Haskell Cheat Sheet

Funktion-Definition

type declaration: `square :: Integer -> Integer`
syntax: ⟨*name*⟩ `::` ⟨*type*⟩
defining equation: `square x = x * x`
syntax: ⟨*name*⟩ ⟨*vars*⟩ = ⟨*exp*⟩
evaluation from left to right
⟨*name*⟩s and ⟨*vars*⟩

Expressions ⟨*exp*⟩

Built-in values: **Integer** (2, -7,...), **Bool** (True, False)
Built-in functions:
Binäre arithmetische Operatoren
Integer → Integer → Integer: + , - , * , 'div'
Binäre Vergleichsoperatoren
Integer → Integer → Bool: < , >, ≤, ≥ , == , / =
Binäre boolesche Operatoren
Bool → Bool → Bool: && , | |
Variablen, benutzerdefinierte Funktionen, Konstanten

Präzedenzen

precedence	operators	associativity
9	!!, .	left(!!), right(.)
8	^, ^^, **	right
7	*, /, `div`	left
6	+, −	left
5	:, ++	right
4	==, /=, <, <=, >, >=	none
3	&&	right
2	||	right
1	>>, >>=	left
0	$, $!, `seq`	right

If-Then-Else

if-then-else :: **Bool → Integer → Integer → Integer**
if a then b else c → if 3 == 2 then 2 + 2 else 4 + 4
Note: if, then, else sind Haskell-Keywords

Reservierte Schlüsselwörter

Die Schlüsselwörter:
case, class, data, default, deriving, do, else, if, import, in, infix, infixl, infixr, instance, let, module, newtype, of, then, type und where
dürfen nicht als Namen für Funktionen oder Parameter verwendet werden.

Parameterübergabe

Eine Funktion muss keine Eingabeparameter haben, aber genau einen Rückgabewert.

```
zwei :: Integer
zwei = 2            → Funktion liefert immer Wert 2
```

Die verschiedenen Parameter sind immer positionstreu, dürfen also nicht vertauscht werden.

```
intbool :: Integer → Bool → Integer
intbool a b = if (b == True) then a else 0
```

ghci> intbool 4 True
ghci> 4

Wild Cards

Wenn ein Parameter für die Berechnung der Funktion nicht benötigt wird, kann das Symbol "_"an dieser Stelle als Platzhalter für den Parameter eingefügt werden.

```
zwei :: Float → Integer
zwei _ = 2
```

Signaturen und Typsicherheit

Die Entwicklung von Haskell-Funktionen ist durch die sog. absolute Typsicherheit geprägt, d.h. Haskell weiß automatisch, welche Signatur dem allgemein gültigen Fall der Eingaben entspricht.

```
increment :: Integer → Integer
increment x = x + 1

decrement x = x - 1
```

ghci> :type increment
ghci> increment :: Integer → Integer
ghci> :type decrement
ghci> decrement :: Num a ⇒ a → a

Num ist hierbei eine sog.**Typklasse**. Zur Klasse **Num** gehören die Datentypen **Int, Integer, Float und Double**. Für jeden dieser Datentypen ist die Addition definiert, daher wird erkannt, dass die Klasse **Num** den allgemeinen Fall darstellt. Die Funktion **decrement** ist somit überladen. Die 4 möglichen Signaturen:

```
decrement :: Int    → Int
decrement :: Integer → Integer
decrement :: Float  → Float
decrement :: Double → Double
```

Pattern Matching

Eine Funktion kann in mehrere Definitionen zerlegt werden, dies nennt man Pattern Matching.

```
xor :: Bool → Bool → Bool
xor  True True   = False
xor  True False  = True
xor  False True  = True
xor  False False = False
```

Hier wird für eine Eingabe von oben beginnend geschaut, welche der Zeilen als erste passt. Um Zeilen zu sparen, kann man hier Wildcards verwenden:

```
xor :: Bool → Bool → Bool
xor  True False = True
xor  False True = True
xor  _ _        = False
```

