K08

Haskell

1. Funktionsdefinitionen
2. List Comprehensions
3. Rekursive Funktionen
4. Funktionen höherer Ordnung
5. Algebraische Datentypen und Typklassen
6. Interaktive Programme

teilweise basiert auf Folien von Graham Hutton und Philip Wadler

`let`-Ausdrücke

- Wie in anderen funktionalen Sprachen (z.B. Scheme, OCaml) bietet auch Haskell die Möglichkeit in Ausdrücken Variablen einzuführen:

```
let
    y = x * 3
    x = 5
in
    x / y
```

Das ist ein Ausdruck!

- Sinnvoll insbesondere um Lesbarkeit zu verbessern und/oder Wiederholungen in komplexeren Ausdrücken zu vermeiden:

```
cylinderArea :: (RealFloat a) => a -> a -> a
cylinderArea r h =
    let sideArea = 2 * pi * r * h
        topArea  = pi * r ^2
    in  sideArea + 2 * topArea
```

Bedingte Ausdrücke

- Wie in vielen Sprachen hat auch Haskell bedingte Ausdrücke:

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

- Bedingte Ausdrücke können natürlich auch verschachtelt sein:

```
signum  :: Int -> Int
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

- Beachte: Bedingte Ausdrücke haben immer einen **`else`**-Zweig! Warum?

Bedingte Ausdrücke mit Guards

- Guards „|“ bieten eine alternative zu bedingten Ausdrücken, die im Falle von vielen Fallunterscheidungen besser lesbar ist.

```
-- wie zuvor, jedoch unter Verwendung von Guards
abs n | n >= 0      = n
      | otherwise   = -n

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

- **otherwise** definiert in Prelude als: **otherwise = True**

Lokale Definitionen mit **where**

- Problem: Wie kann man Mehrfachauswertung von Ausdrücken umgehen?

```
f x y  |  y >  x*x    =  ...
       |  y == x*x    =  ...
       |  otherwise   =  ...
                -- x*x wird u.U. zwei mal berechnet
```

besser:

```
f x y  |  y >  z      =  ...
       |  y == z      =  ...
       |  otherwise   =  ...
    where z = x*x -- x*x wird nur ein mal berechnet
```

- Beachte: **z** ist nicht ausserhalb **f** sichtbar (nested scope)

Pattern Matching (i)

- Allgemein: Technik zur symbolbasierten Verarbeitung von Strukturen (z.B. Daten oder Funktionen)
  - **Muster** wird benutzt zur **Identifikation** (eines Teils) einer Struktur
    - Das Muster „passt“ oder passt eben nicht auf das was gegeben ist.
    - D.h. es muss eindeutig definiert sein, wie Muster aussehen können und wann ein Muster „passt“.

- Simples Beispiel einer Funktion, die Pattern Matching bzgl. der Argumente benutzt:

```
not       :: Bool -> Bool
not False = True
not True  = False
```

  - Funktionen lassen sich häufig mit Hilfe von Pattern Matching definieren.

Pattern Matching (ii)

- Es bestehen oft mehrere Möglichkeiten eine Funktion auf Basis von Pattern Matching zu definieren, z.B.:

```
(&&)           :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

- Wie zuvor, jedoch kompakter:

```
(&&)           :: Bool -> Bool -> Bool
True && True = True
_    && _    = False  -- _ ist Wildcard die auf jedes
                      -- Argument passt (matcht)
```

Pattern Matching (iii)

- Beachte:
  - Muster werden in der Reihenfolge der Zeilen abgearbeitet:

```
_    && _    = False -- liefert immer False
True && True = True
```

  - Muster dürfen Variablen nicht wiederholen:

```
b && b = True   -- Error: Conflicting definitions
_ && _ = False  -- for `b'
```

Pattern Matching – Listenmuster (i)

- Listen lassen sich rekursiv durch den Listenkompositions-operator `(:)` („cons“) erzeugen bzw. darstellen:

`[1,2,3,4]` ist äquivalent zu `1:(2:(3:(4:[])))`

Kopf Restliste

- Funktionen auf Listen lassen sich durch `x:xs` Muster definieren:

```
head       :: [a] -> a
head (x:_)  = x

tail       :: [a] -> [a]
tail (_:xs) = xs
```

Pattern Matching – Listenmuster (ii)

- Beachte:
  - `x:xs` Muster passen nur auf nichtleere Listen:

```
> head []
Error
```

    Ergo: Funktionen auf Listen immer auch für die leere Liste definieren, insofern sie sich für die leere Liste überhaupt definieren lassen.

  - `x:xs` Muster (und mit anderen Operatoren gebildete Muster links von =) müssen geklammert werden, da Funktionsanwendung höhere Präzedenz als `(:)` hat:

```
head     :: [a] -> a
head x:_  = x  -- Parser Error
```

Pattern Matching – Zahlenmuster

- Funktionen auf Ganzzahlen können (wie in der Mathematik) mit Hilfe von **n+k** Mustern definiert werden, wobei **n** eine Variable und **k** eine Konstante ist:

```
pred       :: Int -> Int
pred (n+1) = n
```

verworfen

- Beachte:
  - **n+k** Muster passen nur auf Argumente **≥k**:

```
pred 0
Error
```

  - Wie zuvor schon erwähnt, muss auch (n+k) geklammert werden:

```
pred    :: Int -> Int
head n+1 = n -- Parser Error
```

Lambda-Ausdrücke

- Man kann in Haskell (und anderen Sprachen) Funktionen definieren, die keinen Namen haben, die also anonym sind. Diese bezeichnet man als λ-Ausdrücke.
  - Lambda-Ausdrücke entstammen dem λ-Kalkül

Schreibweise/Beispiel:

Lambda-Kalkül	**Haskell**
$\lambda x . x + x$	`\x -> x+x`

Anonyme Funktion, die x als Argument hat und $x+x$ als Ergebnis.

Wozu benutzt man λ-Ausdrücke?

1. Zur formalen Darstellung von Curryfizierten Funktionen:

```
add x y = x+y
```
bedeutet
```
add = \x -> (\y -> x+y)
```

2. Zur Definition von Funktionen die Funktionen als Ergebnis haben:

```
const    :: a -> b -> a
const x _ = x
```

```
const  :: a -> (b -> a)
const x = \_ -> x
```

3. Um zu vermeiden dass man einer Funktion, die nur einmal referenziert wird, einen Namen geben muss:

```
odds n = map f [0..n-1]
         where
            f x = x*2 + 1
```

Lässt sich vereinfachen zu

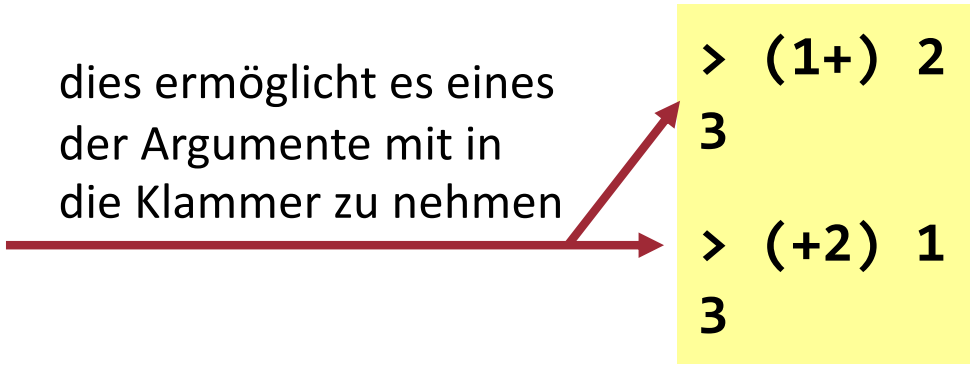
```
odds n = map (\x -> x*2 + 1) [0..n-1]
```

Sektionen

- Da binäre Operatoren curryfizierte Funktionen sind, kann man sie auch partiell anwenden. Dabei wird der Operator-bezeichner (anstatt in Infix-) in geklammerter Präfix- o. Suffix-Schreibweise benutzt.

Beispiel:

```
> 1+2
3

> (+) 1 2
3
```



- **Definition**: Wenn $\oplus$ ein binärer Operator ist, dann werden Funktionen der Form $(\oplus)$, $(x\oplus)$ und $(\oplus y)$ Sektionen genannt.

Wozu benutzt man Sektionen?

- Eine Reihe nützlicher Funktionen lassen sich auf einfache Art und Weise durch Sektionen definieren.

Beispiele:

`(1+)`	- Nachfolgerfunktion	`> (1+) 3` `4`
`(1/)`	- Reziprokwertfunktion	`> (1/) 4` `0.25`
`(*2)`	- Verdopplungsfunktion	`> (*2) 3` `6`
`(/2)`	- Halbierungsfunktion	`> (/2) 1` `0.5`

Sektionen als Lambda-Expression

- Sektionen sind Funktionen. Es gibt also eine äquivalente Lambda-Expression:

Beispiele:

`(1+)`	`\x -> 1 + x`
`(1/)`	`\x -> 1 / x`
`(*2)`	`\x -> x * 2`
`(/2)`	`\x -> x / 2`

K08

Haskell

1. Funktionsdefinitionen
2. **List Comprehensions**
3. Rekursive Funktionen
4. Funktionen höherer Ordnung
5. Algebraische Datentypen und Typklassen
6. Interaktive Programme

List Comprehensions mit Generatoren

- Notation aus der Mathematik zur Mengenbeschreibung existiert quasi analog auch in Haskell; erzeugt werden jedoch Listen (statt Mengen).

Mathematik: $\{ x^2 \mid x \in \{1, \ldots, 5\}\}$

Haskell:

```
[ x^2 | x <- [1..5]]
```

Generator

- Mehrere Generatoren werden durch Komma getrennt; z.B.:

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]

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

Auswirkung der Generatorreihenfolge

- Vertauschung der Reihenfolge der Generatoren bewirkt andere Reihenfolge der Elemente in der Liste:

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

```
> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Mehrere Generatoren sind wie verschachtelte Schleifen, wobei der letzte Generator die innerste Schleife ist.

