Basiskonstrukte von Haskell

PD Dr. David Sabel

Goethe-Universität Frankfurt am Main

29. September 2015

Basistypen und Operationen

- Ganzzahlen:
  - `Int` = Ganzzahlen beschränkter Länge
  - `Integer` = Ganzzahlen beliebiger Länge
- Gleitkommazahlen: Darstellung mit Punkt, z.B. `10.25`
  - `Float` = Fließkommazahlen
  - `Double` = Fließkommazahlen mit doppelter Genauigkeit
- Arithmetische Operationen:
  - Addition (+), Subtraktion (-), Multiplikation (*), Division (/)
  - Ganzzahlige Division: `mod`, `div`, `divMod`

Wahrheitswerte: Der Datentyp Bool

Werte (Datenkonstruktoren) vom Typ `Bool`: `True` und `False`

Basisoperationen (Funktionen):

- Logische Negation: `not`
- Logisches Und: `a && b`
- Logisches Oder: `a || b`

Arithmetische Vergleiche:

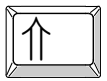
- Gleichheit (==), Ungleichheit (/=), größer (>), größer oder gleich (>=), kleiner (<), kleiner oder gleich (<=)

Zeichen und Zeichenketten

- Der Typ `Char` repräsentiert Zeichen
  Darstellung: Zeichen in einfache Anführungszeichen, z.B. `'A'`
- Zeichenketten: Typ `String`
  Dastellung in doppelten Anfühungszeichen, z.B. `"Hallo"`.
- Genauer ist der Typ `String` gleich zu `[Char]`
  d.h. eine Liste von Zeichen

Infix- und Präfixoperatoren

- Präfix-Operatoren infix verwenden:

  In Hochkommata setzen ( + )

  z.B. `10 ‘mod‘ 3`

- Infix-Operatoren präfix verwenden:

  In runde Klammern setzen

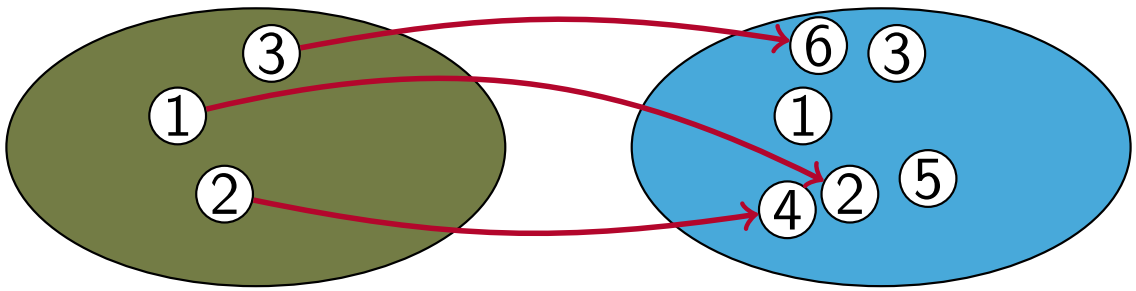
  z.B. `(+) 5 6`

Funktionen

Definition (mathematisch):
Seien D und Z Mengen. Eine **Funktion**

$$f : D \to Z$$

ordnet jedem Element $x \in D$ ein Element $y \in Z$ zu.
D nennt man den **Definitionsbereich** und Z den **Zielbereich**.



Funktionen

Definition (mathematisch):
Seien D und Z Mengen. Eine **Funktion**

$$f : D \to Z$$

ordnet jedem Element $x \in D$ ein Element $y \in Z$ zu.
D nennt man den **Definitionsbereich** und Z den **Zielbereich**.

Beispiel:

$$\mathit{double} : \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} \to \underbrace{\mathbb{Z}}_{\text{Zielbereich}}$$
$$\mathit{double}(x) = x + x$$

Funktionen

Definition (mathematisch):
Seien D und Z Mengen. Eine **Funktion**

$$f : D \to Z$$

ordnet jedem Element $x \in D$ ein Element $y \in Z$ zu.
D nennt man den **Definitionsbereich** und Z den **Zielbereich**.

