Programmierung mit Haskell

Vorsemesterkurs
Sommersemester 2024
Ronja Düffel

08. April 2024

Ganze Zahlen: Int und Integer

- Der Typ `Int` umfasst ganze Zahlen beschränkter Größe
- Der Typ `Integer` umfasst ganze Zahlen beliebiger Größe
- Darstellung der Zahlen ist identisch z.B. 1000
- Defaulting: `Integer`, wenn es nötig ist, sonst offenlassen:

```
Prelude> :type 1000 ⏎
1000 :: Num p => p
Prelude> 1000::Integer ⏎
1000
Prelude> 1000::Int ⏎
1000
Prelude> (1000::Integer) + (1000::Int) ⏎
...Couldn't match expected type 'Integer' with actual type 'Int'...
```

- In etwa 1000 ist vom Typ `p`, wenn `p` ein numerischer Typ ist.
- Genauer: `Num p =>` ist eine sog. Typklassenbeschränkung
- `Int` und `Integer` sind numerische Typen (haben Instanzen für `Num`)

Gleitkommazahlen

- Typen `Float` und `Double` (mit doppelter Genauigkeit)
- Kommastelle wird mit `.` (Punkt) dargestellt
- Typklasse dazu: `Fractional`

```
Prelude> :type 10.5 ↩
10.5 :: (Fractional t) => t
Prelude> 10.5::Double ↩
10.5
Prelude> 10.5::Float ↩
10.5 Prelude> 10.5::Integer ↩
...No instance for (Fractional Integer)
```

- Beachte: Das Rechnen mit solchen Kommazahlen ist ungenau!

Zeichen und Zeichenketten

- Der Typ `Char` repräsentiert Zeichen
  Darstellung: Zeichen in einfache Anführungszeichen, z.B. `'A'`
- Spezielle Zeichen (Auswahl):

Darstellung in Haskell	Zeichen, das sich dahinter verbirgt
`'\\'`	Backslash \
`'\''`	einfaches Anführungszeichen '
`'\"'`	doppeltes Anführungszeichen "
`'\n'`	Zeilenumbruch
`'\t'`	Tabulator

- Zeichenketten: Typ `String`
  Darstellung in doppelten Anführungszeichen, z.B. `"Hallo"`.
- Genauer ist der Typ `String` gleich zu `[Char]`
  d.h. eine Liste von Zeichen (Listen behandeln wir später)

Beispiel Zeichenketten

```
> :set +t
> "Ein \n\'mehrzeiliger\'\nText mit \"Anfuehrungszeichen\""
"Ein \n'mehrzeiliger'\nText mit \"Anfuehrungszeichen\""
it :: [Char]
> putStrLn "Ein \n\'mehrzeiliger\'\nText mit \"Anfuehrungszeichen\""
Ein
'mehrzeiliger'
Text mit "Anfuehrungszeichen"
it :: ()
```

Operatoren auf Zahlen

- Operatoren auf Zahlen in Haskell:
  Addition `+`, Substraktion `-`, Multiplikation `*` und Division `/`
- Beispiele: `3 * 6`, `10.0 / 2.5`, `4 + 5 * 4`
- Beim `-` muss man aufpassen, da es auch für negative Zahlen benutzt wird

```
Prelude> 2 * -2 ⏎

<interactive>:1:0:
    Precedence parsing error
        cannot mix ‘*’ [infixl 7] and prefix ‘-’ [infixl 6]
        in the same infix expression
Prelude> 2 * (-2) ⏎
-4
```

Vergleichsoperationen

Gleichheitstest == und
Ungleichheitstest /=

```
Prelude> 1 == 3 ⏎
False
Prelude> 3*10 == 6*5 ⏎
True
Prelude> True == False ⏎
False
Prelude> False == False ⏎
True
Prelude> 2*8 /= 64 ⏎
True
Prelude> 2+8 /= 10 ⏎
False
Prelude> True /= False ⏎
True
```

größer: >, größer oder gleich: >=
kleiner: <, kleiner oder gleich: <=

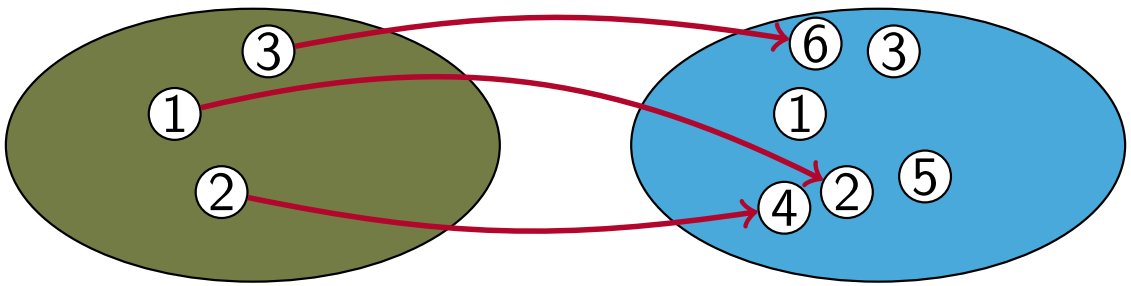
```
Prelude> 5 >= 5 ⏎
True
Prelude> 5 > 5 ⏎
False
Prelude> 6 > 5 ⏎
True
Prelude> 4 < 5 ⏎
True
Prelude> 4 < 4 ⏎
False
Prelude> 4 <= 4 ⏎
True
```

Was ist eine Funktion?

