Groß- und Kleinschreibung in Haskell

Ausführliche Anleitungen zum Erlernen der Sprache gibt es z.B. unter `book.realworldhaskell.org/read/` oder `learnyouahaskell.com/chapters/`.

Haskell unterscheidet Groß- und Kleinschreibung;

- Namen für Typen und Module beginnen mit **Großbuchstaben**;
- Namen für Werte (auch Funktionen) und Typvariablen mit **Kleinbuchstaben** (oder _).

Schlüsselwörter dürfen nicht für Namen benutzt werden:
`case`, `class`, `data`, `default`, `deriving`, `do`, `else`, `foreign`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type`, `where`, _

Kommentare und Gültigkeitsbereiche in Haskell

Ein **Kommentar** beginnt mit -- und endet mit der Zeile.
Oder er beginnt mit {- und endet mit -}; diese Version ist schachtelbar.

Leerzeichen am Zeilenbeginn haben in Haskell eine Bedeutung!

In Haskell gilt die **Abseitsregel**, wenn nach einem der Schlüsselwörter `where`, `let`, `do` oder `of` keine öffnende Klammer { folgt:

- Dann wird diese Klammer { eingefügt und
- bei einer kürzeren Einrückung der nächsten Zeile eine schließende Klammer } bzw.
- bei gleicher Einrückung ein Semikolon.
- Bei einer längeren Einrückung wird nichts eingefügt.

Je nach Editor: Vorsicht beim Mischen von Leerzeichen und Tabulatoren!

Werte

Zu den **Werten** gehören Wahrheitswerte, Zahlen und Zeichen.
Werte können auf Gleichheit (==) oder Ungleichheit (/=) geprüft werden.

- Die **Wahrheitswerte** `True` und `False` bilden den Datentyp `Bool` mit Operationen `not` (Negation), `&&` (Konjunktion), `||` (Disjunktion).
- **Zahlen** gibt es ganzzahlig oder in Gleitpunktdarstellung.
  Für ganze Zahlen gibt es die Datentypen `Int` und `Integer` mit den Operationen `+`, `-`, `*`, `div` und `mod`.
  Eine Zahl des Datentyps `Int` wird durch 32 Bit beschrieben und liegt daher im Intervall $[-2.147.483.648 \ldots 2.147.483.647]$;
  der Datentyp `Integer` hat keine Größenbeschränkung.

  Für Gleitpunktzahlen gibt es die Datentypen `Float` (32 Bit) und `Double` (64 Bit). Der Operator / erzeugt einen Wert vom Typ `Float`.
- Für **Zeichen** gibt es den Datentyp `Char`.
  Alle Symbole von Unicode können bearbeitet werden.
  Zeichen werden in einfachen Anführungsstrichen notiert, z.B. `'a'`.

Tupel (kartesisches Produkt)

Mehrere Werte können zu einem **Tupel** zusammengefasst werden, z.B. `(False, 2013)`.

Runde Klammern machen aus mehreren –durch Komma getrennten– Werten *ein* Tupel.

Ein Tupel hat eine feste Stelligkeit.
Ein Tupel kann Werte unterschiedlicher Typen enthalten.
Auf die Komponenten eines Paares kann mit den Funktionen `fst` und `snd` zugegriffen werden.

Beispiel

Ein Punkt in der Ebene wird durch zwei Koordinaten beschrieben.
Für den entsprechenden Typ `(Double, Double)` kann man ein prägnantes Synonym einführen:

```
type Punkt = (Double, Double)
```

Aufzählungstypen und direkte Summe

Ein **algebraischer Datentyp** wird durch Aufzählung der Werte definiert:

```
data Color = Red | Green | Blue
```

Der **Typ-Konstruktor** `Color` erzeugt einen Typ mit allen Werten, die durch die Daten-Konstruktoren auf der rechten Seite beschrieben werden.
Jedem **Daten-Konstruktor** folgen die Typen seiner Komponenten.
Alle Konstruktoren beginnen mit Großbuchstaben.
Der Daten-Konstruktor `Red` hat keine Parameter, ist also eine Konstante.