Beispiel:

$$\begin{aligned} &double : \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} \to \underbrace{\mathbb{Z}}_{\text{Zielbereich}} \\ &double(x) = x + x \end{aligned}$$

In Haskell, Funktionstypen: Typ_1 -> Typ_2

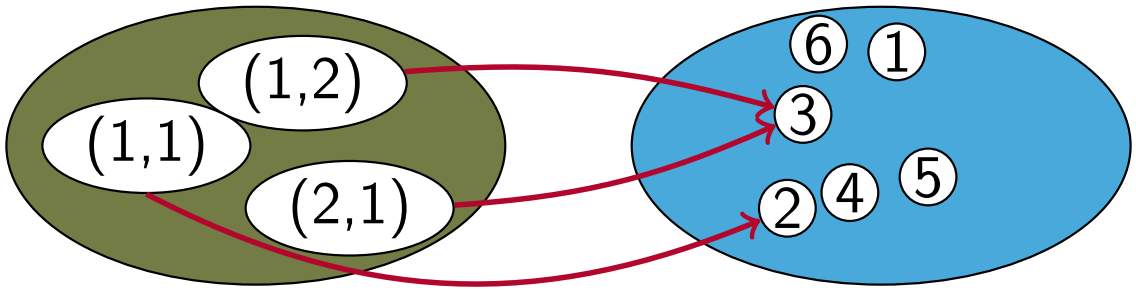
```
double :: Integer -> Integer
double x = x + x
```

Mehrstellige Funktionen

Möglichkeit 1: Definitionsbereich ist eine Menge von Tupeln

$$add : \underbrace{(\mathbb{Z} \times \mathbb{Z})}_{\text{Definitionsbereich}} \rightarrow \underbrace{\mathbb{Z}}_{\text{Zielbereich}}$$