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:

$$\mathit{verdopple} : \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} \to \underbrace{\mathbb{Z}}_{\text{Zielbereich}}$$

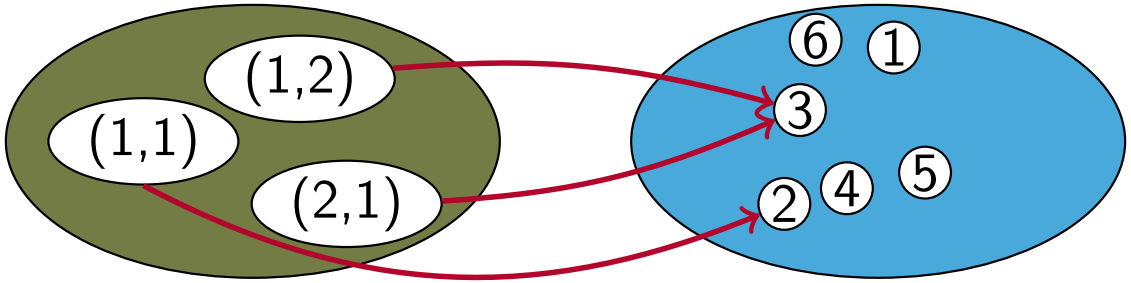
$$\mathit{verdopple}(x) = x + x$$

Mehrstellige Funktionen

Möglichkeit 1: Definitionsbereich ist eine Menge von Tupeln

$$\mathit{summiere} : \underbrace{(\mathbb{Z} \times \mathbb{Z})}_{\text{Definitionsbereich}} \to \underbrace{\mathbb{Z}}_{\text{Zielbereich}}$$

$$\mathit{summiere}(x, y) = x + y$$



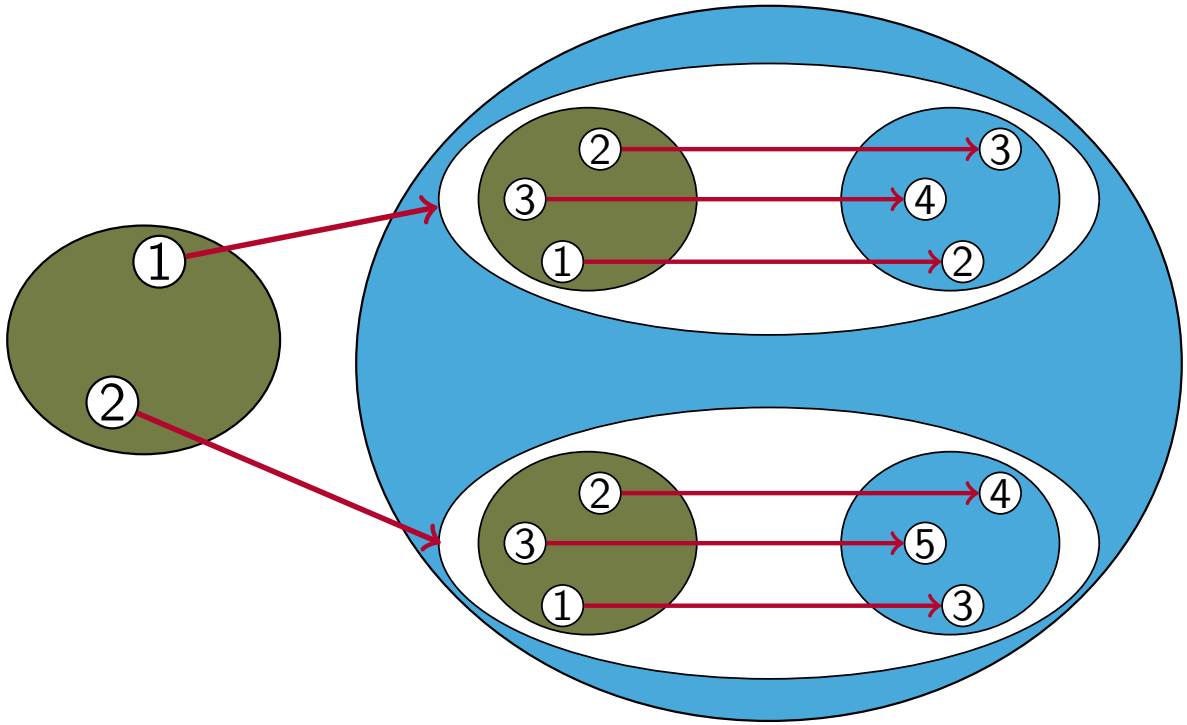
Verwendung: $\mathit{summiere}(1, 2)$

Mehrstellige Funktionen(2)