Generatorabhängigkeit

- Ein Generator kann Variablen eines anderen Generators, der links von ihm steht, benutzen.
  → Abhängigkeit auf Generatoren

```
> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Liste aller Paare (x,y) so dass x,y ∈ {1,2,3} und y ≥ x.

Noch ein Beispiel

- Konkatenation der Elemente einer Liste von Listen:

```
concat     :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

Beispiel:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

Generatoren und Guards

- Guards kann man benutzen zur weiteren Einschränkung der Werte früherer Generatoren:

```
> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

- Funktion, die für eine Zahl alle ihre Teiler berechnet:

```
factors  :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

```
> factors 15
[1,3,5,15]
```

Beispiel – Primzahltest -und Berechnung

- Primzahl: ist nur durch 1 und sich selbst teilbar.

```
prime  :: Int -> Bool
prime n = factors n == [1,n]
```

```
> prime 15
False
> prime 7
True
```

- Berechnung aller Primzahlen von 1 bis n mit Hilfe eines Generators, eines Guards und unserer **`prime`**-Funktion:

```
primes  :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

String Comprehensions

- Da Strings Listen von Zeichen (**Char**) sind, lassen sich alle polymorphen Funktionen auf Listen auch auf Strings anwenden, z.B.:

```
> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

```
> length "abcde"
5
> take 3 "abcde"
"abc”
```

- Aus dem gleichen Grund kann man List Comprehensions zur Definition von Funktionen auf Strings benutzen, z.B.:

```
lowers   :: String -> Int
lowers xs =
  length [x | x <- xs, isLower x]
```

```
> lowers "Haskell"
6
```

K08

Haskell

1. Funktionsdefinitionen
2. List Comprehensions
3. **Rekursive Funktionen**
4. Funktionen höherer Ordnung
5. Algebraische Datentypen und Typklassen
6. Interaktive Programme

(Wiederholung) Rekursion

- Viele Funktionen lassen sich mit Hilfe von anderen Funktionen definieren, z.B.:

```
-- factorial definiert mit Hilfe von product
factorial  :: Int -> Int
factorial n = product [1..n]
```

- Rekursion: Funktion f definiert mit Hilfe von sich selbst

```
factorial  :: Int -> Int
factorial 0 = 1
factorial n+1 = (n+1) * factorial n
```

  - Beachte: **`factorial`** divergiert für n < 0, da Abbruchbedingung niemals erreicht wird:

```
> factorial (-1)
Error: Control stack overflow
```

Rekursive Auswertung von `factorial`

```
    factorial 3
=
    3 * factorial 2
=
    3 * (2 * factorial 1)
=
    3 * (2 * (1 * factorial 0))
=
    3 * (2 * (1 * 1))
=
    3 * (2 * 1)
=
    3 * 2
=
    6
```

Rekursion auf Listen

- Rekursion kann natürlich auch für die Definition von Funktionen auf Listen verwendet werden.

Beispiel:

```
product        :: [Int] -> Int
product []      = 1
product (n:ns) = n * product ns
```

```
    product [2,3,4]
=
    2 * product [3,4]
=
    2 * (3 * product [4])
=
    2 * (3 * (4 * product []))
=
    2 * (3 * (4 * 1))
=
    24
```

Weitere Beispiele

- Rekursive Variante der Länge einer Liste:

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

- Rekursive Variante einer Liste in umgekehrter Reihenfolge:

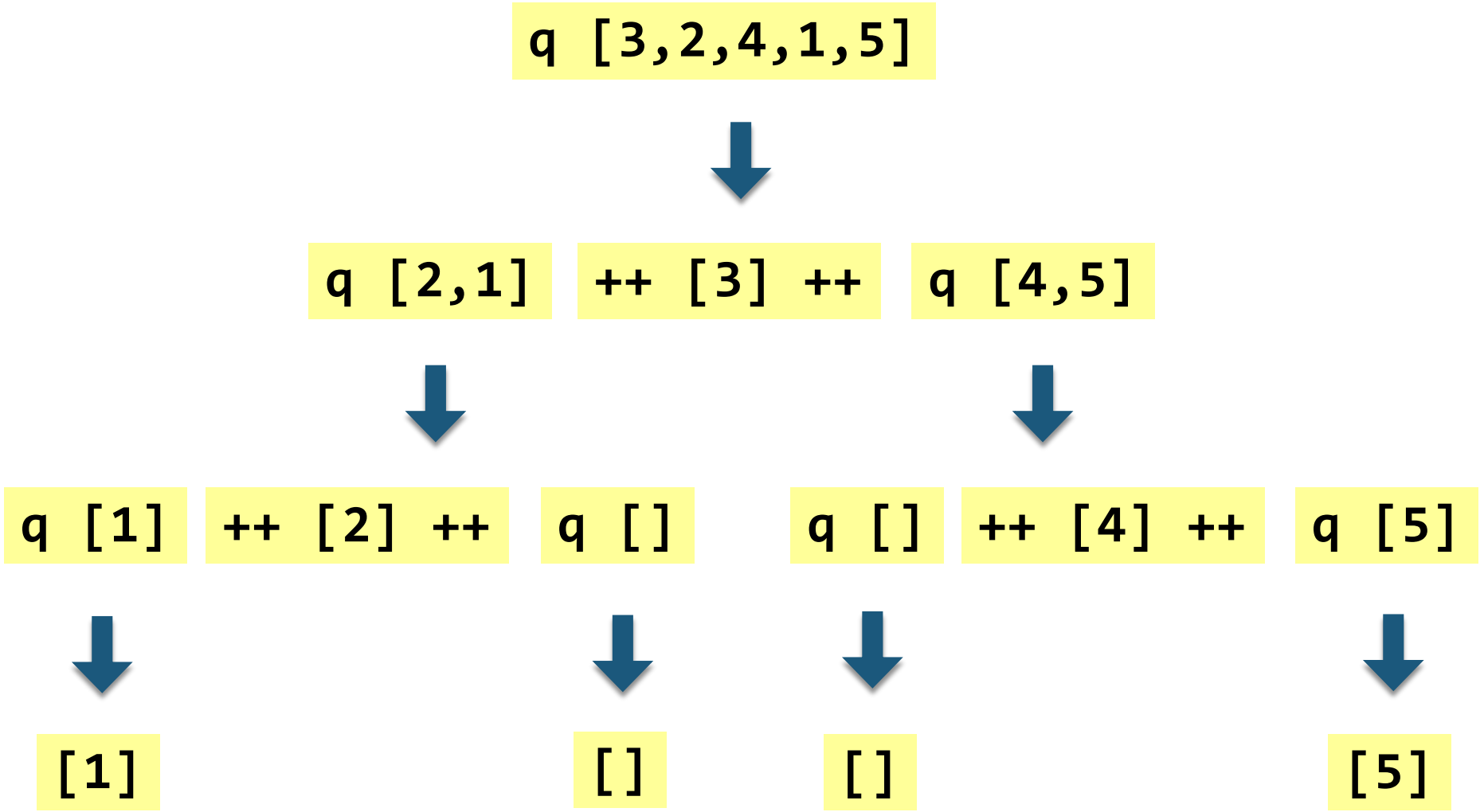
```
reverse        :: [a] -> [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

Beispiel Quick Sort (i)

- Quick Sort kann durch zwei Regeln definiert werden:
  - Leere Liste ist bereits sortiert
  - Nichtleere Listen sortiert man indem man Kopf von Restliste abspaltet und dann
    - Restliste ≤ Kopf rekursiv sortiert
    - Restliste > Kopf rekursiv sortiert
    - Resultierende sortierte Listen an jeweilige Seite des Kopfes anhängt

