` ['a','e','i','o','u'])]
"Dndmpfschffr"
```

Konstanten und Funktionen können es übersichtlicher machen:

```
> [ c | c <- "Donaudampfschiffer", istKonsonant c]
"Dndmpfschffr"
```

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:

```
> 'a':['b','c']
"abc"

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

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

`(:)` 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:

```
> 'a':['b','c']
"abc"

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

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

`(:)` 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]
```

Wir wissen, dass Typen in Haskell immer mit einem Großbuchstaben beginnen müssen. a ist also kein Typ, sondern eine Typvariable. Typvariablen stehen für *einen* beliebigen Typen. Typvariablen werden immer klein geschrieben.

Typvariablen

Welchen Typ hat `(:)`?

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

Wir wissen, dass Typen in Haskell immer mit einem Großbuchstaben beginnen müssen. a ist also 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
(:) 3 :: [Integer] -> [Integer]
> :t ('a' :)
('a' :) :: [Char] -> [Char]
```

Typvariablen

Welchen Typ hat `(:)`?

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

Wir wissen, dass Typen in Haskell immer mit einem Großbuchstaben beginnen müssen. a ist also 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
(:) 3 :: [Integer] -> [Integer]
> :t ('a' :)
('a' :) :: [Char] -> [Char]
```

Der Ausdruck `'a':[1]` ergibt aber einen Typfehler, da er den Typ `(:) :: a -> [b] -> [b]` benötigen würde.

⇒ Polymorphie

Funktionstypen (Wdh.)

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

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.)

Im letzten Code-Beispiel der vorangegangenen Vorlesung sahen wir verschiedene Möglichkeiten, eine Funktionen (mit mehreren Argumenten) zu definieren. Dies werden wir heute noch ausführlicher behandeln.

Generell gilt jedoch, das wir zuerst

- den Funktionstyp deklarieren optional, aber empfehlenswert
- dann den Funktionsnamen gefolgt von den Argumenten
- nach einem = den Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck!

Beispiel:

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

Konstanten

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

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

myLectures :: [String]
myLectures = ["EFP","LDS"]

myNoLectures :: Int
myNoLectures = length myLectures

squareNumbers :: [Int]
squareNumbers = [ x*x | x <- [1..] ]
  -- wird maximal einmal berechnet
```

Funktionsdefinitionen

- Der Rumpf einer Funktion ist immer ein Ausdruck
  - Innerhalb des definierenden Ausdrucks der Funktion können beliebige andere Funktionen verwendet werden.
    Genauer: alle geltenden Definitionen können verwendet werden
  - Die Reihenfolge der Definitionen innerhalb einer 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 zu reagieren:

```
if <bedingung> then <tue_das> else <sonst_dies>
```

In einer funktionalen Sprache wie Haskell ist dies ein Ausdruck, kein Befehl! Der Else-Zweig ist damit nicht optional.

Die `then`/`else`-Ausdrücke können natürlich keine Seiteneffekte, sondern nur ein Ergebnis haben.

If-then-else in Haskell entspricht also dem "· ? · : ·"-Ausdruck in C oder Java.

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

Beispiel

Damit können 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
```

Beispiel

Damit können 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 x in
  let y' = abs y in
  x'+y'
```

- Die Namen `x'` und `y'` sind nur innerhalb des Funktionsrumpfes benutzbar.
- Eine lokale Definition wird nur höchstens einmal ausgewertet
- Eine lokale Definition kann auch wieder eine komplette Funktion definieren

sogar mit Abschnittsweisem Pattern-Matching

Layout

Haskell ist "whitespace"-sensitiv, d.h. man kann mit der richtigen Einrückung viel Tipparbeit sparen, und das Programm lesbarer machen:

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

Mehrere lokale Definitionen benötigen nur einen `let`-Ausdruck, wenn alle Definitionen in der gleichen Spalte beginnen.

Ein Vorteil hierbei ist, dass sich dann alle Definitionen gegenseitig verwenden dürfen.

Wer unbedingt will, kann anstatt Einrückung auch mit `{ }` und `;` arbeiten.

Where-Klausel

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

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

Auch hier ist wieder die **Layout-Regel** zu beachten.

`where` kann alles, was ein `let`-Ausdruck auch kann.
Achtung: `where` ist kein Ausdruck, sondern eine spezielle Syntax.
Es darf pro Definition nur eine `where`-Klausel benutzt werden.

Eine `where`-Klausel kann im Gegensatz zu einem `let`-Ausdruck über mehrere Wächter erstrecken.

Wächter ⇒ später in dieser Vorlesung

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 = show x
```

- 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.
- Haskell kann uns vor der Definition von unsinnigen Mustern warnen.