Pattern Matching mit case

Pattern Matching **ohne case**:

```
abc :: Integer → String
abc  1 = "A"
abc  2 = "B"
abc  3 = "C"
```

Pattern Matching **mit case**:

```
abc :: Integer → String
abc x = case x of
    1 = "A"
    2 = "B"
    3 = "C"
```

Lokale Definitionen mit where

Mithilfe des Schlüsselwortes **where** können Funktionen um so genannte interne Hilfsfunktionen erweitert werden. Hierbei handelt es sich um Funktionen, die nicht nach außen sichtbar sind.

```
decrement :: Integer → Integer
decrement x = increment x - 2
    where
        increment :: Integer → Integer
        increment x = x + 1
```

Guards

Auch bei Fallunterscheidungen mit Guards werden die Fälle von oben nach unten betrachtet und der erste passende Fall genommen. Anders als beim Pattern Matching hat man hier aber die Möglichkeit, beliebig komplizierte Ausdrücke anzugeben. Das Schlüsselwort **otherwise** ist hier ein Anker, der alle Fälle abfängt, die bis dorthin nicht gepasst haben.

```
abc ::  Integer → String
abc x
        | x == 1          = "A"
        | x == 2          = "B"
        | otherwise       = "C"
```

Rekursive Fakultätsfunktion

```
fak :: Integer → Integer
fak 0 = 1
fak x
    | n > 0      = n * fak (n-1)
    | otherwise  = error "Natural numbers only"
```

ghci> **fak 3**
ghci> 6
ghci> **fak 0**
ghci> 1
ghci> **fak (-2)**
ghci> error: Natural numbers only

Das Pattern Matching führt dazu, dass die Funktion sich selbst aufruft, wenn n > 0.
Die rekursive Berechnung erfolgt wie folgt:

```
n > 0 = n * fak (n-1)
fak 3 =
3 * fak 2 =
3 * 2 * fak 1 =
3 * 2 * 1 * 1 =
6
```

Die Fakultätsfunktion in der **Lambda-Notation**:

```
fak :: Integer → Integer
fak =
\n → if n == 0 then 1
else (if n > 0 then n * fak (n-1)
else error "Natural numbers only")
```

Listen

Eine Liste ist eine geordnete Menge von Elementen des gleichen Typs. Listen werden in eckigen Klammern geschrieben und die Elemente innerhalb einer Liste durch ein Komma getrennt.

Eine Liste mit Elementen vom Typ **Integer**:

```
[1,2,3,4,5]
```

Eine Liste definiert wie eine Funktion mit Signatur. Sie ist konstant und liefert nur diese Liste zurück:

```
zahlenListe :: [Integer]
zahlenListe = [1,2,3,4,5]
```

Eine Liste kann auch leer sein und somit keine Elemente enthalten:

```
[ ]
```

Eine Liste von Zeichen:

```
['H', 'A', 'L', 'L', 'O']
```

Es lassen sich auch Listen von Listen definieren, sofern diese Listen ebenfalls den gleichen Typ besitzen:

```
[ [1, 2], [5, -1], [ ] ]
```

Das Kopf-Rest-Prinzip bei Listen

Mit dem (:)-Operator wird eine Liste eines beliebigen Typs in **Kopf (head)** und **Rest (tail)** zerlegt. Der Kopf ist hierbei das erste Element der Liste.

Rückgabe erstes Element einer Liste als Funktion:

```
first :: [a] → a
first (x : _) = x
```

Das Pattern **(x:_)** bedeutet: Nimm das erste Element der Liste, nenne es x und ignoriere den Rest der Liste.
In der Signatur steht nun ein **a** (=Typvariable) anstelle eines Basistypen. Das bedeutet, dass jeder beliebige Datentyp verwendet werden kann. Als Eingabe wird eine Liste dieses Typs verlangt, als Rückgabewert ein einzelnes Element.

Rückgabe des Rests einer Liste als Funktion:

```
rest :: [a] → [a]
rest (_ : xs) = xs
```

Das Pattern **(_:xs)** bedeutet: Ignoriere den Kopf der Liste und nenne den Rest xs.