```
quickSort        :: Ord a => [a] -> [a]
quickSort []      = []
quickSort (x:xs) = quickSort ys ++ [x] ++ quickSort zs
                   where
                      ys = [a | a <- xs, a <= x]
                      zs = [b | b <- xs, b > x]
```

Beispiel Quick Sort (ii)



K08

Haskell

1. Funktionsdefinitionen
2. List Comprehensions
3. Rekursive Funktionen
4. **Funktionen höherer Ordnung**
5. Algebraische Datentypen und Typklassen
6. Interaktive Programme

`map`-Funktion

- **`map`** wendet eine Funktion auf alle Elemente einer Liste an und gibt eine Liste der Ergebnisse zurück.

```
map :: (a -> b) -> [a] -> [b]
```

Beispiel (wende Successor-Funktion an):

```
> map (+1) [1,3,5]
[2,4,6]
```

```
-- Definition mittels List Comprehension
map f xs = [f x | x <- xs]

-- Definition mittels Rekursion
map f []     = []
map f (x:xs) = f x : map f xs
```

`filter`-Funktion

- **`filter`** selektiert alle Elemente einer Liste die ein gegebenes Prädikat – eine Boolesche Funktion – erfüllen.

```
filter :: (a -> Bool) -> [a] -> [a]
```

Beispiel (selektiere nur geraden Zahlen):

```
> filter even [1..10]
[2,4,6,8,10]
```

```
-- Definition mittels List Comprehension
filter p xs = [x | x <- xs, p x]

-- Definition mittels Rekursion
filter p []     = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Faltungsfunktionen

- Sukzessive Faltung (Reduktion) der Elemente einer Liste auf genau ein Endelement gemäss eines Faltungsoperators ⊕.
- Allgemeines rekursives Muster:

```
f []     = v -- beliebiger Wert; meist neutrales
             -- Element bezüglich ⊕
f (x:xs) = x ⊕ f xs    -- von rechts nach links
```

  - Von rechts nach links: **`foldr`**

`[1,2,3]` `(1⊕(2⊕(3⊕v)))`

  - Von links nach rechts: **`foldl`**

`[1,2,3]` `(((v⊕1)⊕2)⊕3)`

Ist ⊕ nicht assoziativ, dann spielt die Richtung eine Rolle!

Beispiele für Faltungsfunktionen

v = 0
⊕ = +

```
sum []     = 0
sum (x:xs) = x + sum xs
```

```
sum = foldr (+) 0
```

v = 1
⊕ = *

```
product []     = 1
product (x:xs) = x * product xs
```

```
product = foldr (*) 1
```

v = True
⊕ = &&

```
and []     = True
and (x:xs) = x && and xs
```

```
and = foldr (&&) True
```

v = False
⊕ = ||

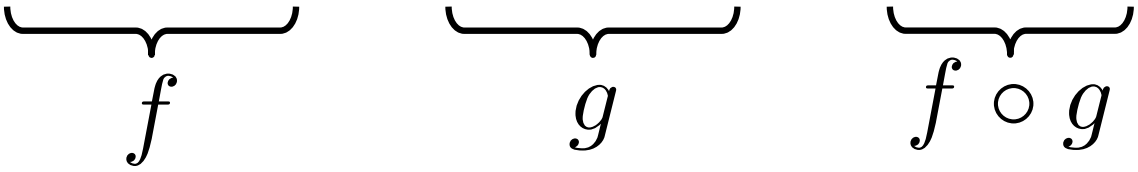
```
or []     = False
or (x:xs) = x || or xs
```

```
or = foldr (||) False
```

Funktionskomposition

- Math. Funktionskomposition $f \circ g$ ist eine (einfache) Funktion höherer Ordnung.
  - In Haskell realisiert durch **(.)**-Operator