Beispiel

Ein Datentyp für Kreise (beschrieben durch Mittelpunkt und Radius) und Rechtecke (beschrieben durch zwei gegenüberliegende Eckpunkte):

```
data Geometrie = Kreis Punkt Double | Rechteck Punkt Punkt
```

Die Daten-Konstruktoren `Kreis` und `Rechteck` haben je zwei Parameter.

Aufzählungstypen und direkte Summe

Auch rekursive Datentypen sind möglich:

Beispiel

```
data IntTree = Empty | Branch IntTree Int IntTree
```

Hier werden binäre Bäume mit ganzzahligen Knotenmarkierungen definiert als Typ `IntTree` mit den Werten `Empty` und `(Branch l x r)` für beliebige ganze Zahlen `x` und Werte `l` und `r` vom Typ `IntTree`.

Listen

Mehrere Daten können in Haskell zu einer Liste zusammengefasst werden.

Eine **Liste** kann verlängert oder verkürzt werden, hat also keine feste Anzahl von Elementen.

Haskell kennt nur **homogene Listen**: alle Elemente sind vom gleichen Typ.

```
[1,2,3]                    -- eine Liste mit drei Elementen
[ ]                        -- die leere Liste mit null Elementen
```

Listen: der cons-Operator

Das Einfügen eines Elements am Anfang erledigt der **:-Operator** (*cons*).

Der `:`-Operator ist rechtsassoziativ,

also `1:2:3:[ ] = (1:(2:(3:[ ])))`.

Die Liste `[1,2,3]` ist also eine Abkürzung für `1:2:3:[ ]`;
sie hat mehrere Darstellungen:

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

Listen haben also die beiden Daten-Konstruktoren `[ ]` und `:` .

Umgekehrt kann man mit dem `:`-Operator eine Liste zerlegen in einen

- **Kopf** (*head*, das erste Element der Liste) und
- **Rumpf** (*tail*, die Liste mit allen restlichen Elementen)

Listen

Vordefinierte Operationen auf Listen sind z.B.:

- `head x` liefert das erste Element einer Liste `x`
- `tail x` liefert den Rest der Liste `x`, ohne das erste Element
- `last x` liefert das letzte Element einer Liste `x`
- `init x` liefert den Anfang der Liste `x`, ohne das letzte Element
- `x ++ y` verbindet zwei Listen `x` und `y` gleichen Typs zu einer Liste (**Konkatenation**).

Das n-te Element einer Liste `x` erhält man mit `x !! n`.
Achtung: Die Indexzählung beginnt bei Null!
`[1,2,3,4] !! 2` liefert also 3.

Listen von Zeichen: Strings

Der Typ einer Liste mit Elementen von Typ a wird mit `[a]` bezeichnet.

Für **Listen von Zeichen** gibt es eine Abkürzung:
`['H','a','s','k','e','l','l']` schreibt man kurz `"Haskell"` .

Für den zugehörigen Typ `[Char]` ist der Name `String` vordefiniert.
Der leere String `""` ist ein Synonym für eine leere Liste `[ ]` von Zeichen.

Fallunterscheidung in Haskell

Verschiedene Fälle können mit dem case-Konstrukt unterschieden werden. Die Ergebnisse aller Fälle müssen vom selben Typ sein.

Beispiel

Um das Vorzeichen einer Zahl `x` zu ermitteln, kann man nebenstehende Fallunterscheidung nutzen:

```
case x of
     x>0     ->  1
     x==0    ->  0
     x<0     -> -1
```

Fallunterscheidung in Haskell

Das traditionelle

```
if e1 then e2 else e3
```

ist dann nur eine Abkürzung für:

```
case e1 of
    True    -> e2
    False   -> e3
```

In funktionalen Sprachen gibt es das `if-then-else` nur für *Ausdrücke*, d.h. hinter `then` bzw. `else` steht keine Anweisung, sondern ein Ausdruck, der einen Wert liefert.