$$add(x, y) = x + y$$



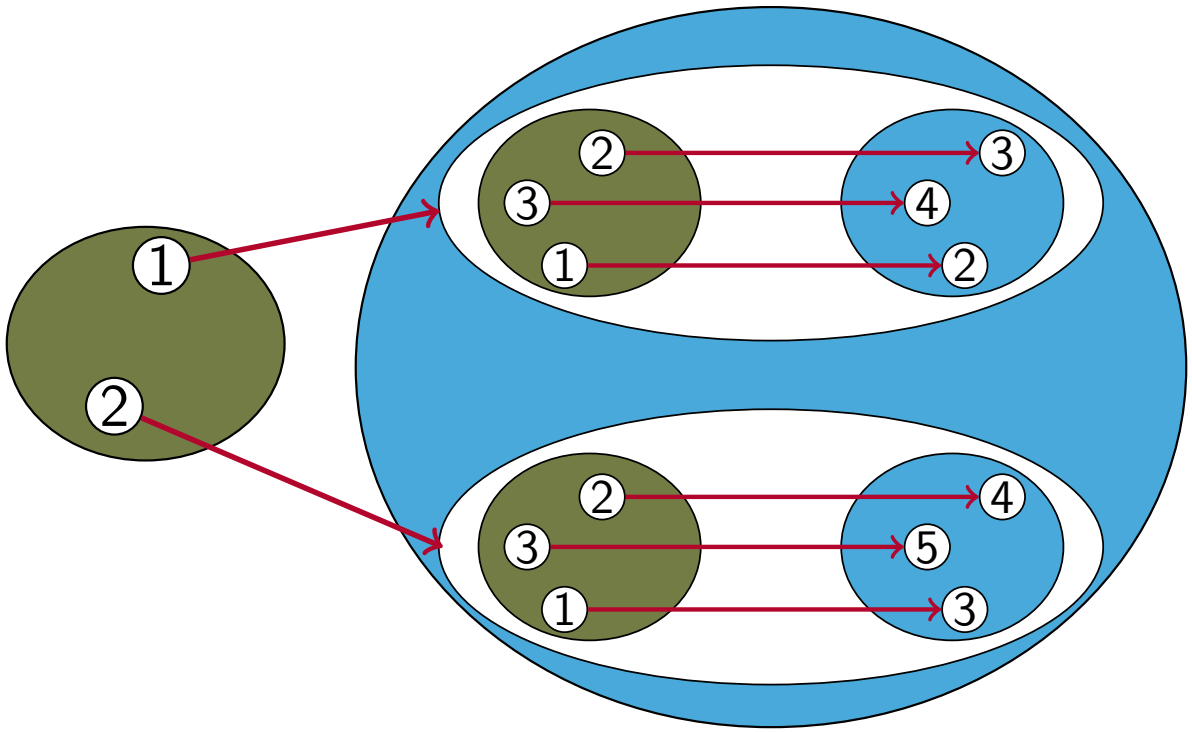
Verwendung: $add(1, 2)$

Mehrstellige Funktionen: Currying

Möglichkeit 2: (Currying) Statt Funktion mit n-Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$add : \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}}$$

$$add\ x\ y = x + y$$



Mehrstellige Funktionen (3)

Möglichkeit 2: (Currying) Statt Funktion mit n-Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$add : \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}}$$
$$add\ x\ y = x + y$$

- Vorteil ggü. Variante 1:
  Man kann partiell anwenden, z.B. *add* 2
- Variante 2 wird in Haskell fast immer verwendet!

```
add :: Integer -> (Integer -> Integer)
add x y = x + y
```

Mehrstellige Funktionen (3)

Möglichkeit 2: (Currying) Statt Funktion mit n-Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$add : \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}}$$
$$add\ x\ y = x + y$$

- Vorteil ggü. Variante 1:
  Man kann partiell anwenden, z.B. *add* 2
- Variante 2 wird in Haskell fast immer verwendet!

```
add :: Integer -> (Integer -> Integer)
add x y = x + y
```

-> in Haskell ist rechts-assoziativ: `a -> b -> c  =  a -> (b -> c)`
Daher kann man Klammern weglassen.

Mehrstellige Funktionen (3)

Möglichkeit 2: (Currying) Statt Funktion mit n-Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$add : \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}}$$
$$add\ x\ y = x + y$$

- Vorteil ggü. Variante 1:
  Man kann partiell anwenden, z.B. *add* 2
- Variante 2 wird in Haskell fast immer verwendet!

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

`->` in Haskell ist rechts-assoziativ: `a -> b -> c  =  a -> (b -> c)`
Daher kann man Klammern weglassen.

Funktionstypen

Einfacher: f erwartet n Eingaben, dann ist der Typ von f:

$$f :: \underbrace{\text{Typ}_1}_{\substack{\text{Typ des}\\ 1.\ \text{Arguments}}} \texttt{->} \underbrace{\text{Typ}_2}_{\substack{\text{Typ des}\\ 2.\ \text{Arguments}}} \texttt{->} \ldots \texttt{->} \underbrace{\text{Typ}_n}_{\substack{\text{Typ des}\\ n.\ \text{Arguments}}} \texttt{->} \underbrace{\text{Typ}_{n+1}}_{\substack{\text{Typ des}\\ \text{Ergebnisses}}}$$

- `->` in Funktionstypen ist rechts-geklammert
- `(&&) :: Bool -> Bool -> Bool`
  entspricht `(&&) :: Bool -> (Bool -> Bool)`
  und nicht `(&&) :: (Bool -> Bool) -> Bool`

Funktionen: Beispiele

Beispiel: `mod` und `div`

- `mod`: Rest einer Division mit Rest
- `div`: Ganzzahliger Anteil der Division mit Rest

Typen von `mod` und `div`:

```
mod :: Integer -> Integer -> Integer
div :: Integer -> Integer -> Integer
```

In Wirklichkeit:

```
Prelude> :type mod ⏎
mod :: (Integral a) => a -> a -> a
Prelude> :type div ⏎
div :: (Integral a) => a -> a -> a
```

In etwa: Für alle Typen `a` die `Integral`-Typen sind,
hat `mod` den Typ `a -> a -> a`

Weitere Funktionen und ihre Typen

```
Prelude> :type (==) ↵
(==) :: (Eq a) => a -> a -> Bool
Prelude> :type (<) ↵
(<) :: (Ord a) => a -> a -> Bool
Prelude> :type (+) ↵
(+) :: (Num a) => a -> a -> a
Prelude> :type (/) ↵
(/) :: Fractional a => a -> a -> a
```