Vordefinierte Funktionen: **head, tail**

Rekursive Listenfunktionen

Um die beiden umgekehrten Fälle, also das letzte Element und die Liste ohne das letzte Element, in Funktionen ausdrücken zu können, muss auf die Rekursion zurückgegriffen werden.

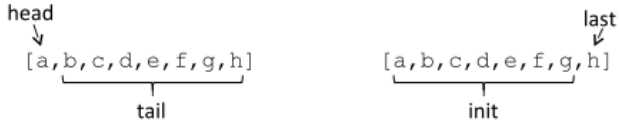
Das letzte Element einer Liste ausgeben:

```
letztes :: [a] → a
letztes [ ]          = error "Empty list"
letztes [ x ]        = x
letztes (_ : xs)     = letztes xs
```

Die Liste ohne das letzte Element ausgeben:

```
ohneLetztes :: [a] → [a]
ohneLetztes [ ]          = error "Empty list"
ohneLetztes [ x ]        = [ ]
ohneLetztes (x : xs)     = x : ohneLetztes xs
```

Die vordefinierten Haskell-Funktionen:




```
letztes :: [a] → a
letztes = last
```

```
ohneLetztes :: [a] → [a]
ohneLetztes = init
```

Überprüfen, ob Element in einer Liste enthalten ist:

```
enthalten :: Eq t ⇒ [t] → t → Bool
ohneLetztes = init
```

Das i-te Element einer Liste ausgeben:

```
elementAt :: Integer → [a] → a
elementAt  0 (x : _)= x
elementAt  i (_ : xs)= elementAt (i - 1) xs
```

Zusammenfassen von Listen

Zwei Listen, die den gleichen Typ aufweisen, können mittels des **(++)-Operators** zu einer Liste zusammengefasst werden. Dabei stehen die Elemente der ersten Liste vor denen der zweiten.

```
konkatenieren :: [a ] → [a] → [ a ]
konkatenieren  [ ] [ ] = [ ]
konkatenieren  [ ] ys = ys
konkatenieren  (x : xs) ys = x : konkatenieren xs ys
```

beziehungsweise:

```
konkatenieren  xs ys = xs ++ ys
```

Automatische Erzeugung von Listen

```
[ a | a ← [1,2,3,4,5] ]
```

Hierbei werden alle Elemente a aus der Liste [1,2,3,4,5] von links nach rechts durchgegangen und - sofern keine weiteren Bedinungen gelten - sofort in die neue Liste übernommen.

Eine Liste mit geraden Zahlen:

```
evenNumber:: [Integer]
evenNumber= [ a | a ← [1,2 .. 20], even a]
```

Eine Liste mit ungeraden Zahlen:

```
oddNumber :: [Integer]
oddNumber = [ a | a ← [1,2 .. 20], odd a]
```

Eine Liste mit Primzahlen:

```
prime :: [Integer]
prime = calculatePrime [2,3 .. 100]

calculatePrime :: Integral a ⇒ [a] → [a]
calculatePrime  [ ] = [ ]
calculatePrime  (x : xs) = x : calculatePrime
[n | n ← xs, n 'mod' x > 0]
```

Listen zerlegen

Vordefinierte Funktionen: **drop, take**

```
takeThree :: [Integer]
takeThree  = take 3 takeThreeList
  where takeThreeList = [1,2,3,4,5]
```

Hier werden mit **take** die ersten 3 Elemente der Liste in eine neue eingefügt.

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

```
dropThree :: [Integer]
dropThree  = drop 3 dropThreeList
  where dropThreeList = [100,200,300,400,500]
```

Hier werden mit **drop** die ersten 3 Elemente der Liste verworfen und die restlichen in eine neue Liste eingefügt

```
ghci> dropThree
ghci> [400,500]
```

```
konkatenieren :: [Integer] → p → [Integer]
konkatenieren  _ _= takeThree ++ dropThree
```

```
ghci> konkatenieren [ ] [ ]
ghci> [1,2,3,400,500]
```

Tupel

Im Gegensatz zu Listen, lassen sich in Tupeln auch unterschiedliche Datentypen zusammenfassen.

```
("Peter", 44)
```

An der ersten Position des zweistelligen Tupels steht hier ein **String** und an der zweiten Position ein **Integer**.