Wildcards

Das Muster "Variable" besteht jeden Vergleich:

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

Man kann das Muster _ verwenden, wenn der Wert egal ist. Dies ist oft lesbarer, da man sofort erkennt, welche Argumente verwendet werden:

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

Es vermeidet auch die hilfreiche Warnung "defined but not used"
Es ist auch möglich, benannte Wildcards zu verwenden `_kosten`

Tupel-Muster

Mit Patterns können wir auch ganz einfach 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 (_:_)  =  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 (x:y:z:_ )  = x+y+z -- eh, too many


> 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` aufrufen:

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

Muster kombinieren

Wir können verschiedene Muster natürlich genauso verschachteln, wie wir Listen und Tupeln (und andere Datentypen) ineinander verschachteln können.

Manchmal ist es hilfreich, ein Muster in verschiedene Teile zu zerlegen. Dies kann man mit as-Patterns erreichen. Dazu schreibt man einen Variablennamen, gefolgt von dem @-Symbol vor ein in einer Klammer gefasstes Muster. Das ganze ist wieder ein Muster.

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

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

wobei die Infixfunktion (++) in der Standardbibliothek definiert ist, und einfach zwei Listen miteinander verkettet.

Pattern-Matching überall

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

- linke Seite eines Generators in List-Comprehensions:

```
> [x| (x,_,2)<-[(1,9,2),(2,8,3),(3,4,2),(5,6,2)]]
[1,3,5]
```

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

- linke Seite von `let`-Definitionen
- `where`-Klauseln

In den letzten beiden Fällen sollte man drauf achten, dass das Pattern-Matching nicht fehlschlagen kann (wenn keine Funktion definiert wird), also z.B. kann man immer eine Tupel auseinander nehmen, aber ein Pattern-Match mit einer Liste ist nicht sicher.

Case Ausdruck

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

Der `case`-Ausdruck kann wie jeder andere Ausdruck verwendet werden:

```
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 imperativer Sprachen würden entsprechend schreiben:

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

Meist geht es jedoch ohne case-Ausdruck etwas schöner:

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

Beispiel

Freunde imperativer Sprachen würden entsprechend schreiben:

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

Meist geht es jedoch ohne case-Ausdruck etwas schöner:

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

Beispiel

Freunde imperativer Sprachen würden entsprechend schreiben:

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

Meist geht es jedoch ohne case-Ausdruck etwas schöner:

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

Wächter

Ein Musterabgleich, der zur Fallentscheidung genutzt wird (Funktionsdefinition, Case), kann zusätzlich mit Wächtern oder **Guards** versehen werden. Ein **Wächter** ist eine Bedingung, also ein Ausdruck des Typs `Bool`, der zusätzlich alle Variablen des bewachten Patterns verwenden darf:

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

Es wird der erste Zweig gewählt, welcher zu `True` auswertet.

Wächter-Patterns sind besser lesbar als verschachtelte if-then-else.

`otherwise` ist **kein Schlüsselwort**, sondern eine Konstante der Standardbibliothek und gleich `True`.
Es ist einfach lesbarer als direkt `True` hinzuschreiben.

Pattern Guards

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

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

Wie bei List-Comprehensions ist dann auch der Rückpfeil <- ist erlaubt, um weitere Pattern-Matches auf anderen Ausdrücken durchzuführen.

Pattern Guards

Zusätzlich zu bool'schen Bedingung, können auch noch weitere Pattern-Matches auf anderen Ausdrücken durchgeführt werden.

```
habGeldNachAktion :: Int -> Int -> String
habGeldNachAktion 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

Zusätzlich zu bool'schen Bedingung, können auch noch weitere Pattern-Matches auf anderen Ausdrücken durchgeführt werden.

```
habGeldNachAktion :: Int -> Int -> String
habGeldNachAktion 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 hier auch oft und ist schöner

Wächter

Wächter jeglicher Art treten immer in Verbindung mit einem Pattern-Match auf, und man kann natürlich von beide gleichzeitig verwenden, auch wenn in den bisher behandelten Beispielen die Pattern immer fail-safe waren:

```
concatReplicate :: Int -> [a] -> [a]  -- replicates each
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

concatReplicate n = concat . map (replicate n)
```

Wächter

Wächter jeglicher Art treten immer in Verbindung mit einem Pattern-Match auf, und man kann natürlich von beide gleichzeitig verwenden, auch wenn in den bisher behandelten Beispielen die Pattern immer fail-safe waren:

```
concatReplicate :: Int -> [a] -> [a]  -- replicates each
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

concatReplicate n = concat . map (replicate n)
```