HASKELL

KAPITEL 2.1

Notationen: Currying und das Lambda-Kalkül

Bisheriges (Ende VL-Teil 1)

```
weite :: (Float,Float) ->Float

weite (v0, phi) = (square(v0)/9.81) *
sin(2 * phi) (10, 30°)


smaller ::(Integer, Integer) -> Integer

smaller (x,y) = if x <= y then x else y
```

`(Float,Float) oder (Integer,Integer)`

- bisher genutzt um **EIN** Funktionsargument bzw. **EINEN** Wertebereich zu haben, da wir dies so „gewöhnt“ sind

- in HASKELL kann man Funktionen mit mehreren Argumenten aber auch in einer „**curried**“ Form spezifizieren (nach Haskell B. Curry)

Bsp.:

```
multiply  :: (Int,Int) -> Int

multiply (x,y) = x * y
```

Bemerkung : maxBound von `Int` ist 2.147.483.647;
`Integer` beinhaltet beliebig grosse ganze Zahlen

`multiply` mit **currying**:

```
multiply_c  :: Int -> Int -> Int

multiply_c x y = x*y
```

Sinn :

Partielle Anwendungen im Sinne der Funktionalität werden möglich. So gibt `multiply_c 2` in HASKELL etwas Sinnvolles zurück, nämlich eine Funktion „ 2 * “ f: Int -> Int mit f(y) = 2 * y .

Verallgemeinerung :

$f(x_1, \ldots, x_n) = y$ sei vom Typ $(t_1, \ldots, t_n) \rightarrow t$

Curry – Version hat Typ:

$$t_1 \rightarrow t_2 \rightarrow .... \rightarrow t_n \rightarrow t$$

Der Typ einer partiellen Anwendung von f auf die Argumente $x_1, x_2, \ldots, x_k$ mit $k \leq n$ erzeugt als Ergebnistyp:

$$t_{k+1} \rightarrow t_{k+2} \rightarrow .... \rightarrow t$$

Erinnerung:

Die Anwendung einer Funktion auf ihre Argumente ist **links-assoziativ**:

f x y = (f x) y nicht : f x y = f (x y) !

Das Typ-Abbildungssymbol "->" ist **rechts-assozitiv:**

a -> b -> c bedeutet a -> (b -> c)

Ergo ist:

```
multiply_c  :: Int -> Int -> Int

multiply_c x y = x*y
```

eine Abkürzung für :

```
multiply_c  :: Int -> (Int -> Int)

multiply_c x y = x*y
```

und kann daher auf einen Wert des Typs "`Int`" angewendet werden:

```
multiply_c 2 :: Int -> Int
```

```
multiply_c 2 :: Int -> Int
```

ist nun eine Funktion, die wiederum auf ein Argument des Typs "`Int`" angewendet werden kann:

```
(multiply_c 2) 5 :: Int
```

welches, da Funktionsanwendungen links-assoziativ sind, wieder geschrieben werden kann als:

```
multiply_c 2 5 :: Int
```

Die Lambda Notation

Funktion: Zuordnung eines Wertes zu Argumenten

```
f = λ x , y . x * y     Kommentar : vor x,y steht ein „Lambda“
```

“f ist diejenige Funktion, die den Argumenten x und y den Wert x * y zuordnet”.

Haskell:

```
f = \(x, y) -> x * y
```
oder

```
f = \ x y -> x * y
```
oder

anonymisiert

```
\ x y -> x * y
```

Sinn :

- schnelles Implementieren einfacher Hilfsfunktionen
- das Lambda-Kalkül (- Notation in Haskell) ist allgemeiner als die partielle Funktionsanwendung und kann daher in Situationen angewendet werden, wo diese versagt (Beispiele später, wenn mehr Stoff vorhanden)

HASKELL

KAPITEL 3

Ausdrücke

weitere Ausdrucksmittel

bisher:

ein Programm ist ein elementarer arithmetischer Ausdruck.

Wie schreibt man ein komplizierteres?