Erstes Element eines Tupels:

```
firstElement :: (a, b)→ a
firstElement  (a, _) = a
```

Zweites Element eines Tupels:

```
secondElement :: (a, b)→ b
secondElement  (_, b) = b
```

Drittes Element im dreistelligen Tupel:

```
secondElement :: (a, b, c)→ c
secondElement  (_, _, c) = c
```

Zweites und drittes Element im dreistelligen Tupel:

```
secondElement :: (a, b, c)→ (b, c)
secondElement  (_, b, c) = (b, c)
```

Zeichenketten

Zeichenketten sind Ketten von Zeichen also Elemente des Datentyps **Char**.

Kleinbuchstaben aus einer Liste entfernen:

```
removeNonUppercase :: [Char]→ [Char]
removeNonUppercase  x = [ c | c ← x,
c 'elem' ['A' .. 'Z']]
```

Großbuchstaben aus einer Liste entfernen:

```
removeUppercase :: [Char]→ [Char]
removeUppercase  x = [ c | c ← x,
c 'elem' ['a' .. 'z']]
```

Kleinbuchstaben in Großbuchstaben umwandeln:

```
downUp :: Char → Char
downUp  x
  | fromEnum x ≥ 97 && fromEnum x ≤ 122
    = toEnum (fromEnum x - 32) :: Char
  | otherwise = x
```

Großbuchstaben in Kleinbuchstaben umwandeln:

```
upDown :: Char → Char
upDown  x
  | fromEnum x ≥ 65 && fromEnum x ≤ 90
    = toEnum (fromEnum x + 32) :: Char
  | otherwise = x
```

Weitere Char-Funktionen

upDown und downUp kombiniert:

```
combined :: Char → Char
combined  x
  | fromEnum x ≥ 65 && fromEnum x ≤ 90
= upDown x
  | otherwise = downUp x
```

```
ghci> map combined "Hallo"
ghci> "hALLO"
```

Buchstaben in Zahlen umwandeln:

```
pos :: Char → Int
pos  c = foldr (\ (m, n, o) acc → c == n | | c == o
then m else acc)
0 (zip3 [1 .. 26] ['A' .. 'Z'] ['a' .. 'z'])

posGen :: Char → Int
posGen  c
  | c == 'Ä' || c == 'ä' = 1
  | c == 'Ö' || c == 'ö' = 15
  | c == 'Ü' || c == 'ü' = 21
  | c == 'ß' = 19
  | otherwise = pos c
```

Mapping

```
mapping :: [Integer] → [Integer]
mapping = map helper
  where helper x = x * 5
```

Die Funktion multipliziert mittels map jedes Listenelement mit 5.

```
ghci> mapping [2,3,4]
ghci> [10,15,20]
```

Faltung

Bei einer Faltung werden die Elemente einer Liste mit Hilfe eines Operators zusammengefasst.

Summe aller Elemente einer Liste:

```
summe :: Num a ⇒ [a] → a
summe [ ]= 0
summe (x : xs)= x + summe xs
```

Produkt aller Elemente einer Liste:

```
produkt :: Num a ⇒ [a] → a
produkt [ ]= 1
produkt (x : xs)= x * produkt xs
```

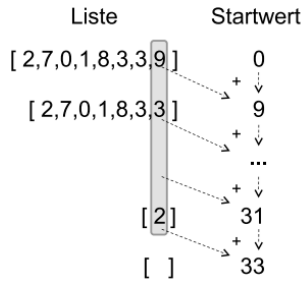
Faltung von rechts mit Startwert

Von rechts beginnend mit einem Startwert **s** und einer zweistelligen Funktion **f** auf eine Liste $[x_1, x_2, ..., x_n]$ ist die Faltung von rechts wie folgt definiert:

$$f x_1\ (f x_2 (...(f x_n s)))$$

foldr:

```
rechtsFalten :: Integer
rechtsFalten  = foldr (+) 0 [2,7,0,1,8,3,3,9]
```



