

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

6. November 2007

```
partition p []      = ([], [])
partition p (h:t)
  | (p h)           = (h:oks, noks)
  | otherwise      = (oks, h:noks)
                    where (oks, noks) = partition p t
```

```
quicksort []       = []
quicksort (h:t)    = quicksort kleine ++ (h : (quicksort grosse))
                    where (kleine, grosse) = partition (< h) t
```

Alternativ, unter Verwendung von filter:

```
qs []              = []
qs (h:t)           = qs (filter (<= h) t) ++
                    (h : (qs (filter (> h) t)))
```

Verknüpfen aller Elemente einer Liste, z.B.

- Summe aller Elemente einer Liste
- Produkt aller Elemente einer Liste
- Vereinigung aller Mengen (in einer Liste)
- Alle Listen in einer Liste von Listen zusammenhängen.

Dies kann man auf mehrere Arten machen, z.B.

- Summe von $(1, 2, 3, 4)$ in der Form $((1 + 2) + 3) + 4)$
- Summe von $(1, 2, 3, 4)$ in der Form $(1 + (2 + (3 + 4)))$

binäre Operation op
und zugehöriges Anfangselement e :

$foldr\ op\ e\ [] = e$

$foldr\ op\ e\ (h:t) = h\ 'op'\ (foldr\ op\ e\ t)$

Wirkung von `foldr`:

$1 : (2 : (3 : (4 : []))) \rightarrow (1 + (2 + (3 + (4 + 0))))$

wichtiger Spezialfall:

binäre, assoziative Operation op
und zugehöriges Einheitselement e :

Für endliche Listen xs sind gleichwertig:

`foldl f e`

`foldr f e`

und der Typ ist: $(\text{foldl } f \ e) :: [\tau] \rightarrow \tau$
 $(\text{foldr } f \ e) :: [\tau] \rightarrow \tau$

FOLDR: abgeleitete Funktionen

```
summe    = foldr (+) 0
produkt  = foldr (*) 1
concat   = foldr (++) []
```

Auch für unendliche Listen.

nur für **endliche** Listen

$$\text{foldl op e []} = e$$
$$\text{foldl op e (h:t)} = (\text{foldl op (e 'op' h) t})$$

Wirkung von foldl:

$$\text{summe_1} = \text{foldl (+) 0}$$
$$\text{summe_1 [1,2,3,4]} \rightarrow (((((0 + 1) + 2) + 3) + 4).$$

`strict f` ist strikte Variante von `f`

Die neue Version von `foldl` sieht dann so aus:

`foldls op e [] = e`

`foldls op e (h:t) = strict (foldls op) (e 'op' h) t`

foldr :: (a → b → b) → b → [a] → b

foldl :: (a → b → a) → a → [b] → a

Allgemeiner als: (a → a → a) → a → [a] → a

Z.B.:

```
lengthfl = foldl ladd 0
          where ladd x _ = 1+x
```

```
lengthfl = foldl ladd 0
          where ladd x _ = 1+x
```

In diesem Fall sind foldr und foldl nicht austauschbar.

Analog dazu ist:

```
lengthfr = foldr radd 0
          where radd _ x = 1+x
```

Sammeln die sukzessiven Ergebnisse von `foldr` bzw `foldl`.

```
scanl (+) 0 [1,2,3,4] --> [0,1,3,6,10]
```

naive Definition:

```
inits []      = [[]]  
inits (h:t) = []: (map (h:) (inits t))  
  
scanl f e    = map (foldl f e) . inits
```

Nicht effizient: worst case-Komplexität ist ($O(n^2)$),
da für jede Anfangsliste der Wert jedesmal neu berechnet wird.