Weitere Ausdrucksmittel:

Fallunterscheidung, „bewachte“ Teilausdrücke

$$max, min :: Float \to Float \to Float$$

$$max\ a\ b = \textbf{if}\ a \geq b\ \textbf{then}\ a\ \textbf{else}\ b$$

$$min\ a\ b = \textbf{if}\ a \leq b\ \textbf{then}\ a\ \textbf{else}\ b$$

$$abs :: Float \to Float$$

$$abs\ a = \textbf{if}\ a \geq 0\ \textbf{then}\ a\ \textbf{else}\ -a$$

sign

$$sign :: Float \rightarrow Integer$$

$$sign\ a$$

$$\mid a > 0 = +1$$

$$\mid a = 0 = 0$$

$$\mid a < 0 = -1$$

Alternativen sind hier disjunkt und vollständig.

Allgemeinerer Fall

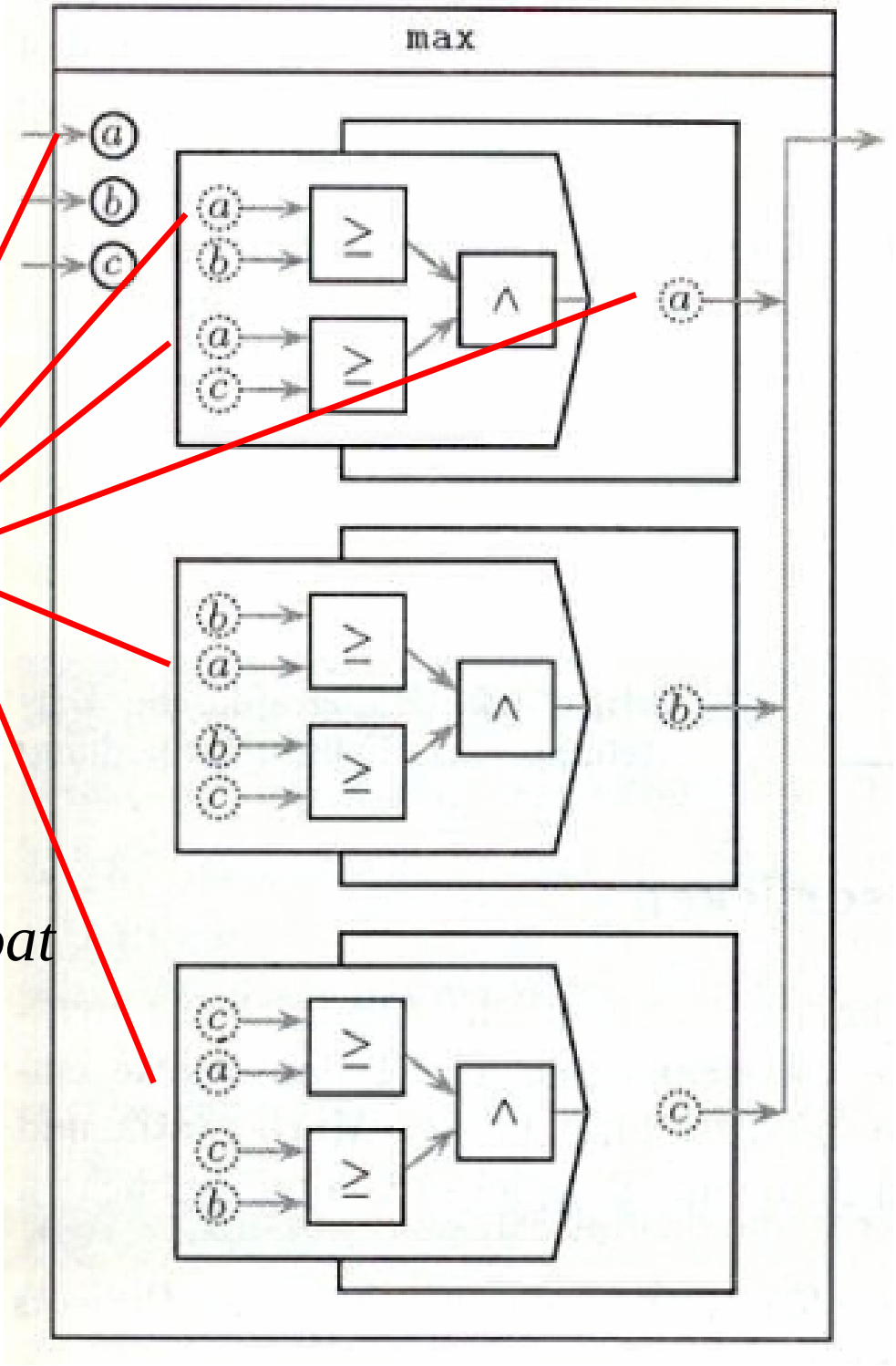
„... *disjunkt und vollständig“* ist nicht zwingend:
Sei *process* ein Datentyp mit einer Priorität. Dann möglich:

schedule :: process → process → process
schedule P Q
| *priority(P)* $\geq$ *priority(Q) = P*
| *priority(Q)* $\geq$ *priority(P) = Q*
nicht disjunkt; *nichtdeterministisch.*

schedule :: process → process → process
schedule P Q
| *priority(P) > priority(Q) = P*
| *priority(Q) > priority(P) = Q*
nicht vollständig; *partiell.*

Fallunterscheidung graphisch

Anschluss-punkte: Knoten gleichen Namens identifizieren



$max :: \quad Float \to Float \to Float \to Float$

$max \quad a \quad b \quad c$

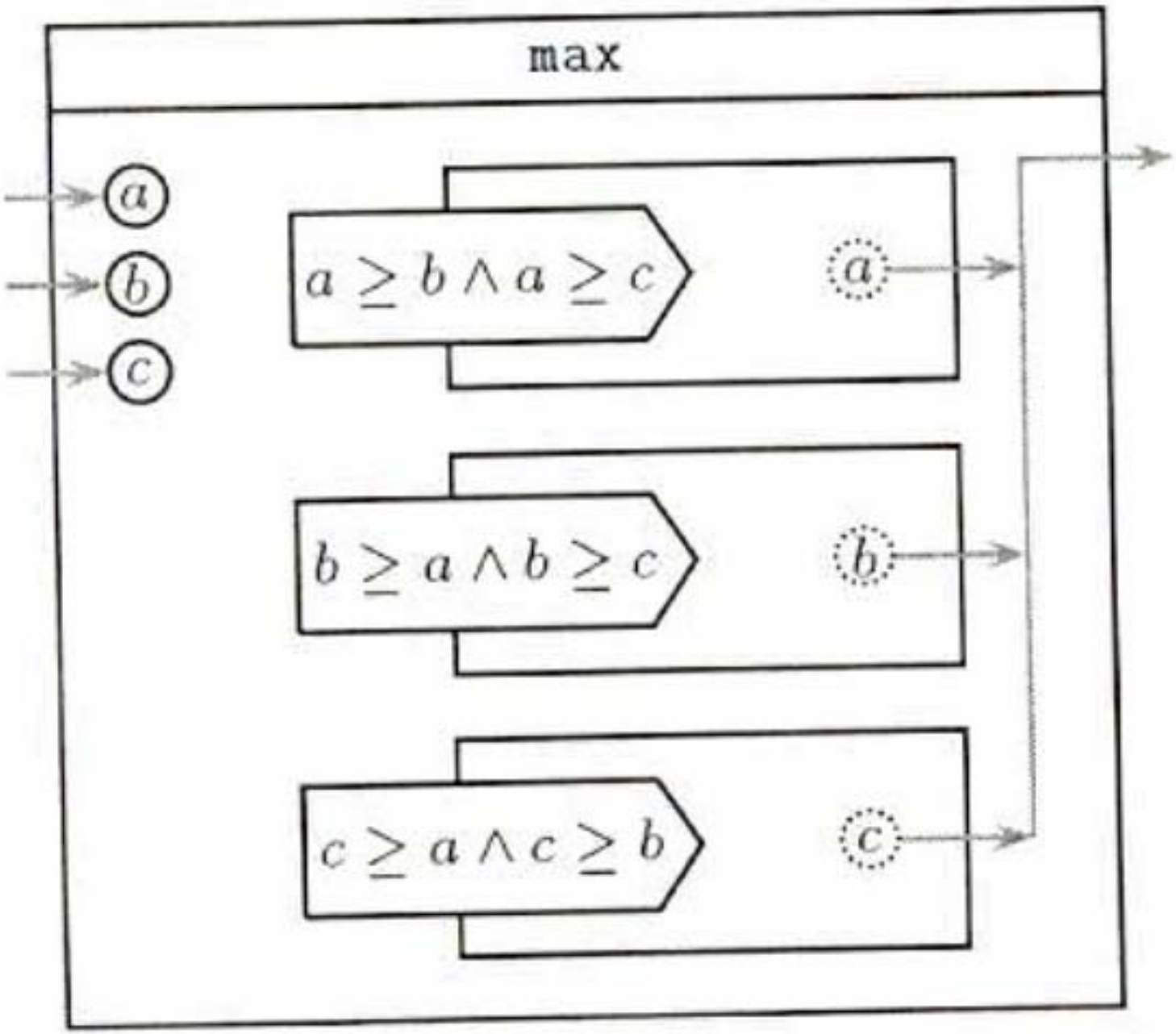
$\quad | a \geq b \wedge a \geq c \;\; = \;\; a$