Der `else`-Teil kann daher nicht weggelassen werden!

Ein `if-then-else`-Ausdruck kann Teil eines größeren Ausdrucks sein.

Funktionen

Eine Funktion in Haskell besteht aus Definitionen und einem Ausdruck, dessen Wert als Funktionsergebnis zurückgeliefert wird.
(Wir schreiben Funktionsdefinitionen in eine Datei, starten GHCI aus dem Verzeichnis der Datei und laden dann diese Datei).

Werden **Parameter** in imperativen Programmiersprachen durch Komma getrennt und die Parameterliste geklammert, z.B. `max(2,1)`,
so werden sie in funktionalen Programmiersprachen nur durch Zwischenraum getrennt hinter den Funktionsnamen geschrieben: `max 2 1`.

Klammern werden nötig, wenn ein Parameter erst berechnet werden muss:
`max (min 2 3) 1`.

Beispiel (Die Wirkung von Klammern; Fakultät)

```
fac n = if n==0 then 1          > fac 5  > fac (5-1)  > fac 5-1
        else n * fac (n-1)      120      24           119
```

Funktionen

Eine Funktion wird durch (eine oder mehrere) Gleichungen definiert. Nach dem Funktionsnamen folgen –durch Zwischenraum getrennt– die Parameter und das Gleichheitszeichen, danach die Berechnungsvorschrift.

Ist der Parameter vom Typ `t` und die Berechnungsvorschrift vom Typ `u`, so hat die Funktion den Typ `t -> u`.

Beispiel für die Definition einer Funktion (lies `=` als „ist definiert als“):

```
dec x = x-1
```

Funktionen ohne Parameter definieren **Konstanten**:

```
e = 2.71828
```

Mustervergleich

Eine Funktionsdefinition kann übersichtlich in mehrere definierende Gleichungen zerlegt werden.

Dabei kann eine Fallunterscheidung für die Parameter auf der *linken* Seite der Definition vorgenommen werden.

Bei einer Funktionsanwendung wird die erste Definition verwendet, die für das aktuelle Argument passt (**Mustervergleich**, *pattern matching*).

Beispiel

Die Definition `xor x y = x /= y` für die boolesche Funktion `xor` (*exclusive or*) schreiben wir mit Mustervergleich:

```
xor True  True  = False
xor True  False = True
xor False True  = True
xor False False = False
```

anonyme Parameter

Wird ein Parameter zur Berechnung einer Funktion nicht benötigt,

- so bekommt er keinen Namen (**anonymer Parameter**) und
- seine Stelle wird mit dem Symbol _ markiert.

Mit anonymen Parametern beschreiben wir die restlichen Fälle kürzer:

```
xor True  False = True
xor False True  = True
xor _     _     = False
```

Wächter

Für eine Fallunterscheidung innerhalb der Funktionsdefinition kann man in Haskell **Wächter** (*guard*) benutzen, die durch | eingeleitet werden.

`otherwise` (als syntaktischer Zucker für `True`) beschreibt den Restfall.

```
xor x y                                  -- kein = vor dem ersten Wächter!
    | x && not y = True
    | not x && y = True
    | otherwise  = False
```

if-then-else-Ausdruck

Ein `if-then-else`-Ausdruck ist nicht so übersichtlich:

```
xor x y =
    if x && not y then True
    else if not x && y then True
    else False
```

Mustervergleich mit Listen

Das Muster kann z.B. aus Listen oder aus Datenkonstruktoren mit Parametern bestehen.

Beispiel

Bei Listenoperationen zeigen sich die Vorteile rekursiver Definitionen besonders gut, z.B. um die Länge einer Liste zu bestimmen.

```
length [ ]     = 0                    length :: [a] -> Int
length (_:xs) = 1 + length xs
```

Beispiel

Die Funktion `zip` macht aus zwei Listen eine einzige, indem sie aus den beiden Listenelementen auf gleicher Position ein Paar macht und partnerlose Elemente ignoriert.