Eine $O(n)$ - Version ist:

```
scanl f e [] = [e]
```

```
scanl f e (x:xs) = e : (scanl f (f e x) xs)
```

Beispielauswertung (nicht in Normalordnung, sondern auch in der Tiefe):

```
scanl (*) 1 [1,2,3]
--> 1: (scanl (*) (1 * 1) [2,3] )
--> 1: 1 : scanl (*) (1 * 2) [3]
--> 1: 1 : 2 : scanl (*) (2*3) []
--> 1: 1 : 2 : [6]
= [1,1,2,6]
```

Teilsummen zur Berechnung der Eulerschen Zahl e :

$$\text{mittels } e = 1 + 1 + \frac{1}{2} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

```
scanl (+) 0.0 (map (1.0 /) (scanl (*) 1.0 [1.0 ..]))
```

Sukzessive Ergebnisse eines foldr Listentails:

```
tails [] = [[]]
```

```
tails (x:xs) = (x:xs) : tails xs
```

```
scanr f e = (map (foldr f e)) . tails
```

Effizientere Version:

```
scanr f e [] = [e]
```

```
scanr f e (x:xs) = (f x (head ys)) : ys
```

```
    where ys = scanr f e xs
```

```
> scanr (+) 0 [1,2,3,4]  
[10,9,7,4,0]
```

```
> scanr (++) [] (repeat [1..])  
[[1,2,3,4,5,6,7,8,9,10,11,12,
```

```
> map head (scanr (++) [] (repeat [1..]))  
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
```

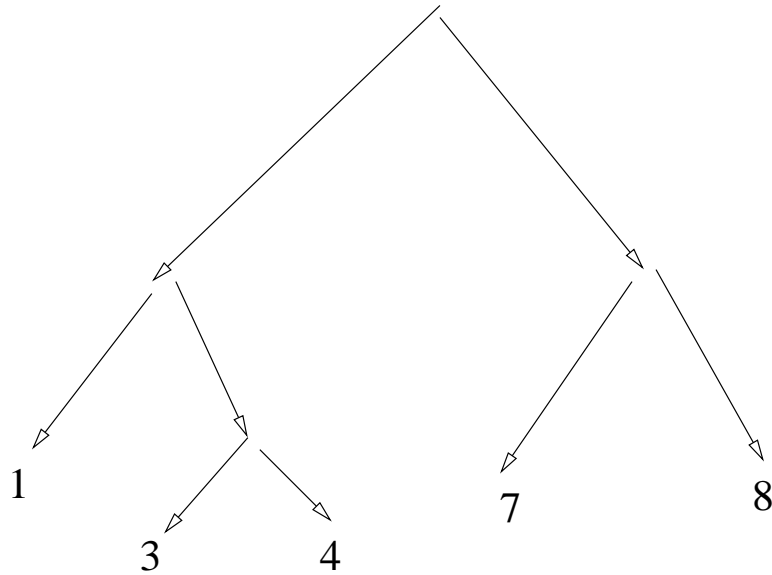
Beispielauswertung (nicht NO)

```
scanr (*) 1 [1,2,3]
--> (1 * (hd ys)):ys   where ys = scanr * 1 [2,3]
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys'
                        where ys' = scanr * 1 [3]
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys'
                        where ys' = 3 * (hd ys''):y'' where ys'' = scanr * 1 []
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys'
                        where ys' = 3 * (hd ys'') :y'' where ys'' = [1]
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys' where ys' = [3,1]
--> (1 * (hd ys)):ys   where ys = [6,3,1]
--> [6,6,3,1]
```

Binäre, geordnete Bäume in Haskell:

```
data Binbaum a = Bblatt a | Bknoten (Binbaum a) (Binbaum a)
```

Der folgende binäre Baum



hat eine Darstellung als

```
Bknoten (Bknoten (Bblatt 1)
                  (Bknoten (Bblatt 3) (Bblatt 4)))
      (Bknoten (Bblatt 7) (Bblatt 8))
```