Typgerecht programmieren

Die Typen müssen stets passen, sonst gibt's einen Typfehler:

```
Prelude> :type not
not :: Bool -> Bool
Prelude> not 'C'
<interactive>:1:4:
  Couldn't match expected type 'Bool' against inferred type 'Char'
  In the first argument of 'not', namely 'C'
  In the expression: not 'C'
  In the definition of 'it': it = not 'C'
```

(Unerwartete) Fehlermeldung:

```
Prelude> not 5
<interactive>:1:4:
    No instance for (Num Bool)
      arising from the literal '5' at <interactive>:1:4
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of 'not', namely '5'
    In the expression: not 5
    In the definition of 'it': it = not 5
```

Funktionen selbst definieren

```
double x = x + x
```

Funktionen selbst definieren

```
double x = x + x
```

Allgemein:

$$\textit{funktion\_Name} \quad \textit{par}_1 \quad \ldots \quad \textit{par}_n \quad = \quad \textit{Haskell\_Ausdruck}$$

wobei

- $\textit{par}_i$: Formale Parameter, z.B. Variablen `x`, `y`, ...
- Die $\textit{par}_i$ dürfen rechts im *Haskell_Ausdruck* verwendet werden
- *funktion_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

Funktionen selbst definieren

```
double :: Integer -> Integer
double x = x + x
```

Allgemein:

$$\mathit{funktion\_Name} \quad \mathit{par}_1 \quad \ldots \quad \mathit{par}_n \quad = \quad \mathit{Haskell\_Ausdruck}$$

wobei

- par_i: Formale Parameter, z.B. Variablen `x`, `y`, ...
- Die par_i dürfen rechts im *Haskell_Ausdruck* verwendet werden
- *funktion_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

Man darf auch den Typ angeben!

Fallunterscheidung: `if-then-else`

Syntax: `if` b `then` e_1 `else` e_2

Beispiel:

```
doubleEven :: Integer -> Integer
doubleEven x = if even x then double x else x
```

`even` testet, ob eine Zahl gerade ist:

```
even x = x `mod` 2 == 0
```

```
*Main> :reload ⏎
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )
Ok, modules loaded: Main.
*Main> doubleEven 50 ⏎
100
*Main> doubleEven 17 ⏎
17
```

Guards

Guards (Wächter): Boolesche Ausdrücke,
die die Definition der Funktion festlegen

```
f x1 ... xm
  | 1. Guard  = e1
  ...
  | n. Guard  = en
```

- Abarbeitung von oben nach unten
- Erster Guard, der zu `True` auswertet, bestimmt die Definition.

```
doubleEven a
  | even a    = double a
  | otherwise = a
```

Vordefiniert: `otherwise = True`

Einstiegsaufgabe zu Funktionen

Aufgabe

Implementiere eine Funktion `max3` in Haskell,
die das Maximum von drei Ganzzahlen berechnet.

Strukturiertes Vorgehen zur Problemlösung:

- Was sind die Eingaben und Ausgaben von `max3`?
- Wie lautet der Typ von `max3` in Haskell?
- Wie findet man das Maximum von drei Zahlen a, b, c?
- Wie implementiert man `max3` in Haskell?

Einstiegsaufgabe zu Funktionen

Aufgabe

Implementiere eine Funktion `max3` in Haskell,
die das Maximum von drei Ganzzahlen berechnet.

Strukturiertes Vorgehen zur Problemlösung:

- Was sind die Eingaben und Ausgaben von `max3`?
  - Eingabe: Drei Zahlen vom Typ `Integer`
  - Ausgabe: Eine Zahl vom Typ `Integer`
- Wie lautet der Typ von `max3` in Haskell?
- Wie findet man das Maximum von drei Zahlen a, b, c?
- Wie implementiert man `max3` in Haskell?

Einstiegsaufgabe zu Funktionen

Aufgabe

Implementiere eine Funktion `max3` in Haskell,
die das Maximum von drei Ganzzahlen berechnet.