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```



  - Beispiel (Anzahl Wörter die mit einem Grossbuchstabe beginnen).

```
import Data.Char
capCount :: [Char] -> Int
capCount = length . filter (isUpper . head) . words
```

```
> capCount "Hello there, Mom!"
2
```

Funktionsapplikation mit $

- „normale“ Funktionsapplikation mit Leerzeichen ist linksassoziativ:

```
f a b c   -- ist äquivalent zu (((f a) b) c)
sum (filter (>10) (map (*2) [2..10])) -- Klammern notwendig
```

- Kann man diese unschönen Klammern loswerden?
  Lösung: **$**-Operator:

```
($) :: (a -> b) -> a -> b
($) f x = f x  -- bzw. mit Infixschreibweise: f $ x = f x
```

- **$**-Operator hat niedrigste Präzedenz und ist rechtsassoziativ:

```
f $ g $ h x --äquivalent zu f $ (g $ (h x)) bzw. f (g (h x))

sum (filter (>10) (map (*2) [2..10]))
sum $ filter (>10) $ map (*2) [2..10] -- leichter lesbar
```

Weitere Beispiele f. Funktionen höherer Ordnung

```
-- all testet ob jedes Listenelement Prädikat p erfüllt
all     :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
```

```
> all even [2,4,6,8,10]
True
```

```
-- any testet ob irgend ein Listenelement Prädikat p erfüllt
any     :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]
```

```
> any isSpace "abc def”
True
```

```
-- selektiert Listenelement solange Prädikat p erfüllt
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []      = []
takeWhile p (x:xs)
   | p x            = x : takeWhile p xs
   | otherwise      = []
```

Analog existiert **dropWhile**

```
> takeWhile isAlpha "abc def”
"abc"
```

K08

Haskell

1. Funktionsdefinitionen
2. List Comprehensions
3. Rekursive Funktionen
4. Funktionen höherer Ordnung
5. **Algebraische Datentypen und Typklassen**
6. Interaktive Programme

Typdeklarationen (Synonyme)

- **`type`** bietet die Möglichkeit einen neuen Namen (Synonym) für einen existierenden Typ zu vergeben (Listen, Tupel, Funktionen, algebraische Datentypen).

Beispiel:

```
type String = [Char]
```

`String` ist jetzt eine Synonym für **`[Char]`**

- Wird hauptsächlich verwendet zur Verbesserung der Lesbarkeit von komplexen Typen.

```
-- sei (Int,Int) ein
-- Punkt in der Ebene
type Pos = (Int,Int)
```

Verwendung →

```
origin     :: Pos
origin      = (0,0)

left       :: Pos -> Pos
left (x,y) = (x-1,y)
```

Weitere Möglichkeiten f. Typdeklarationen

- Analog zu Funktionen können Typdeklarationen Parameter haben:

```
type Pair a = (a,a)
```

Verwendung →

```
mult      :: Pair Int -> Int
mult (m,n) = m*n

copy      :: a -> Pair a
copy x     = (x,x)
```

- Verschachtelung möglich:

```
type Pos   = (Int,Int)
type Trans = Pos -> Pos
```

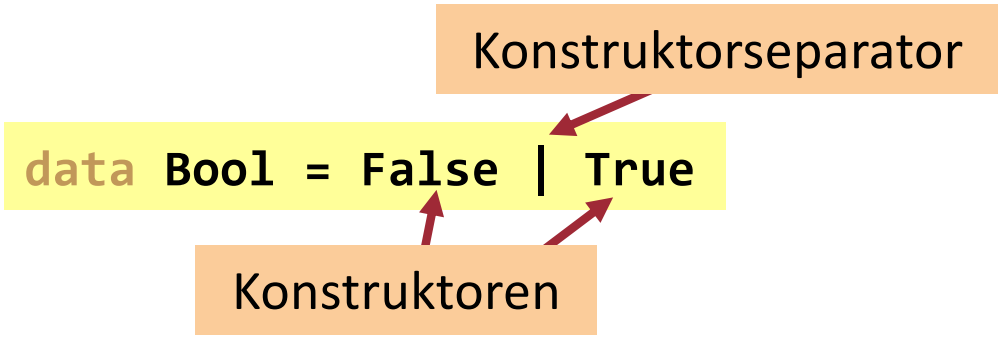
- Rekursive Deklarationen nicht möglich:

```
type Tree = (Int,[Tree])
```

Algebraische Datentypen

- Ein algebraischer Datentyp kann mehr als einen Konstruktor (engl. value constructor) haben mit dem sich Werte des Typs erzeugen lassen.
  - Werden mit dem Schlüsselwort **`data`** deklariert.
  - Sind vergleichbar mit kontextfreien Grammatiken.

Beispiel (aus Prelude):



Beispiel 2:

```
type Id = Int
type Title = String
type Authors = [String]
data BookInfo = Book Id Title Authors
```

components

type constructor

value constructor

Werte algebraischer Datentypen erzeugen

- Gegeben:

```
data Answer = Yes | No | Unknown
```

- Dann können wir wie folgt davon Gebrauch machen:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown] -- Liste aller mögl. Antworten

flip         :: Answer -> Answer -- Funktion auf Antworten
flip Yes      = No
flip No       = Yes
flip Unknown  = Unknown
```

Strukturelle Äquivalenz und Typgleichheit

- Beachte: Algebraische Datentypen mit identischer Struktur aber unterschiedlichem Name sind **nicht** vom selben Typ!

Beispiel:

```
-- x, y Koordinaten
data Cartesian2D = Cartesian2D Double Double

-- Winkel und Abstand
data Polar2D     = Polar2D Double Double
```

```
> Cartesian2D (sqrt 2) (sqrt 2) == Polar2D (pi/4) 2
Couldn’t match expected type 'Cartesian2D’
       against inferred type 'Polar2D'
```

Weiteres Beispiel zu Konstruktoren mit Parametern (analog **`BookInfo`**)

- **Shape** ist entweder ein Kreis mit einem Durchmesser oder ein Rechteck mit den beiden Seitenlängen:

```
data Shape = Circle Float
           | Rect Float Float
```

Circle und Shape lassen sich wie Funktionen auffassen

```
Circle :: Float -> Shape
Rect   :: Float -> Float -> Shape
```

- Damit lassen sich z.B. folgende Funktionen definieren:

```
square          :: Float -> Shape
square n         = Rect n n

area            :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Alegebr. Datentypen mit Parametern

- Analog zu Typdeklarationen sind auch hier Parameter möglich.

```
data Maybe a = Nothing | Just a
```

- **Maybe** (aus Prelude) erlaubt insbesondere partielle Funktionen elegant zu definieren:

```
safediv    :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Alegebr. Datentypen mit Parametern

- Parametrisierte Wertkonstruktoren (value constructors) sind auch Funktionen.

```
Nothing :: Maybe a
Just 10 :: Maybe Int
Just :: a -> Maybe a
```

- Deshalb kann man sie entsprechend einsetzen:

```
map Just [1..5]
```

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

Rekursive algebr. Datentypen

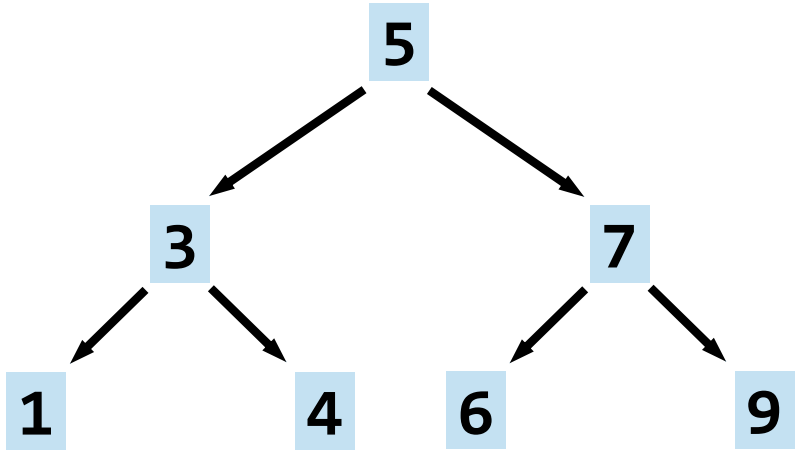
- In Haskell kann man rekursive algebr. Datentypen definieren.

Beispiel: Binärbaum (Baum in dem jeder Knoten genau zwei Kindknoten hat; m.a.W. Baum mit Verzweigungsgrad 2)

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

Anwendung:

```
Node (Node (Leaf 1) 3 (Leaf 4))
     5
     (Node (Leaf 6) 7 (Leaf 9))
```



Typklassen

- Eine Typklasse spezifiziert eine Menge von Funktionen, die alle Instanzen dieser Klasse unterstützen müssen.
  - Dazu wird das Schlüsselwort **`class`** verwendet
  - Realisieren *ad hoc* Polymorphie bzw. Überladen von Verhalten
  - Grob gesagt vergleichbar mit Interfaces in Java

Beispiel:

```
class Eq a where
  (==),(/=) :: a -> a -> Bool
  -- default Implementierung:
  x /= y     = not (x == y)
  x == y     = not (x /= y)
```

Achtung: Instanzen müssen einen der beiden Operatoren überschreiben (definieren). Ansonsten kommt es zu einer Endlosschleife wegen der gegenseitigen Abhängigkeit.

Typ der Operatoren ist demzufolge (type inference):

```
(==),(/=) :: Eq a => a -> a -> Bool
```

Typklassen – Beispiele

- Einige der am häufigsten verwendeten Typklassen in Haskell:

```
Num   + - * abs signum negate etc.
```

```
Fractional / recip
```

```
Integral   div mod etc.
```

```
Show  show :: a -> String
```

```
Ord   < <= > >= max min
```

```
Enum  succ pred
```

Instanz einer Typklasse erzeugen

- Von einer Typklasse erzeugt man mit dem Schlüsselwort **`instance`** eine Instanz.
  - Falls Typklasse Default-Implementierung der definierten Funktionen besitzt, muss Instanz diese nur überschreiben wenn sie umdefiniert werden soll.
  - Man kann keine Instanzen von Typsynonymen erzeugen.

Gegeben:

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

Dann:

```
instance Eq Color where
  -- überschreiben von ==
  Red   == Red   = True
  Green == Green = True
  Blue  == Blue  = True
  _     == _     = False
```

Automatische Instanzerzeugung

- **`data`** bietet ein optionales Schlüsselwort **`deriving`** bei dessen Verwendung automatisch Instanzen für die angegebenen Typklassen erzeugt werden (vom Compiler).
  - Programmierer muss dann die Funktionen der Typklassen nicht mehr selbst definieren.
  - Dies ist nur für eine Reihe vorhandener Typklassen möglich, deren Definition „trivial“ übertragen werden kann:
    `Read`, **`Show`**, **`Bounded`**, **`Enum`**, **`Eq`**, **`Ord`**

Beispiel:

```
data Color = Red | Green | Blue
     deriving (Read, Show, Eq, Ord)
```

K08

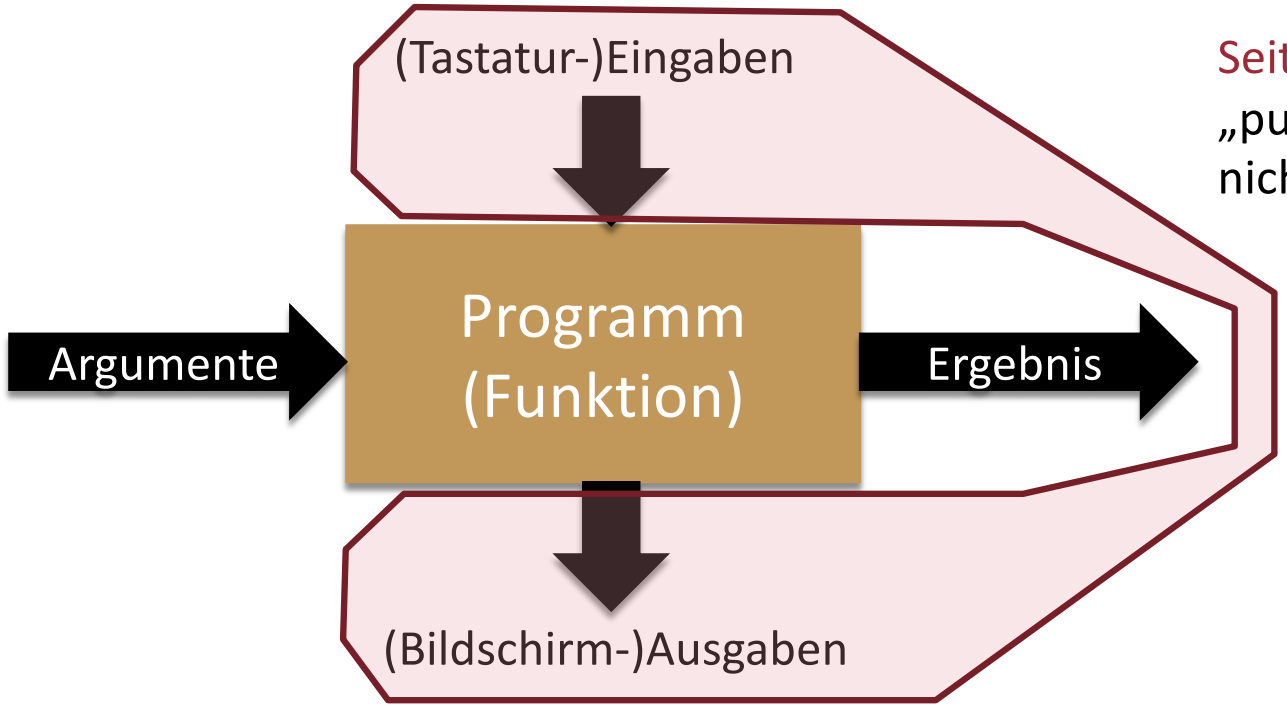
Haskell

1. Funktionsdefinitionen
2. List Comprehensions
3. Rekursive Funktionen
4. Funktionen höherer Ordnung
5. Algebraische Datentypen und Typklassen
6. **Interaktive Programme**

Interaktive Programme

- Mit den bisher kennen gelernten Mitteln können wir nur Programme schreiben, die beim Start alle Argumente entgegen nehmen, die sodann die programmierte Gesamtfunktion berechnen und am Ende ein Ergebnis liefern.

  → Interaktion mit der Umgebung bisher nicht möglich



Seiteneffekte; mit „puren“ Funktionen nicht realisierbar.

Lösung

- Interaktive Programme können in Haskell geschrieben werden auf Basis von Typen durch die sich „reine“ Ausdrücke unterscheiden lassen von „unreinen“ Aktionen (engl. Actions) welche Seiteneffekte haben dürfen.

`IO a` Typ von Aktionen die, als Ergebnis ihrer **Ausführung**, einen Wert vom Typ `a` zurückgeben.

`IO Char` Aktionen die `Char` zurückgeben.

`IO ()` Aktionen die **„nichts“** zurückgeben und demzufolge nur Seiteneffekte haben können.

Aktionen (i)

- Genau genommen ist **`IO a`** ein Typsynonym:

```
type IO a  =  RealWorld -> (RealWorld, a)
```

  - **`RealWorld`** ist ein fingierter (engl. fake) Typ, dessen fingierte Werte jeweils einen aktuellen Weltzustand repräsentieren.
  - Aktionen sind also Funktionen des Weltzustandes, der abgebildet wird auf einen neuen Weltzustand und ein optionales Ergebnis (**`a`**).
    - Da Weltzustand fiktiv, macht Auswertung im Sinne einer Funktion keinen Sinn: man muss Aktionen ausführen!
    - Weltzustand wird zur Compilezeit sogar „wegoptimiert“. Ergo: Weltzustand existiert nur auf Sprachebene und verursacht keine Kosten zur Laufzeit.

- **`IO`** ist gleichzeitig eine Instanz der Typklasse **`Monad`**:

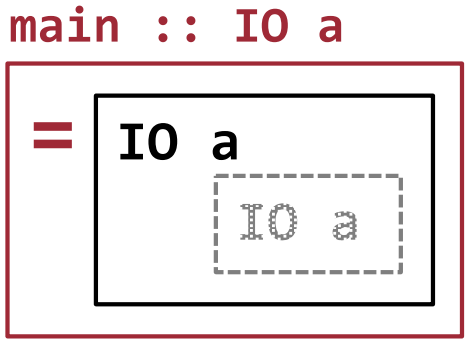
```
instance Monad IO
```

Aktionen - Beispiele (ii)

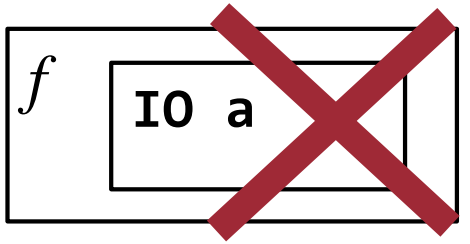
```
putStr :: String -> IO ()    -- gibt String auf stdout aus