Rand: Liste der Markierungen der Blätter:

```
b_rand (Bblatt x)      = [x]
```

```
b_rand (Bknoten bl br) = (b_rand bl) ++ (b_rand br)
```

testet, ob Element im Baum ist:

```
b_in x (Bblatt y)      = (x == y)
```

```
b_in x (Bknoten bl br) = b_in x bl || b_in x br
```

wendet eine Funktion auf alle Blätter des Baumes an,
Resultat: Baum der Resultate

```
b_map f (Bblatt x)      = Bblatt (f x)
```

```
b_map f (Bknoten bl br) =  
    Bknoten (b_map f bl) (b_map f br)
```

Größe des Baumes:

$$\text{b_size (Bblatt } x) = 1$$
$$\text{b_size (Bknoten } bl \text{ } br) = 1 + (\text{b_size } bl) + (\text{b_size } br)$$

Summe aller Blätter, falls Zahlen:

$$\text{b_sum (Bblatt } x) = x$$
$$\text{b_sum (Bknoten } bl \text{ } br) = (\text{b_sum } bl) + (\text{b_sum } br)$$

Tiefe des Baumes:

$$\text{b_depth (Bblatt } x) = 0$$
$$\text{b_depth (Bknoten } bl \text{ } br) = \\ 1 + (\max (\text{b_depth } bl) (\text{b_depth } br))$$

schnelles fold über binäre Bäume:

```
foldbt :: (a -> b -> b) -> b -> Binbaum a -> b
```

```
foldbt op a (Bblatt x) = op x a
```

```
foldbt op a (Bknoten x y) = (foldbt op (foldbt op a y) x)
```

foldbt mit optimiertem Stackverbrauch:

```
foldbt' :: (a -> b -> b) -> b -> Binbaum a -> b
```

```
foldbt' op a (Bblatt x) = op x a
```

```
foldbt' op a (Bknoten x y) = (((foldbt' op) $! (foldbt' op a y)) x)
```

effizientere Version von `b_rand` und `b_sum`

```
b_rand_eff = foldbt (:) []
```

```
b_baum_sum' = foldbt' (+) 0
```

Werte `foldbt (+) 0 "(((1,2),3),(4 ,5))"` aus:

```
foldbt (+) 0 "(((1,2),3),(4 ,5))"  
--> foldbt (+) (foldbt (+) 0 (4 ,5))      ((1,2),3)  
--> foldbt (+) (foldbt (+) (foldbt (+) 0 (5)) (4)      ((1,2),3))  
--> foldbt (+) (foldbt (+) (5+0) (4)      ((1,2),3))  
--> foldbt (+) (4+ (5+0))      ((1,2),3)  
--> foldbt (+) (foldbt (+) (4+ (5+0))      (3))      (1,2)  
--> foldbt (+) (3+ (4+ (5+0)))      (1,2)  
--> foldbt (+) (foldbt (+) (3+ (4+ (5+0)))      (2)      (1))  
--> foldbt (+) (2+ (3+ (4+ (5+0))))      (1)  
--> 1+ (2+ (3+ (4+ (5+0))))
```

```
data BinMbaum a m = BMLEER
    | BMblatt m a
    | BMknoten m (BinMbaum a m) (BinMbaum a m)
    deriving (Eq, Show)
```

Beispiel Typ BinMbaum String Integer:

- Inhalt vom Typ `String`
- Knoten und Blätter sind
markiert mit Objekten vom Typ `Integer`

Liste der Inhalte der Blätter

```
b_rand :: BinMbaum a m -> [a]
b_rand BMbleer          = []
b_rand (BMblatt _ x) = [x]
b_rand (BMknoten b bl br) =
    (b_rand bl) ++ (b_rand br)
```

Liste der Markierungen aller Knoten und Blätter: inorder

```
b_mrand :: BinMbaum a m -> [m]
b_mrand BMleer          = []
b_mrand (BMblatt b x) = [b]
b_mrand (BMknoten b bl br) =
    (b_mrand bl) ++ [b] ++ (b_mrand br)
```

```
["aaa", "aa", "aaba", "aab", "aabb", "a", "aba", "ab", "abb"]
```