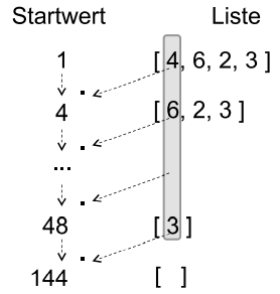
Faltung von links mit Startwert

Von links beginnend mit einem Startwert **s** und einer zweistelligen Funktion **f** auf eine Liste $[x_1, x_2, ..., x_n]$ ist die Faltung von links wie folgt definiert:

$$f(...(f(f(f s x_1) x_2) x_3)...) x_n$$

foldl:

```
linksFalten :: Integer
linksFalten  = foldl (*) 1 [4,6,2,3]
```



Unterschiede foldr und foldl:

```
foldr (\ elem acc → <term>) <start_acc> <list>
foldl (\ acc elem → <term>) <start_acc> <list>
```

Einfache algebraische Typen mit data

Ganz neue Datentypen werden durch das Schlüsselwort **data** erzeugt.

```
data Einfach = Wert deriving Show
```

Einfach ist nun ein neuer Datentyp, den man für die Definition von Funktionen verwenden kann.

```
funktion :: Einfach → String
funktion  Wert = "Das war einfach"
```

```
ghci> funktion Wert
ghci> "Das war einfach"
```

Soll ein Datentyp mehrere Werte haben, werden diese durch einen senkrechten Strich getrennt:

```
data Wochenende =
   Samstag | Sonntag deriving Show

funktion :: Wochenende → Int
funktion  Samstag = 6
funktion  Sonntag = 7
```

Datentyp Tupel

Es gibt hier nur einen Datenkonstruktor, nennen wir ihn **T**, hinter dem zwei Elemente von verschiedenen Typen (=Typvariablen) stehen können.

```
data Tupel a b = T a b
```

```
myFst :: Tupel a b → a
myFst  (T a _) = a
```

```
mySnd :: Tupel a b → b
mySnd  (T _ b) = b
```

```
ghci> myFst (T 2 3)
ghci> 2

ghci> mySnd (T 2 3)
ghci> 3
```

Tupel sind tatsächlich so in Haskell definiert.
Die Schreibweise mit den Klammern ist bloß syntaktischer Zucker, um die Lesbarkeit zu erhöhen.

Datentyp Maybe

Angenommen, beim Durchsuchen eines Telefonbuchs kann der gewünschte Eintrag nicht gefunden werden. In diesem Fall soll ein Sonderwert zurückgegeben werden, der einen Fehlschlag signalisiert. Für diesen Zweck gibt es den Datentyp **Maybe**.

```
teilen :: Int → Int → Maybe Int
teilen  x 0 = Nothing
teilen  x y = Just (div x y)
```

```
ghci> teilen 4 2
ghci> Just 2

ghci> teilen 4 0
ghci> Nothing
```

Überprüfung, ob Nothing zurückgegeben wurde. Falls ja, soll ein entsprechender String zurückgegeben werden:

```
teilenMöglich :: Int → Int → String
teilenMöglich  x y = case teilen x y of
   Just _ → "Division erfolgreich"
   Nothing → "Division nicht möglich"
```

Datentyp mit mehreren Feldern

Sollte der Fall eintreten, dass Datenkonstruktoren mit mehreren Werten benötigt werden, kann es unschön sein, diese mit Pattern Matching zu verarbeiten.