$\quad | b \geq a \wedge b \geq c \;\; = \;\; b$

$\quad | \; c \geq a \wedge c \geq b \;\; = \;\; c$

Fallunterscheidung einfacher

halbgraphisch:



Teilausdrücke benennen

Beispiel: Heron‘sche Formel für den Flächeninhalt eines Dreiecks

heron :: (Float, Float, Float)→ Float

heron (a, b, c) = sqrt(s (s-a)*(s-b)*(s-c))*

where *s = (a + b + c) /2*

mehrere Teilausdrücke zugleich

Beispiel: die beiden Lösungen einer quadratischen Gleichung

$ax^2 + bx + c = 0$

$roots :: (Float, Float, Float) \rightarrow (Float, Float)$

$roots\ \ (a, b, c) = (x1, x2)$

where

$x_1 = (-b + d) / (2*a)$

$x_2 = (-b - d) / (2*a)$

$d = sqrt((b*b) - (4*a + c))$

Funktionsausdrücke als Teilausdrücke

Beispiel: $a^2 + b^2$

squaresum :: (Float, Float)→Float

squaresum (a, b) = sq(a) + sq(b)

where

*sq x = x*x*

Tupel als Ausdrücke

Beispiel: die beiden Lösungen von $\surd x$

gegeben: *sqrt(x)* berechnet die positive $\surd x$.

sqrts :: Float → (Float, Float)

sqrts x = (sqrt(x) , – sqrt(x))

zusammengefasst: Lokale Deklaration

abkürzende Namen für Teilausdrücke

lokal: **let .. in ..** **let** *s = (a + b + c) /2* **in**
sqrt(s (s-a)*(s-b)*(s-c))*

where: sq(a) + sq(b) **where** sq x = x*x

möglich: mehrere neue Namen zugleich: *(x1, x2)* **where** ...
Reihenfolge egal

Tupel auch links ... **let** *(quote, rest) = divmod(a,b)* **in** ....

rechts: beliebige Ausdrücke, auch Funktions –Ausdrücke und Fallunterscheidungen

nur innerhalb einer Deklaration bekannt

HASKELL

KAPITEL 3.1

Berechnung von Ausdrücken

Erinnerung:

Gegeben sei f (x_1, … , x_n).

f (a_1, … , a_n) mit a_i als aktuellen Parametern – hier Ausdrücken - wird „berechnet“, indem man die a_i an die entsprechenden Stellen der (formalen) Variablen in der Funktionsdefinition einsetzt und dann den Term auflöst.

Bsp:

```
f x y = x + y
```

`f (9-3) (f 34 3)` wird dann zu

`(9-3) + (f 34 3)`

da wir x durch (9-3) und y durch (f 34 3) ersetzen. Beachte, dass die Ausdrücke (9-3) und (f 34 3) nicht ausgewertet werden, bevor sie an die Funktion f übergeben werden!