Liste der Markierungen aller Knoten und Blätter, preorder

```
b_mdfsrand :: BinMbaum a m -> [m]
b_mdfsrand (BMleer) = []
b_mdfsrand (BMblatt b x) = [b]
b_mdfsrand (BMknoten b bl br) =
    b :(b_mdfsrand bl) ++ (b_mdfsrand br)
```

```
["a", "aa", "aaa", "aab", "aaba", "aabb", "ab", "aba", "abb"]
```

Liste der Markierungen aller Knoten und Blätter: postorder

```
b_mporand :: :: BinMbaum a m -> [m]
b_mporand BMleer          = []
b_mporand (BMblatt b x) = [b]
b_mporand (BMknoten b bl br) =
    (b_mporand bl) ++ (b_mporand br) ++ [b]

["aaa", "aaba", "aabb", "aab", "aa", "aba", "abb", "ab", "a"]
```

Wendet Funktionen auf alle Inhalte und Markierungen
des Baumes an. Resultat: Baum der Resultate

```
b_mmap :: (a->c) -> (b -> d) -> (BinMbaum a b) -> (BinMbaum c d)
b_mmap f g BMleer = BMleer
b_mmap f g (BMblatt b x) = BMblatt (f b) (g x)
b_mmap f g (BMknoten b bl br) =
    BMknoten (f b) (b_mmap f g bl) (b_mmap f g br)
```

Fold mit Initialwert für leere Bäume

```
foldmbt  :: (a -> b -> b) -> b -> BinMbaum a b -> b
foldmbt  op e BMleer          = e
foldmbt  op e (BMblatt _ x)  = op x e
foldmbt  op e (BMknoten _ x y) = (foldmbt op (foldmbt op e y) x)
```

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

Vordefinierter Typ.

Zweck: Markieren und Weitergeben von Daten

Nothing	Kein Wert gefunden
Just s	s wurde gefunden und wird weitergegeben

```
b_dfs p baum =  
  b_mgetfirstJust (b_mmap  
                  (\x -> if p x then Just x else Nothing)  
                  id baum)
```

`b_mgetfirstJust` Extrahiert den ersten Just-Eintrag
(in dfs-Reihenfolge)

`b_mmap` erzeugt neuen Baum mit Just x bzw. Nothing-Einträgen
in dfs-Reihenfolge wg. Triggerung durch die Suche

Ergibt Kombination von Funktionen in einem dfs-Durchlauf.

```
testDFS = b_dfs (== "aab") baum13478  
testDFS2 = b_dfs (== "aabb") baum13478
```

Hilfsfunktion

```
b_mgetfirstJust BMleer = (Nothing,Nothing)
b_mgetfirstJust (BMblatt (Just x) r) = (Just x,Just r)
b_mgetfirstJust (BMblatt Nothing r) = (Nothing,Nothing)
b_mgetfirstJust (BMknoten (Just x) _ _) = (Just x,Nothing)
b_mgetfirstJust (BMknoten Nothing bl br) =
    case (b_mgetfirstJust bl) of (Just x,y) -> (Just x,y)
                                (Nothing,_) -> b_mgetfirstJust br
```

Liste der Unterknoten in Breitensuchordnung:

```
b_bslists b = b_bslists_r [b]
```

```
b_bslists_r [] = []
```

```
b_bslists_r xs = xs ++ b_bslists_r (concatMap b_blists1 xs)
```

```
b_blists1 BMleer = []
```

```
b_blists1 (BMblatt m x) = []
```

```
b_blists1 (BMknoten m x y) = [x,y]
```

Breitensuchfunktion:

```
b_bsf p b = b_topmarke (head (filter p (b_bslists b)))
```