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

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

$$summiere\ x\ y = x + y$$



Mehrstellige Funktionen (3)

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

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

$$\textit{summiere}\ x\ y = x + y$$

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

```
summiere :: Integer -> (Integer -> Integer)
summiere :: Integer -> Integer -> Integer
summiere 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`

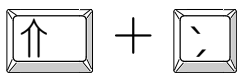
Funktionen: Beispiele

Beispiel: `mod` und `div`

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

```
Prelude> mod 10 3 ⏎
1
Prelude> div 10 3 ⏎
3
Prelude> mod 15 5 ⏎
0
Prelude> div 15 5 ⏎
3
Prelude> (div 15 5) + (mod 8 6) ⏎
5
```

Präfix und Infix

- `+`, `*`, ... werden infix verwendet:
  zwischen den Argumenten, z.B. `5+6`
- `mod`, `div` werden präfix verwendet:
  vor den Argumenten, z.B. `mod 10 3`
- 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 selbst definieren

```
verdopple :: Integer -> Integer
verdopple 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

Man darf auch den Typ angeben!

Formale Parameter

Gesucht:

Funktion erhält zwei Eingaben und liefert
`"Die Eingaben sind gleich"`,
wenn die beiden Eingaben gleich sind.

Falscher Versuch:

```
sonicht x x = "Die Eingaben sind gleich!"
```

```
    Conflicting definitions for ‘x’
    In the definition of ‘sonicht’
Failed, modules loaded: none.
```

Die formalen Parameter müssen unterschiedliche Namen haben.

```
vergleiche x y =
  if x == y then "Die Eingaben sind gleich!" else "Ungleich!"
```

Funktion testen

```
Prelude> :load programme/einfacheFunktionen.hs ⏎
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs)
Ok, modules loaded: Main.
*Main> verdopple 5 ⏎
10
*Main> verdopple 100 ⏎
200
*Main> verdopple (verdopple (2*3) + verdopple (6+9)) ⏎
84
```

Fallunterscheidung: `if-then-else`

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

Bedeutung:

$$\texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2 = \begin{cases} e_1 & \text{wenn } b \text{ zu } \texttt{True} \text{ auswertet} \\ e_2 & \text{wenn } b \text{ zu } \texttt{False} \text{ auswertet} \end{cases}$$

Beispiel

```
verdoppleGerade :: Integer -> Integer
verdoppleGerade x = if even x then verdopple 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> verdoppleGerade 50 ⏎
100
*Main> verdoppleGerade 17 ⏎
17
```

Weitere Beispiele zu if-then-else

Verschachteln von `if-then-else`:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
                if x <= 1000 then 3*x else x
```

Falsche Einrückung:

```
jenachdem x =
if x < 100 then 2*x else
  if x <= 1000 then 3*x else x
```

```
Prelude> :reload ⏎
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )
programme/einfacheFunktionen.hs:9:0:
    parse error (possibly incorrect indentation)
Failed, modules loaded: none.
Prelude>
```

Aufgabe

Ordnung BSc Informatik §36 Abs.9

Die Gesamtnote einer bestandenen Bachelorprüfung lautet:

Bei einem Durchschnitt bis einschließlich 1,5:	sehr gut
bei einem Durchschnitt von 1,6 bis einschließlich 2,5:	gut
bei einem Durchschnitt von 2,6 bis einschließlich 3,5:	befriedigend
bei einem Durchschnitt von 3,6 bis einschließlich 4,0:	ausreichend“

wobei nur die erste Dezimalstelle hinter dem Komma berücksichtigt wird.

Implementiere eine Haskellfunktion

```
gesamtnote :: Double -> String
```

die bei Eingabe eines Durchschnitts die Gesamtnote als String ausgibt.

Higher-Order Funktionen

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

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

```
*Main> wende_an_und_addiere verdopple 10 20 ⏎
60
*Main> wende_an_und_addiere jenachdem 150 3000 ⏎
3450
```

Daher spricht man auch von Funktionen höherer Ordnung!

Nochmal Typen

Typ von `wende_an_und_addiere`

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

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

```
wende_an_und_addiere 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

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden 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`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