Erst jetzt (danach) werden die Ausdrücke bezüglich der „Berechnung“ aufgelöst:

6 + (34+3)
6 + 37
43

Im vorherigen Beispiel wurden beide Argumente ausgewertet! Das ist nicht immer der Fall!

Bsp:

```
g x y = x + 12
```

damit führt der Aufruf :

```
g  (9-3) (g 34 3)
```

zu:

```
(9-3) + 12
6 + 12
18
```

D.h. (9-3) wurde für x substituiert, aber da auf der rechten Seite der Funktionsdefinition gar kein y auftaucht, wird auch das Argument (g 34 3) nicht ins Resultat eingebunden, also auch nicht ausgewertet.

Lazy Evaluation (I):

Argumente einer Funktion werden nur ausgewertet, wenn sie zur Berechnung des Gesamtergebnisses der Funktion benötigt werden!

Anmerkung 1:
Sicher wird kaum jemand eine Funktion definieren, deren zweites Argument nie gebraucht wird .....

Bsp 2 (realistischer):

```
switch  :: Int -> a -> a -> a

switch n x y
   | n>0          = x
   | otherwise    = y
```

Hier werden beide Argumente benötigt, aber nur je eines abhängig von `n` ausgewertet.

Anmerkung 2: `a` ist hier umfassendster Typ, für den die Funktion Sinn macht;
siehe später: Typklassen

Lazy Evaluation (II):

Es werden nur die Teile einer Funktion ausgewertet, die für das Gesamtresultat der Funktion nötig sind!

Bsp 3:

```
h x y  = x + x
```
(siehe double in VL Teil 1)

`h (9-3)  (h  34  3)` wird ausgewertet zu:

```
(9-3) + (9-3)     (*)
6 + 6
12
```

In Zeile (*) taucht nun der Ausdruck (9-3) zweimal auf und man wäre versucht zu glauben, dass dieser auch zweimal „berechnet“ / ausgewertet wird.

Das ist nicht der Fall !

HASKELL merkt sich intern bereits ausgewertete Ausdrücke.

Lazy Evaluation (III):

Ein Ausdruck/Argument wird höchstens einmal ausgewertet!

HASKELL

KAPITEL 4

Listen

Typen

Was für Typen verwenden wir?	*Haskell* Typen
natürliche Zahlen	*Integer*
floating point Zahlen	*Float*
Wahrheitswerte	*Bool*
Symbole	*Char*
selbst definierte und abgeleitete Typen	*später*
jetzt neu: Listen	[]

Sei α irgend ein Datentyp. Dann ist $[\alpha]$ der Typ aller Listen mit Werten aus α.

Liste: die wichtigste Datenstruktur

alles was auch Prolog bietet

.... und noch mehr

Beispiele:

$$
\begin{aligned}
[1, 3, 3] &:: [Int] \\
[\texttt{`h`}, \texttt{`a`}, \texttt{`l`}, \texttt{`l`}, \texttt{`o`}] &:: [Char] \\
[(+), (*)] &:: [Int \to Int \to Int] \\
[[1, 2], [3]] &:: [[Int]]
\end{aligned}
$$

keine Liste: [1, +]

weil nicht alle Elemente den selben Typ haben

alternative Darstellungen

induktiver Aufbau von Listen

Sei α irgend ein Datentyp,

seien $a_0, a_1, ..., a_n$ Elemente von α.

[] ist eine Liste

Für $L = [a_1, ..., a_n]$ ist $a_0 : L$ die Liste $[a_0, a_1, ..., a_n]$

Beispiele:

[1, 3, 3] = 1 : [3, 3] = 1 : 3 : [3] = 1 : 3 : 3 : []

Klammern nach Bedarf: .... = 1 : (3 : (3 : []))

Gleichheit

Gleichheit:

$[] == [] = \mathit{True}$

$[] == (y : ys) = \mathit{False}$

$(x : xs) == [] = \mathit{False}$

$(x : xs) == (y : ys) = (x == y) \wedge (xs == ys)$

Sei α irgend ein Datentyp.