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _           = [ ]
```

Mustervergleich mit Datenkonstruktoren

Beispiel (Mustervergleich, der die leere Liste ausschließt)

`head` und `tail` sind –nicht für leere Listen– definiert als

```
head (x:_) = x          tail (_:xs) = xs
```

Beispiel (Summe der Knotenmarkierungen eines Baums)

Hier wird unterschieden, ob der Wert des Parameters durch den Datenkonstruktor `Empty` oder `Branch` erzeugt wurde:

```
sum Empty          = 0
sum (Branch l n r) = n + sum l + sum r
                                  -- Klammern, weil ein Parameter!
```

Haskell erlaubt nur **lineare Muster**, d.h. jede Variable darf nur einmal im Muster vorkommen.

Präfix- und Infixnotation bei Funktionen

Ein Funktionsname, der nur aus Sonderzeichen besteht, heißt **Operator**.

Operatoren werden standardmäßig in Infixnotation verwendet, andere Funktionen in Präfixnotation.

- Um einen Operator in Präfixnotation verwenden zu können, wird das Operationszeichen in runde Klammern eingeschlossen:
  aus `3+4` wird `(+) 3 4` .
- Um eine binäre Funktion in Infixnotation zu verwenden, schließt man den Funktionsnamen mit accent grave (*backtick*) ein:
  aus `xor a b` wird `a `xor` b` .

Typvariablen

Den Typ eines Namens oder die **Signatur** einer Funktion kann Haskell selbst ermitteln; sie kann mit `:t` bzw. `:type` abgefragt werden.

Auf die Frage `:type dec` hätten wir folgende Signatur erwartet:

```
dec :: Int -> Int
```

Stattdessen erhalten wir (lies `::` als „hat Typ“):

```
dec :: Num a => a -> a
```

Dabei ist `a` eine Typvariable, die hier an die Typklasse `Num` gebunden ist.

Eine **Typvariable** steht für einen beliebigen Typ; allerdings müssen alle Vorkommen einer Typvariablen durch denselben Typ ersetzt werden.

Bei der Funktion `dec` muss also der Typ des Parameters mit dem Typ des Ergebnisses übereinstimmen.

Typvariablen beginnen mit Kleinbuchstaben und sind stets implizit allquantifiziert.

Typklassen

Eine **Typklasse** beschreibt eine Menge von Datentypen; zur Klasse `Num` gehören die nummerischen Typen `Int`, `Integer`, `Float` und `Double`.

Für jeden Datentyp dieser Typklasse ist die Subtraktion definiert, so dass mehrere Signaturen möglich sind: Die Funktion ist **überlagert**.

- Die Typklasse `Show` enthält die Typen, deren Elemente anzeigbar sind.
- Die Typklasse `Eq` enthält alle Typen, deren Elemente man auf (Un-)Gleichheit testen kann.
- Die Typklasse `Ord` enthält die Typen, deren Elemente zusätzlich total geordnet sind (<).

Typklassen

Um den Knotentyp für binäre Bäume frei wählen zu können, definieren wir einen **polymorphen Typ** mit einer Typvariable a:
`data Tree a = Empty | Branch (Tree a) a (Tree a)`

Dann ist `Branch (Branch Empty 5 Empty) 2 Empty`
ein Wert des Typs `Tree Int`,
aber `Branch (Branch Empty False Empty) 2 Empty`
ist nicht vom Typ `Tree`,
da ein Knoten vom Typ `Int` ist, ein anderer vom Typ `Bool`.

Hier sind Typen über gemeinsame Typvariablen gekoppelt.

Lokale Definitionen für eine Funktion

Hilfsfunktion oder gemeinsame Teilausdrücke nach Schlüsselwort `where`

Beispiel: Eine quadratische Gleichung $ax^2 + bx + c = 0$ hat keine, eine oder zwei Lösungen ($x = -\frac{b}{2a} \pm \frac{\sqrt{b^2-4ac}}{2a}$). Da der `then` und `else`-Teil vom gleichen Typ sein müssen, wird eine *Liste* der Lösungen erzeugt:

```
quadeq a b c =
    if diskrim == 0.0 then [mid]
       else if diskrim > 0.0 then [mid+radix, mid-radix]
       else [ ]
    where diskrim = b*b - 4.0*a*c
          radix   = (sqrt diskrim)/(2.0*a)
          mid     = -b/(2.0*a)
```

```
> quadeq 1.0 4.0 4.0
[-2.0]
```

```
> quadeq 1.0 (-6.0) 8.0
[4.0, 2.0]
```

```
> quadeq 1.0 0.0 4.0
[ ]
```

Lokale Definitionen für einen Ausdruck

Nach dem `let` werden Namen für z.B. lokale Funktionen oder gemeinsame Teilausdrücke definiert, die dann im Ausdruck nach dem Schlüsselwort `in` verwendet werden können.

```
quadeq' a b c =
    let diskrim = b*b - 4.0*a*c
        radix   = (sqrt diskrim)/(2.0*a)
        mid     = -b/(2.0*a)
    in  if diskrim == 0.0 then [mid]
        else if diskrim > 0.0 then [mid+radix, mid-radix]
        else [ ]
```

Die Definitionen werden nicht in der notierten Reihenfolge ausgeführt, sondern bei Bedarf. Die Interpretation beginnt hier also mit dem `if`.

Bildung von Listen durch Eigenschaften

Listen können kompakt durch charakterisierende Eigenschaften (*list comprehension*) beschrieben werden.
Die Syntax hat Analogien zur mathematischen Beschreibung von Mengen.

Für die ersten geraden Quadratzahlen schreibt man

statt $\{x^2 \mid x \in \{1,2,3,4,5\} \ \wedge \ x \text{ ist gerade Zahl}\}$

in Haskell: `[x*x | x <- [1,2,3,4,5], mod x 2 == 0]`

Auf **Generatoren** für Grundmengenlisten folgen Bedingungen als **Filter**:

```
[x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50, x*y < 100]
```

liefert `[55,80]`

Achtung: Die Reihenfolge der Generatoren spielt eine Rolle:

`[(a,b) | a<-[1..3], b<-[1..2]]` ergibt

`[(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]`,

`[(a,b) | b<-[1..2], a<-[1..3]]` liefert

`[(1,1), (2,1), (3,1), (1,2), (2,2), (3,2)]`.

Bildung von Listen durch Eigenschaften

Beispiel

Quicksort lässt damit kurz und übersichtlich formulieren:

```
quicksort([ ]) = [ ]
quicksort(a:x) = quicksort([b|b <- x, b<=a]) ++ [a] ++
                 quicksort([b|b <- x, b>a])
```

Die leere Liste bleibt unverändert;
das erste Element einer nicht-leeren Liste dient als Pivot-Element.
Die restlichen Elemente werden der Größe nach in zwei Listen aufgeteilt;
diese *kürzeren* Listen werden ihrerseits sortiert und
schließlich werden die sortierten Listen konkateniert.

Bildung von Listen durch arithmetische Folgen

Eine Liste `[a, b .. c]` lässt sich festlegen durch
den **Startwert** `a`, die **Schrittweite** `b-a` und die **Obergrenze** `c`.
Fehlt `b`, so wird die Schrittweite 1 benutzt.

Die Obergrenze ist nicht unbedingt das letzte Element der Liste:
Für ganze Zahlen kommen alle Elemente $x \leq$ `c` in der Folge vor.
Warnung: Für Gleitkommazahlen gilt: $x \leq \texttt{c} + \frac{\texttt{b}-\texttt{a}}{2}$ ☹.