putStrLn :: String -> IO () -- gibt String auf stdout
                               aus mit Zeilenumbruch

getLine :: IO String         -- liest Zeile (fortlaufende
                               Zeichen bis Zeilenumbruch)
                               von stdin

print :: Show a => a -> IO () -- gibt "druckbaren" Wert
                                 auf stdout aus

readLn :: Read a => IO a     -- liest "parsbaren" Wert aus
                               Zeile an stdin
```

Aktionen (iii)

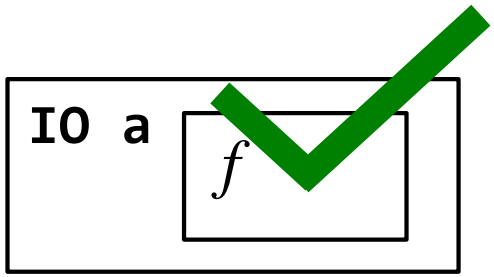
- Die einzige Möglichkeit eine Aktion auszuführen, egal ob direkt oder indirekt, geschieht durch das Binden an **`main`**.



- Aktionen können nicht aus reinen Funktionen aufgerufen werden → Funktion würde damit ihre funktionale Eigenschaft verlieren (wegen der Seiteneffekte) und würde zu einer Aktion.



- Umkehrung möglich: Aktionen können sehr wohl Funktionen aufrufen.



Komposition von Aktionen (i)

- Erfolgt auf Basis von Bindeoperatoren auf monadischen Typen

- Komposition zweier Aktionen, so, dass diese sequenziell ausgeführt werden, wobei das Ergebnis der ersten Aktion stillschweigend ignoriert wird.

```
(>>) :: Monad m => m a -> m b -> m b
```

Beispiel:

```
putChar '?' >> putChar '!'
```

Zusammengesetzte Aktion, deren Ausführung "?!" auf der Standardausgabe ausgibt.

```
putChar :: String -> IO ()
```

Komposition von Aktionen (ii)

- Komposition zweier Aktionen, so, dass diese sequenziell ausgeführt werden, wobei das Ergebnis der ersten Aktion als Eingabe der zweiten dient.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Beispiel:

```
readLn >>= (\a -> print a)
```

Zusammengesetzte Aktion, deren Ausführung zuerst eine Zeile von der Standardeingabe liest, diese sodann auf der Standardausgabe ausgibt.

```
readLn :: Read a => IO a
print :: Show a => a -> IO ()
```

do-Notation

- Benutzung von **>>** und **>>=** wird schnell schwer lesbar.
  - Abhilfe: Strukturen auf Basis von **do** (Syntactic shugar)

Beispiel: Lese einen String von der Standardeingabe

```
getLine :: IO String
getLine  = do x <- getChar
              if x == '\n' then
                 return []
              else
                 do xs <- getLine
                    return (x:xs)
```

Binde Ergebnis an Variable

Aktion **return a** gibt lediglich Ergebnis **a** zurück ohne dass ihre Ausführung irgendeine Interaktion bewirkt.

Funktionen innerhalb **do** auswerten

- Wie bettet man Funktionsauswertung in **do**-Sequenzen ein?

pal.hs

```
main :: IO ()
main = do l <- getLine
          let lrev = reverse l
          let le = (show . length) l
          let el = reverse le
          putStrLn (l ++ le ++ el ++ lrev)
```

Ausdrücke

```
-- show (length l)
```

```
> ./pal
Hello, World!
Hello, World!1331!dlroW ,olleH
```

Weiteres Beispiel

- Fordere zur Eingabe eines Strings auf, berechne die Länge des Strings und gib selbige auf der Standardausgabe aus.

```
main :: IO ()
main  = do putStr "Enter a string: "
           s <- getLine
           putStr "The string has "
           putStr (show (length s))
           putStrLn " characters."
```

K08

Haskell

... One last thing ...

Parallele Ausführung nutzen

test.hs

```
import Control.Parallel

main = a `par` b `par` c `pseq` print (a + b + c)
    where
        a = ack 3 10
        b = fac 1000
        c = 2 ^ 1000

fac 0 = 1
fac n = n * fac (n-1)

ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Berechnung von **a** erfolgt parallel zu **b** und **c**. Sobald alle drei berechnet wurden erfolgt Ausgabe der Summe (**pseq**).

```
> ghc -O2 -o test test.hs -threaded -rtsopts
```

```
> time ./test +RTS -N2
> 402387260077093773543 ... 652624386837205668077565
> ./test +RTS -N2  1.24s user 0.06s system 149% cpu 0.65 total
```