

Pattern Matching

Prof. Dr. Johannes Kinder

Ludwig-Maximilians-Universität München, Institut für Informatik

Lehrstuhl für Programmiersprachen und Künstliche Intelligenz

Programmierung und Modellierung, SoSe 2024

Zusammenfassung der letzten Vorlesung

- Funktionsbegriff
- Stelligkeit einer Funktion
- Infix-/Präfixnotation für Funktionsanwendung
- Funktionen sind Werte in funktionalen Sprachen
- Listenkonstruktion mit Comprehensions

$A \rightarrow B$

`(+) 1 2 `foo` 3`

`[x | x <- [1..9], even x]`

Quiz

- Was ist der Wert des Ausdrucks `[x | x <- [3,6..20], y <- [4,8..20], x == y]`
 - `[]`
 - `[4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]`
 - `[3,4,6,8,20]`
 - `[12]`
- Was ist der Wert des Ausdrucks `(*) 2 ((+) 3 4)`
 - Parsing-Fehler
 - `9`
 - `14`
 - `24`
- Was ist der Rückgabetypp der Funktion `f :: Integer -> (Char -> Bool)`
 - `Integer`
 - `Bool`
 - `Char -> Bool`
 - `(Char, Bool)`



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**
- Klammerkonvention
 - Funktionstypen sind implizit rechtsgeklammert:
`Integer -> Integer -> Integer` wird gelesen als `Integer -> (Integer -> Integer)`
 - Entsprechend ist die Funktionsanwendung implizit linksgeklammert:
`bar 1 8` wird gelesen als `(bar 1) 8`
- Das bedeutet: `(bar 1)` ist eine Funktion des Typs `Integer -> Integer`! Funktionen sind also normale Werte in einer funktionalen Sprache



Konstanten

- Die einfachste Definition ist eine Funktion ohne Argumente (Konstante)

```
course :: String  
course = "ProMo"
```

```
pi = 3.1415
```

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

- Top-level Konstanten werden maximal einmal ausgewertet



Funktionsdefinitionen

- Funktionsrumpf ist immer ein Ausdruck, innerhalb dessen dürfen verwendet werden:
 - Funktionsparameter
 - Alle gültigen Top-Level Definitionen (Reihenfolge der Definitionen innerhalb der Datei ist unerheblich; Typdeklaration schreibt man aber üblicherweise zuerst)
- Funktionsnamen müssen immer mit einem Kleinbuchstaben beginnen, danach folgt eine beliebige Anzahl an Zeichen:
 - Klein- und Großbuchstaben, Zahlen, Apostroph `'`, Unterstrich `_`
 - Beispiel: `thisIsAn_Odd_Fun'Name`
- Allerdings sind einige Schlüsselwörter verboten, z.B. `type`, `if`, `then`, `else`, `let`, `in`, `where`, `case`, ...

If-Then-Else

- Ein wichtiges Konstrukt in vielen Programmiersprachen ist das Konditional, **if-then-else**
- Sobald die Bedingung zu True oder False ausgewertet wurde, können wir den gesamten Ausdruck zu dem entsprechenden Zweig auswerten:

```
ghci> if True then 43  
else 23  
43
```

```
ghci> if False then 43  
else 23  
23
```

- 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 Java oder C

Beispiele

```
abs :: Integer -> Integer
```

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

```
signum :: Integer -> Integer
```

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

```
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
- Lokale Definition kann auch Funktionen definieren



Layout

- Einrückung gemäß Layout-Regel erspart Tipparbeit und erhöht die Lesbarkeit:

```
betragSumme :: Integer -> Integer -> Integer
```

```
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: vorherige Zeile geht weiter (gültige Spalte ist erstes Zeichen nach **let**)
 - gleiche Spalte: nächste lokale Definition
 - Spalte weiter links: Definition beendet



where-Klausel

- Wie in der Mathematik erlaubt Haskell auch, lokale Definition hintenanzustellen:

```
betragSumme :: Integer -> Integer -> Integer
```

```
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 selbst kein Ausdruck, sondern eine spezielle Syntax für Top-Level Definitionen



Pattern-Matching

- **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 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:

```
findZero :: Int -> Int -> Int -> String
```

```
findZero 0 _ _ _ = "Erstes"
```

```
findZero _ 0 _ _ = "Zweites"
```

```
findZero _ _ 0 _ = "Drittes"
```

```
findZero _ _ _ _ = "Keines"
```

- Zeigt an, welche Argumente verwendet werden und vermeidet Warnung „defined but not used“
- Auch benannte Wildcards sind möglich, z.B. `_cost`, um anzuzeigen, was nicht verwendet wird

Tupel-Muster

Patterns können Tupel zerlegen:

```
type Vector = (Double,Double)
```

```
addVectors:: Vector -> Vector -> Vector
```

```
addVectors (x1,y1) (x2,y2) = (x1+x2, y1+y2)
```

```
fst3 :: (a,b,c) -> a
```

```
fst3 (x,_,_) = x
```

```
snd3 :: (a,b,c) -> b
```

```
snd3 (_,y,_) = y
```



Listen-Muster

- Patterns können auch Listen zerlegen:

```
null :: [a] -> Bool
```

```
null [] = True
```

```
null (_head : _tail) = False
```

- Wir können auch spezielle Listen matchen:

```
count :: [a] -> String
```

```
count [] = "Null"
```

```
count [_] = "Eins"
```

```
count [_,_] = "Zwei"
```

```
count [_,_,_] = "Drei"
```

```
count _ = "Viele"
```



Listen-Muster (2)

- Mit Variablen lassen sich die Elemente der Liste verwenden:

```
sum3 :: [Integer] -> Integer
```

```
sum3 [] = 0
```

```
sum3 [x] = x
```

```
sum3 [x, y] = x + y
```

```
sum3 (x:y:z:_) = x + y + z
```

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

```
6
```

```
ghci> sum [1,2,3,4,5]
```

```
15
```



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:

```
ghci> head []
```

```
*** Exception: Prelude.head: empty list
```

- Das bedeutet, dass head und tail **partielle Funktionen** definieren
- Wenn es sich nicht vermeiden lässt, dann sollte man wenigstens die Funktion `error :: String -> a` verwenden:

```
myHead (h:_) = h
```

```
myHead [] = error "myHead of empty list undefined"
```

Verschachtelte Muster

- Verschiedene Muster dürfen kombiniert und verschachtelt werden

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

```
ghci> sumHeads [(1,2),(3,4),(5,6),(7,8)]  
[(4,6),(5,6),(7,8)]
```

- 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]
```

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


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

- let-Definitionen: `let (_,y) = ...`
- Definitionen in `where`-Klauseln
- Darauf achten, dass das Pattern-Matching nicht fehlschlagen kann
 - Tupel können z.B. immer erfolgreich matchen, aber Listen-Pattern können fehlschlagen und zu Ausnahmen führen

Case Ausdruck

- Ein Musterabgleich ist auch innerhalb eines beliebigen Ausdrucks möglich:

```
case <Ausdruck> of
```

```
  <Muster> -> <Ausdruck>
```

```
  <Muster> -> <Ausdruck>
```

```
  <Muster> -> <Ausdruck>
```

- Es gilt wieder die Layout-Regel
 - Alle Muster müssen in derselben Spalte 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

- Mit case Ausdruck

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

- Ohne case-Ausdruck, mit **where**:

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



Wächter / Guards

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

```
signum :: Int -> Int
```

```
signum n
```

```
    | n < 0    = -1
```

```
    | n > 0    = 1
```

```
    | otherwise = 0
```

- Die Wächter-Ausdrücke werden der Reihe nach ausgewertet
 - Es wird der erste Zweig gewählt, welcher zu True ausgewertet
 - Schlagen alle Wächter fehl, wird das nächste Pattern geprüft
 - **otherwise** ist kein Schlüsselwort, sondern eine Konstante mit dem Wert **True**



Vergleich der 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
```



Vergleich der 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
```



Kommentare

- Programme sollten immer sinnvoll kommentiert werden
- Einzeilige Kommentare
 - Haskell ignoriert bis zum Ende einer Zeile alles, was nach einem doppelten Minus kommt

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

- Mehrzeiliger Kommentar

- Für längeren Text, beginnen mit {- und werden mit -} beendet

```
{- We define some useful constants  
   for high-precision computations here. -}
```