Strukturiertes Vorgehen zur Problemlösung:

- Was sind die Eingaben und Ausgaben von `max3`?
  - Eingabe: Drei Zahlen vom Typ `Integer`
  - Ausgabe: Eine Zahl vom Typ `Integer`
- Wie lautet der Typ von `max3` in Haskell?
  ```
  max3 :: Integer -> Integer -> Integer -> Integer
  ```
- Wie findet man das Maximum von drei Zahlen a, b, c?
- Wie implementiert man `max3` in Haskell?

Einstiegsaufgabe zu Funktionen

Aufgabe

Implementiere eine Funktion `max3` in Haskell,
die das Maximum von drei Ganzzahlen berechnet.

Strukturiertes Vorgehen zur Problemlösung:

- Was sind die Eingaben und Ausgaben von `max3`?
  - Eingabe: Drei Zahlen vom Typ `Integer`
  - Ausgabe: Eine Zahl vom Typ `Integer`
- Wie lautet der Typ von `max3` in Haskell?
  ```
  max3 :: Integer -> Integer -> Integer -> Integer
  ```
- Wie findet man das Maximum von drei Zahlen a, b, c?
  Z.B. berechne erst das Maximum von a und b, und anschließend das Maximum des Ergebnisses und c
- Wie implementiert man `max3` in Haskell?

Varianten von max3

```
max3 a b c = if a >= b
               then if a >= c then a else c
               else if b >= c then b else c
```

Mit Guards:

```
max3 a b c
  | a >= b && a >= c = a
  | b >= a && b >= c = b
  | otherwise        = c
```

Mit max2

```
max2 a b
  | a >= b = a
  | otherwise = b

max3 a b c = max2 (max2 a b) c
```

Let-Ausdrücke: Lokale Definitionen

$$
\begin{array}{lll}
\texttt{let} & f_1\ x_{1,1}\ \ldots\ x_{n,1} & = \quad \mathit{Ausdruck}_1 \\
 & f_2\ x_{1,2}\ \ldots\ x_{n,2} & = \quad \mathit{Ausdruck}_2 \\
 & \ldots & \\
 & f_m\ x_{1,m}\ \ldots\ x_{n,m} & = \quad \mathit{Ausdruck}_m \\
\texttt{in} & \text{Ausdruck} &
\end{array}
$$

Beispiel:

```
max3 a b c =
    let max2 x y = if x >= y then x else y
        maxab    = max2 a b
    in  max2 maxab c
```

where-Ausdrücke: nachgestellte lokale Definitionen

$$
\begin{array}{l}
\texttt{fun } y_1 \ \ldots \ y_m \texttt{ = } e \\
\quad \texttt{where} \\
\qquad \begin{array}{lcl}
f_1 \ x_{1,1} \ \ldots \ x_{n,1} & = & Ausdruck_1 \\
f_2 \ x_{1,2} \ \ldots \ x_{n,2} & = & Ausdruck_2 \\
\ldots & & \\
f_m \ x_{1,m} \ \ldots \ x_{n,m} & = & Ausdruck_m
\end{array}
\end{array}
$$

Beispiel:

```
max3 a b c = max2 maxab c
 where
   max2 x y
    | x >= y = y
    | otherwise = y
   maxab = max a b
```

Anonyme Funktionen

- Lambda-Ausdrücke $\lambda x.e$ stellen anonyme Funktionen dar
- Analog zu $f\ x\ = e$, nur dass die Funktion keinen Namen hat
- Haskell-Notation `\x -> e`
- Mehrstellige Lambda-Ausdrücke: `\x1 x2 ... xn -> e`

Beispiel: Die Funktion `double` als anonyme Funktion:

```
\x -> x + x
```

Higher-Order Funktionen

- D.h.: **Rückgabewerte** dürfen in Haskell auch Funktionen sein
- Auch Argumente (Eingaben) dürfen Funktionen sein:

```
applyAndAdd f x y = (f x) + (f y)

(.) f g x         = f (g x)
```

```
*Main> applyAndAdd double 10 20 ↩
60
*Main> applyAndAdd (\a -> a^2) 3 4 ↩
25 *Main> (even . double) 3 ↩
True
```