$\mathit{null} :: [\alpha] \rightarrow \mathit{Bool}$

$\mathit{null}\ [] = \mathit{True}$

$\mathit{null}\ (x : xs) = \mathit{False}$

Operationen auf Listen

Konkatenation ++ :

[1, 2, 3] ++ [4, 5] ergibt [1, 2, 3, 4, 5]

[1, 2, 3] ++ [] ergibt [1, 2, 3]

formal:

$(++) \quad :: \quad [\alpha] \to [\alpha] \to [\alpha]$

$[] \; ++ \; ys \quad = \quad ys$

$(x : xs) \; ++ \; ys \quad = \quad x : (xs \; ++ \; ys)$

reverse, length

reverse :: $[\alpha] \rightarrow [\alpha]$

reverse [] = []

reverse [*x* : *xs*] = *reverse xs* ++ [*x*]

dann gilt also: *reverse* (*reverse xs*) = *xs*

length :: $[\alpha] \rightarrow$ *Integer*

length [] = 0

length (*x* : *xs*) = 1 + *lenght xs*

dann gilt also:

length (*xs* ++ *ys*) = *length xs* + *length ys*

head, tail, last, init

$head :: [\alpha] \to \alpha$

$head\ (x : xs) = x$

head [] ????

undefiniert, $\perp$

$tail :: [\alpha] \to [\alpha]$

$tail\ (x : xs) = xs$

$last :: [\alpha] \to \alpha$

$last = head \bullet reverse$

$init :: [\alpha] \to [\alpha]$

$init = reverse \bullet tail \bullet reverse$

die ersten n Elemente und was übrig bleibt:

take :: *Integer* $\rightarrow [\alpha] \rightarrow [\alpha]$

take 0 xs = []

take (n+1) [] = []

take (n+1) (x : xs) = *x : take n xs*

drop :: *Integer* $\rightarrow [\alpha] \rightarrow [\alpha]$

drop 0 xs = *xs*

drop (n+1) [] = []

drop (n+1) (x : xs) = *drop n xs*

n-tes Element

$(!!) \quad :: [\alpha] \rightarrow Integer \rightarrow \alpha$

$(x : xs) \; !! \; 0 = x$

$(x : xs) \; !! \; n+1 = xs \; !! \; n$

map

Beispiele:

map square [*9, 3*] *ergibt* [81, *9*]

map (<3) [*1, 2, 3*] *ergibt* [*True, True, False*]

allgemein:

map :: $(\alpha \to \beta) \to [\alpha] \to [\beta]$

map f [] = []

map f (x : xs) = *f x : map f xs*

filter

Beispiele:

filter even [*1, 2, 4, 5, 32*] *ergibt* [2, 4, 32]

allgemein:

filter :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$

filter p [] = []

filter p (x : xs) = **if** *p x* **then** *x* : *filter p xs* **else** *filter p xs*

zip

Beispiele:

zip [*1, 2, 3, 4*] [´*a*´, ´*b*´, ´*c*´, ´*d*´] *ergibt*

[(*1,´a´*), (*2,´b´*), (*3,´c´*), (*4,´d´*)]

allgemein:

zip :: $[\alpha] \rightarrow [\beta] \rightarrow [(\alpha, \beta)]$

zip [] ys = []

zip *xs* [] = []

zip (*x* : *xs*) (*y* : *ys*) = (*x, y*) : *zip* (*xs, ys*)

Konsequenz:

zip [] (*tail* []) = *zip* [] $\perp$ = []

zip [*x*] (*tail* [*x*]) = *zip* [*x*] [] = []

unzip

Operationen auf Paaren:

pair :: $(\alpha \to \beta, \alpha \to \gamma) \to \alpha \to (\beta, \gamma)$

pair (f, g) x = (fx, gx)

fst :: $(\alpha, \beta) \to \alpha$

fst (x,y) = x

snd :: $(\alpha, \beta) \to \beta$

snd (x,y) = y

damit definieren wir:

unzip :: $[(\alpha, \beta)] \to ([\alpha], [\beta])$

unzip = pair (map fst, map snd)

HASKELL

KAPITEL 4.1

Integrierte Listenoperationen – Funktionale auf Listen

(Ergänzung von Prof. Karsten Schmidt)