Vorher:

```
data Eintrag = Eintrag String String String
```

Nachher:

```
data Eintrag = Eintrag {
   a,
   b,
   c :: String}
```

Typklasse Eq

Operatoren:

(==) und (/=)

```
equalities :: Eq a ⇒ a → a → a
equalities  x y = if a == b then a else b
```

Typklasse Show

Die Typklasse **Show** konvertiert mit der Funktion **show** Elemente zu Strings, um sie ausgeben zu können.

```
data Size = Small |Medium |Large deriving Show

s1,s2,s3 :: Size
s1 = Small
s2 = Medium
s3 = Large

ghci> (s1,s2,s3)
ghci> (Small, Medium, Large)
```

oder:

```
showNumber :: [Char]
showNumber  = "Du siehst eine "++ show x
   where x = 2

ghci> showNumber
ghci> "Du siehst eine 2"
```

Weitere nützliche Funktionen

1 Überprüfung, ob eine Liste ein Palindrom ist:

```
isPalindrome :: Eq a ⇒ [a] → Bool
isPalindrome xs = xs == reverse xs
```

2 Den größten Primfakor einer Zahl ermitteln:

```
sieb :: Integral a ⇒ [a] →  [a]
sieb [ ] = [ ]
sieb (x : xs) = x : sieb [ n | n ← xs, n 'mod' x > 0 ]

primes ::  [Integer]
primes  = sieb [1,2 .. 100 ]

primeFactors :: Integer →  Integer
primeFactors x =
   last [ n | n ← primes, x 'mod' n == 0 ]
```

3 Minimum einer Liste:

```
mm :: Ord a ⇒ [a] → a
mm [ x ] = x
mm (a : b : xs) = mm (min a b : xs)
```

IO

1 Kompilieren:

```
G:\...> ghc - -make dateiname.hs
G:\...> ./dateiname
```

Programme müssen mit GHC kompiliert werden, nicht mit GHCI.

2 Funktionen:

putStrLn : Die Funktion gibt einen String inkl. Zeilenausbruch aus.
putChar : Die Funktion gibt ein einzelnes Zeichen aus.
print : Die Funktion wandelt jeden Wert automatisch in einen String um.
getLine : Die Funktion liest eine Zeile ein.
getLine : Die Funktion liest ein einzelnes Zeichen ein.

3 Anwendungsbeispiel:

```
downUp :: Char → Char
downUp  x
   | fromEnum x ≥ 97 && fromEnum x ≤ 122
     = toEnum (fromEnum x - 32) :: Char
   | otherwise = x

main :: IO()
main = do
   putStrLn "What's your name?"
   name ← getLine
   let answer = map downUp name
   if name =/ "Georg Moser"then do
      putStrLn ("Input: "++ name)
      main
   else
      putStrLn $
        "Hello"++ answer
```

```
> What's your name?
> Rene Thiemann    (=manuelle Eingabe)
> Input: Rene Thiemann
> What's your name?
> Georg Moser    (=manuelle Eingabe)
> Hello GEORG MOSER
```

Typklasse Ord

(<) , (≤) , (>) und (≥)

```
greater :: Ord a ⇒ a → a → a
greater  x y = if a > b then a else b
```

Tipps

1

Die äquivalente Version des folgenden Codes:

```
function :: Bool → Bool → Bool
function x y
      | (x == True) && (y == False)    = True
      | otherwise                      = False
```

Ist diese:

```
function :: Bool → Bool → Bool
function x y
      | x && not y                     = True
      | otherwise                      = False
```

2

Eine Liste von Zeichen kann auch vereinfacht in der Notation von **Zeichenketten = Strings** angegeben

```
ghci> "HALLO"== ['H', 'A', 'L', 'L', 'O']
ghci> True
```

3

Automatische Erzeugung von Listen - Beispiele:

```
ghci> [2,4 .. 10]
ghci> [2,4,6,8,10]

ghci> [2,4 .. 11]
ghci> [2,4,6,8,10]

ghci> [0, -1 .. -5]
ghci> [0, -1, -2, -3, -4, -5]

ghci> [3.1 .. 6.5]
ghci> [3.1, 4.1, 5.1, 6.1]
```

4

Weitere äquivalente Notationsformen:

```
ghci> f(g(x))
ghci> (f.g) x
ghci> f.g $ x
ghci> f $ g $ x
```