```
> [2,4..8]    > [2,4..9]    > [0,-1..-4]       > [4.1..6.5]     > [4.1..6.7]
[2,4,6,8]     [2,4,6,8]     [0,-1,-2,-3,-4]    [4.1,5.1,6.1]    [4.1,5.1,6.1,7.1]
```

Beispiel (Pythagoräische Tripel)

Tripel (x,y,z) aus natürlichen Zahlen $x < y < z$ mit $x^2 + y^2 = z^2$.

```
pT n = [(x,y,z) | x<-[1..n-2], y<-[x+1..n-1], z<-[y+1..n],
          x*x+y*y==z*z]
```

Bildung von Listen durch arithmetische Folgen

Ohne Obergrenze wird eine **unendlich lange Liste** definiert.
Durch die Bedarfsauswertung berechnet Haskell stets nur den benötigten Teil der Liste.

Beispiel (Liste aller Quadratzahlen)

```
[n*n | n <- [0..] ]
```

Bildung von Listen durch arithmetische Folgen

Beispiel (Liste aller Primzahlen durch das Sieb des Eratosthenes)

```
primzahlen = sieb [2..]
    where sieb (p:x) = p : sieb [n | n <- x, n `mod` p > 0]
```

Die erste Zahl `p` der Liste kommt als Primzahl in die Ergebnisliste und alle Vielfachen von `p` werden aus der Liste entfernt.

Die Funktion `take` liefert die ersten *n* Elemente einer Liste,
die Funktion `drop` die restlichen Elemente:

```
take _ [ ]     = [ ]              drop 0 xs      = xs
take 0 _       = [ ]              drop _ [ ]     = [ ]
take n (x:xs) = x:take (n-1) xs   drop n (_:xs) = drop (n-1) xs
```

Die ersten 10 Primzahlen erhält man also durch

```
> take 10 (let sieb (p:x) = p : sieb [n | n<-x, mod n p > 0]
               in sieb [2..])
[2,3,5,7,11,13,17,19,23,29]
```

Funktionale: Funktionen höherer Stufe

In Funktionalen kommen Funktionen vor auch

- als Parameter,
- als Ergebnis oder
- als Teil einer Datenstruktur.

Viele Algorithmen der Funktionalen Programmierung basieren auf dem **Map-Filter-Reduce**-Prinzip.
In Haskell sind entsprechende Funktionale vordefiniert.

Häufig benutzte Funktionale: map und filter

Das Funktional `map` wendet die Funktion `f` auf jedes Element einer Liste an:

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (x:xs) = f x : map f xs
```

`map f [`x_1`, `x_2`, ..., `x_n`]` liefert also `[f `x_1`, f `x_2`, ..., f `x_n`]`.

Das Funktional `filter` belässt nur die Elemente in einer Liste, die das Prädikat `p` erfüllen:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [ ] = [ ]
filter p (x:xs) | p x       = x : rest
                | otherwise = rest
                where rest = filter p xs
```

Häufig benutzte Funktionale: reduce (foldr1, foldl1)

Das Funktional `reduce`, das alle Elemente einer Liste mit Hilfe einer binären Funktion `f` zusammenfasst, gibt es in verschiedenen Versionen.

`foldr1` wertet die Listenelemente *von rechts nach links* aus:

```
foldr1 :: ((a, a) -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f [x:xs] = f x (foldr1 f xs)
```

Aggregationsfunktionen `f` sind z.B. Summe, Produkt oder Maximum.

```
foldr1 (+) [a,b,c,d,e] = a + (b + (c + (d + e)))
```

Die Reduce-Operation ist für die leere Liste `[ ]` zunächst nicht definiert. Meist nimmt man das neutrale Element der Operation `f` als Ergebnis.

Die Auswertung *von links nach rechts* erledigt ein Funktional `foldl1`.

```
foldl1 (-) [a,b,c,d,e] = (((a - b) - c) - d) - e
```