Nochmal Typen

Typ von `applyAndAdd`

```
applyAndAdd :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

```
applyAndAdd f x y = (f x) + (f y)
```

Nochmal Typen

Typ von `applyAndAdd`

```
applyAndAdd :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

- `(Integer -> Integer)`: Typ von `f`
- `Integer`: Typ von `x`
- `Integer`: Typ von `y`
- `Integer`: Typ des Ergebnisses

```
applyAndAdd f x y = (f x) + (f y)
```

Nochmal Typen

Typ von `applyAndAdd`

```
applyAndAdd :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

```
applyAndAdd f x y = (f x) + (f y)
```

Achtung: Im Typ

```
(Integer -> Integer) -> Integer -> Integer -> Integer
```

darf man die Klammern nicht weglassen:

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

denn das entspricht

```
Integer -> (Integer -> (Integer -> (Integer -> Integer))).
```

Polymorphe Typen

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

Polymorphe Typen

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
twice :: (a -> a) -> a -> a
```

Polymorphe Typen

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
twice :: (Int -> Int) -> Int -> Int
```

Polymorphe Typen

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
twice :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
twice :: (a -> a) -> a -> a
```

Polymorphe Typen

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
twice :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
twice :: (Bool -> Bool) -> Bool -> Bool
```

Polymorphe Typen

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
twice :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
twice :: (Bool -> Bool) -> Bool -> Bool
```

z.B. `a = Char -> Char`

```
twice :: (a -> a) -> a -> a
```

Polymorphe Typen

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
twice :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
twice :: (Bool -> Bool) -> Bool -> Bool
```

z.B. `a = Char -> Char`

```
twice::((Char->Char)->(Char->Char))->(Char->Char)->(Char->Char)
```

Rekursion

- **Rekursionsanfang**: Der Fall, für den sich die Funktion nicht mehr selbst aufruft, sondern abbricht
- **Rekursionsschritt**: Der rekursive Aufruf

Beispiel:

```
fakultaet x =
 if x <= 1
  then 1                        -- Rekursionsanfang
  else x*(fakultaet (x-1)) -- Rekursionsschritt
```

Problemlösen mit Rekursion:

- Rekursionsanfang:
  Der einfache Fall, für den man die Lösung direkt kennt.
- Rekursionsschritt:
  Man löst ganz wenig selbst, bis das Problem etwas kleiner ist.
  Das (immer noch große) Restproblem erledigt die Rekursion,

Aufgabe zur Rekursion

Aufgabe

Die Quersumme einer nichtnegativen Ganzahl ist die Summe ihrer Ziffern. Implementiere in Haskell eine rekursive Funktion, welche die Quersumme einer nichtnegativen Ganzzahl berechnet.

- Was ist der Rekursionsanfang?
- Was ist der Rekursionsschritt?
- Welche Teilprobleme sind zu lösen?

Aufgabe zur Rekursion

Aufgabe

Die Quersumme einer nichtnegativen Ganzahl ist die Summe ihrer Ziffern. Implementiere in Haskell eine rekursive Funktion, welche die Quersumme einer nichtnegativen Ganzzahl berechnet.

- Was ist der Rekursionsanfang?
  Die Zahl ist einstellig.
- Was ist der Rekursionsschritt?
  Rekursiv die Quersumme der Zahl ohne letzte Ziffer berechnen
  und die letzte Ziffer hinzuaddieren.
- Welche Teilprobleme sind zu lösen?
  - testen, ob eine Zahl einstellig ist: Division mit Rest durch 10 ergibt 0 als ganzzahligen Anteil
  - letzte Ziffer einer Zahl bestimmen: Rest der Division mit Rest durch 10
  - Zahl ohne letzte Ziffer berechnen: Ganzzahliger Anteil der Division durch 10

Quersumme in Haskell

```
quersumme x =
  if x `div` 10 == 0
   then x `mod` 10  -- Zahl ist einstellig
   else (x `mod` 10) + (quersumme (x `div` 10))
```

Variante mit `divMod` und `let`

```
quersumme x =
   let (d,r) = divMod x 10
   in if d == 0 then r else r + (quersumme d)
```

Pattern matching (auf Zahlen)

N-maliges Verdoppeln:

```
double_n_times :: Integer -> Integer -> Integer
double_n_times x n =
  if n == 0 then x
  else double_n_times (double x) (n-1)
```

Man darf statt Variablen auch Pattern in der Funktionsdefinition verwenden, und mehrere Definitionsgleichungen angeben. Die Pattern werden von oben nach unten abgearbeitet.