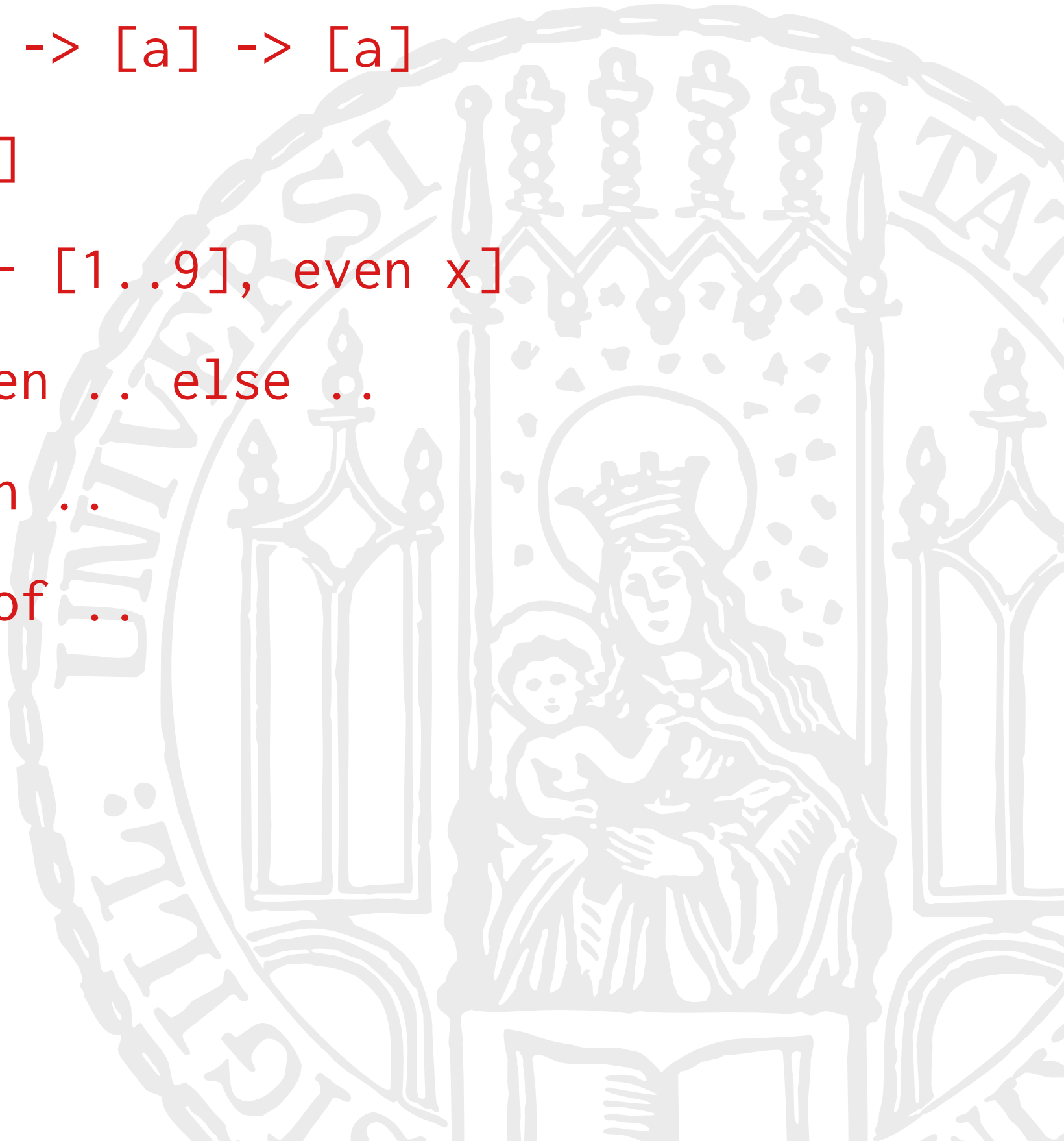
```
pi = 3.0
```

```
e  = 2.7
```



Zusammenfassung

- Behandelte Haskell Ausdrücke:
 - Cons-Constructor
 - Listen Aufzählungen
 - List-Comprehensions
 - Konditional
 - Lokale Definitionen
 - Pattern Matching
- Verschiedene Notationen zur Funktionsdefinition:



```
(:) :: a -> [a] -> [a]  
[1,3..99]  
[x | x <- [1..9], even x]  
if .. then .. else ..  
let .. in ..  
case .. of ..
```

Funktionsdefinitionen in Haskell

- Typdeklaration (optional)

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

Funktionsdefinitionen in Haskell

- Typdeklaration (optional)
- Funktionsname (in gleicher Spalte beginnen)

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


Funktionsdefinitionen in Haskell

- Typdeklaration (optional)
- Funktionsname (in gleicher Spalte beginnen)
- Funktionsparameter

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

Funktionsdefinitionen in Haskell

- Typdeklaration (optional)
- Funktionsname (in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf

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

Funktionsdefinitionen in Haskell

- Typdeklaration (optional)
- Funktionsname (in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche

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

Funktionsdefinitionen in Haskell

- Typdeklaration (optional)
- Funktionsname (in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter

```
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
```

Funktionsdefinitionen in Haskell

- Typdeklaration (optional)
- Funktionsname (in gleicher Spalte beginnen)
- Funktionsparameter
- Funktionsrumpf
- Fallunterscheidung durch Mustervergleiche
- Verfeinerung des Pattern-Match durch Wächter
- Nachgeschobene lokale Definitionen pro Funktionsgleichung

```
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
```

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 (2)

```
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
```

concatReplicateAux wird rekursiv aufgerufen (\Rightarrow Rekursion)

Alternative Definition (\Rightarrow Funktionen höherer Ordnung): `concatReplicate n = concatMap (replicate n)`