Funktionale auf Listen

Erster Vorgeschmack auf Funktionen höherer Ordnung

1. Map: (A → B) x [A] → [B]

wendet eine Funktion f elementweise auf eine Liste an

```
m f []  = []
m f (x:xs) = (f x) : (m f xs)
```

In Haskell in die Listenschreibweise integriert:
[f x | x <- list]

```
>>> [x * x | x <- [1,2,3,4,5]]
[1,4,9,16,25]
```

Funktionale auf Listen

2. Zip: (A x B → C) x [A] x [B] → [C]

wendet eine Funktion f elementweise auf mehrere Listen an

```
z f [] [] = []
z f (x : xs) (y : ys) = (f x y) : (z f xs ys)
```

Vordefiniert: zipWith

```
>>> zipWith (+) [1,2,3]  [4,5,6]
[5,7,9]
```

Funktionale auf Listen

3. Filter: (A →Bool) x [A] → [A]

Gibt Liste derjenigen Elemente zurück, die Bedingung erfüllen

```
f b [] = []
f b (x:xs) = (if b x then [x] else []) ++ (f b xs)
```

Haskell: In Listenschreibweise integriert

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

Funktionale auf Listen

4. Reduce: (A x A→A) x [A] → A
oder (A x A → A) x A x [A] → A

Rechnet Operationskette auf Liste aus (mit oder ohne Startwert)

```
r f i [] = i
r f i (x:xs) = f x (r f i xs)
r1 f [x] = x
...
```

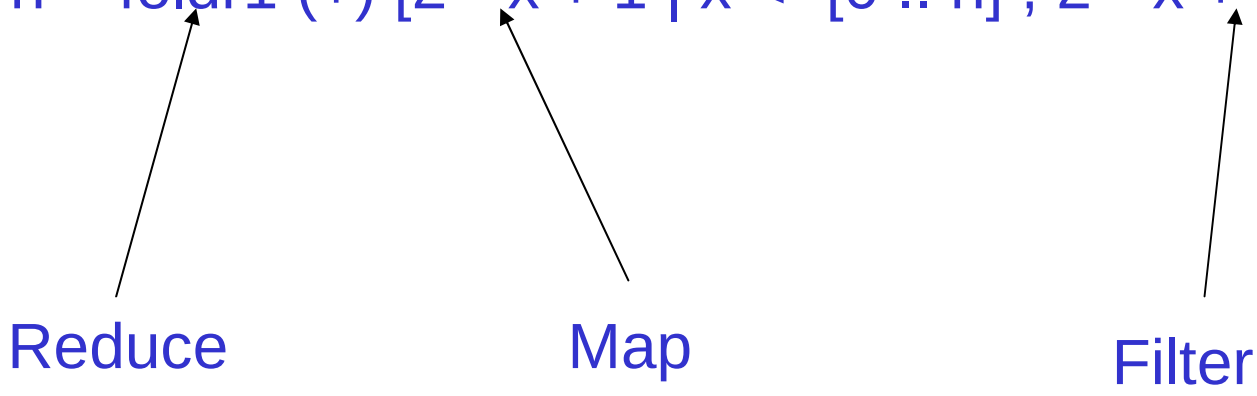
Haskell: vordefinierte foldr und foldr1

```
>>> foldr1 (+) [1,2,3,4,5]
15
>>> foldr (+) 2 [1,2,3,4,5]
17
```

Ein Beispiel:

Summe aller ungeraden Zahlen bis n

```
oddsum n = foldr1 (+) [2 * x + 1 | x <- [0 .. n] , 2 * x + 1 <= n]
```



Noch ein Beispiel : Primzahlen

```
teilbar a b
    | a < b   = False
    | a == b  = True
    | a > b   = teilbar (a - b) b
prim n = foldr (&&) True [not (teilbar n k) |
         k <- [2 .. (n - 1)]]
member y [] = False
member y (x:xs) = (y == x) || member y xs
sieve [] = []
sieve (x:xs) = x : sieve ([k | k <- xs,
   not (member k [a * x | a <- [2 .. length xs]])]
```