```
double_n_times2 :: Integer -> Integer -> Integer
double_n_times2 x 0 = x
double_n_times2 x n = double_n_times2 (double x) (n-1)
```

Falsch:

```
double_n_times2 :: Integer -> Integer -> Integer
double_n_times2 x n = double_n_times2 (double x) (n-1)
double_n_times2 x 0 = x
```

Die Error-Funktion

```
double_n_times3 :: Integer -> Integer -> Integer
double_n_times3 x n
  | n == 0    = x
  | otherwise = double_n_times3 (double x) (n-1)
```

Was passiert bei negativem `n`?

Die Error-Funktion

```
double_n_times3 :: Integer -> Integer -> Integer
double_n_times3 x n
  | n == 0    = x
  | otherwise = double_n_times3 (double x) (n-1)
```

Was passiert bei negativem n?

- `error :: String -> a`

```
double_n_times4 :: Integer -> Integer -> Integer
double_n_times4 x n
  | n < 0     = error "Negatives Verdoppeln verboten!"
  | n == 0    = x
  | otherwise = double_n_times4 (double x) (n-1)
```

```
*Main> double_n_times4 10 (-10)
*** Exception:
 in double_n_times4: negatives Verdoppeln ist verboten
```

Ein- und Ausgabe

- Seiteneffekte liegen außerhalb der reinen funktionalen Welt
- In Haskell: Monadisches IO
- Sehr analog zu imperativem Programmieren
- Typ einer IO-Aktion: `IO a`
  bezeichnet eine Seiteneffekt-behaftet Berechnung mit Rückgabe vom Typ `a`
- IO-Aktion mit Typ `IO ()` entspricht "keiner Rückgabe"
- `return :: a -> IO a`
  liefert IO-Aktion, die das Argument zurückliefert.

Ein- und Ausgabe: do-Notation

Mit `do` kann sequentiell Seiteneffekt-behaftet programmiert werden.
Auf Ergebnisse einer Berechnung kann nur im do-Block mit

```
do
  ...
  result <- monadischeBerechnung
  ...
```

zugegriffen werden.

- es gibt keine Möglichkeit, dies außerhalb zu tun!
- insbesondere keine Funktion `IO a -> a`

Primitive IO-Aktionen

- `getChar :: IO Char`
- `getLine :: IO String`
- `putChar :: Char -> IO ()`
- `putStrLn :: String -> IO ()`

Beispiel: `echo`

```
echo :: IO ()
echo = do
       line <- getLine
       putStrLn line
```

File-IO

Haskell unterstützt lazy-IO

- Datei lazy einlesen: `readFile :: FilePath -> IO String`
- Datei schreiben: `writeFile :: FilePath -> String -> IO ()`

Beispiele:

```
copy :: FilePath -> FilePath -> IO ()
copy from to = do
 content <- readFile from
 writeFile to content

cat :: FilePath -> IO ()
cat file = do
 content <- readFile file
 putStrLn content
```

Bibliotheken

- Haskell hat ein hierarchisches Modul-System
- Bibliotheken (siehe z.B. haskell.org/hackage) liegen in der Hierarchie vor
- Zeichenverarbeitung: `Data.Char`
- Listenprogrammierung: `Data.List`

Einbinden von Bibliotheken

- mit dem Schlüsselwort `import` Modulname
- am Anfang der Quellcodedatei
- Modul laden im GHCi: `:m + Modulname`

Z.B.

```
import Data.Char

copyAndUpper from to = do
 content <- readFile from
 writeFile to (map toUpper content)
```