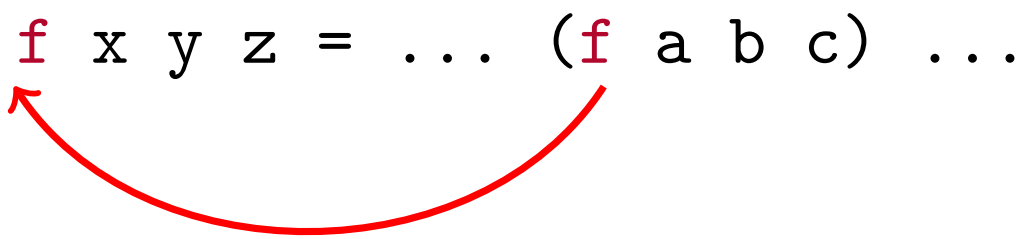
```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

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

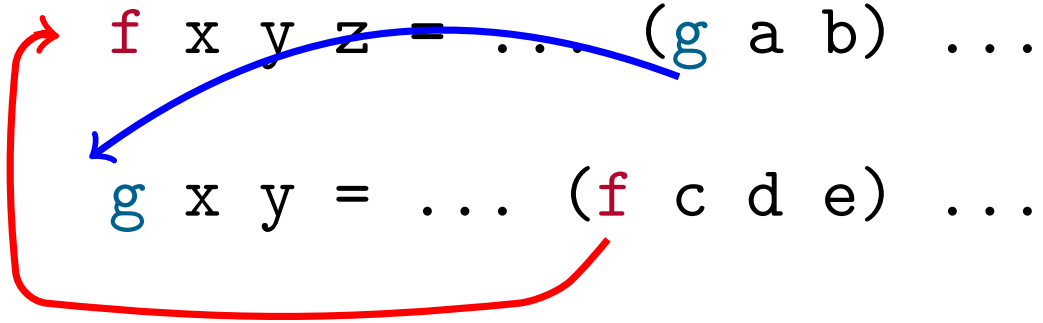
Programmieren mit Haskell
Rekursion

Rekursion

Eine Funktion ist **rekursiv**, wenn sie sich selbst aufrufen kann.



oder z.B. auch



Rekursion (2)

Bei Rekursion muss man aufpassen:

```
endlos_eins_addieren x = endlos_eins_addieren (x+1)
```

```
    endlos_eins_addieren 0
--> endlos_eins_addieren 1
--> endlos_eins_addieren 2
--> endlos_eins_addieren 3
--> endlos_eins_addieren 4
--> ...
```

`endlos_eins_addieren 0` terminiert nicht!

Rekursion (3)

So macht man es richtig:

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

Darauf achten, dass der Rekursionsanfang stets erreicht wird.

Beispiel:

```
erste_rekursive_Funktion x =
  if x <= 0 then 0                                -- Rekursionsanfang
   else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Rekursion (4)

```
erste_rekursive_Funktion x =
 if x <= 0 then 0                              -- Rekursionsanfang
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Was berechnet `erste_rekursive_Funktion`?

- `erste_rekursive_Funktion` n ergibt 0 für $n \leq 0$
- $n > 0$?

Testen:

```
*Main> erste_rekursive_Funktion 5
15
*Main> erste_rekursive_Funktion 10
55
*Main> erste_rekursive_Funktion 11
66
*Main> erste_rekursive_Funktion 12
78
*Main> erste_rekursive_Funktion 1
1
```

Rekursion (5)

```
erste_rekursive_Funktion x =
 if x <= 0 then 0                              -- Rekursionsanfang
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Ein Beispiel nachvollziehen:

```
  erste_rekursive_Funktion 5
= 5 + erste_rekursive_Funktion 4
= 5 + (4 + erste_rekursive_Funktion 3)
= 5 + (4 + (3 + erste_rekursive_Funktion 2))
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 15
```

Rekursion (6)

```
erste_rekursive_Funktion x =
 if x <= 0 then 0                              -- Rekursionsanfang
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Allgemein:

```
  erste_rekursive_Funktion x
= x + erste_rekursive_Funktion (x-1)
= x + (x-1)  + erste_rekursive_Funktion (x-2))
= x + (x-1) + (x-2) + erste_rekursive_Funktion (x-3)))
= ...
```

Das ergibt $x + (x-1) + (x-2) + \ldots + 0 = \sum_{i=0}^{x} i$

Rekursion (7)

Warum ist Rekursion nützlich?

- Man kann damit schwierige Probleme einfach lösen

Wie geht man vor? (z.B. implementiere $f(n) = \sum_{i=0}^{n} i$)

- Rekursionsanfang:
  Der einfache Fall, für den man die Lösung direkt kennt
  $f(0) = \sum_{i=0}^{0} i = 0$
- Rekursionsschritt:
  Man löst ganz wenig selbst, bis das Problem etwas kleiner ist.
  Das (immer noch große) Restproblem erledigt die Rekursion,
  $f(n) = \sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i = n + f(n-1)$ (für $n > 0$)

Fragen?

?