Unendlicher Baum der binären Strings:

```
bsplbsf2 x = BMknoten x (bsplbsf2 ('0':x)) (bsplbsf2 ('1':x))
```

Markierungen in Breitensuchordnung:

```
> testbfsfolge2  
["", "0", "1", "00", "10", "01", "11", "000", "100", "010", "110", "001",  
"101", "011", "111",  
"0000", "1000", "0100", "1100", "0010", "1010", "0110", "1110", "0001",  
"1001", "0101", "1101", "0011", "1011", "0111", "1111", ...
```

Breitensuche nach einem bestimmten Knoten
im unendlicher Baum der binären Strings:

```
testbsf3 = b_topmarke (head
  (b_bsf
    (\b -> length (b_topmarke b) == 10 &&
      (head (b_topmarke b) == '1'))
    (bsplbsf2 "")) )
```

```
> testbsf3
"1000000000"
```

mit beliebiger Anzahl Töchter

Die Programmierung ist analog zu einfachen Bäumen

```
data Nbaum a = Nblatt a | Nknoten [Nbaum a]
```

```
nbaumrand :: Nbaum a -> [a]
```

```
nbaumrand (Nblatt x) = [x]
```

```
nbaumrand (Nknoten xs) = concatMap nbaumrand xs
```

Plan: Angabe der Syntax von Haskell; Besonderheiten

Übersetzung der höheren Konstrukte in Kernsprache

Analog zum Compiler, der sukzessive transformiert

(siehe auch (Online-) Handbuch bzw. Haskellreport)

verschiedene Zahlen, Character, rationale Zahlen und Zusatz-Libraries

Es gibt als vordefinierte Typen:

Character 'a', 'A',

Strings sind Listen von Charactern; `String = [Char]`

Listen: `[1, 2, 3]`; abkürzend für `1 : (2 : (3 : Nil))`.

Operatoren können folgende syntaktischen Eigenschaften haben:

Präfix-, Infix-, Postfix-, Prioritäten und Gruppierungseigenschaften

Auch benutzerdefiniert!

Fixity aufheben mittels Einklammern:

$((+) 1 2)$ ist äquivalent zu $(1 + 2)$.

ad-hoc Infix benutzen mit (den richtigen) Anführungszeichen:
'primPlusInt'.

Gruppierung von 2-stelligen Funktionen kann linksassoziativ oder rechtsassoziativ oder weder/noch sein:

$1 - 2 - 3 \equiv (1 - 2) - 3$ aber

$1 : 2 : 3 : [] \equiv 1 : (2 : (3 : Nil))$

```
infixl 7 *
infixl 6 +,-,/, 'div'
infixr 5 ++
infix 4 ==, /= < <=, >=, >
infixr 3 && (logisches und)
infixr 2 || (logisches oder)
```

$1 < 2 < 3$ ergibt einen Syntaxfehler ergeben,
da $<$ nicht gruppierbar ist.

Miranda hat einen Typ *Num*, aus reellen und ganzen Zahlen;
mit automatischer Umwandlung (*coerce*)

Haskell wandelt Zahltypen nicht automatisch ineinander um.
Umwandlung muss programmiert werden
das Typklassensystem unterstützt das.

Ebenso die Typen der Konstanten: zB $5 :: (\text{Num } t) \Rightarrow t$

Dasselbe Symbol wird für **verschiedene** Operationen benutzt

z.B. $+$ für reelle und integer-Addition.

Überladungsauflösung zur Compilezeit und Laufzeit
abhängig vom Programm-Kontext
und den Fähigkeiten des Typcheckers / Compilers

z.B. $(2 +)$ ist die Funktion $\lambda x.2 + x$,

ebenso: $(/ 2)$ ist die Funktion $\lambda x.x / 2$,

Ausnahme: $(-x)$ ist $\lambda x.(-x)$

werden vom Haskell-Parser beachtet,

Es werden Klammern “{“ bzw. “}” eingefügt und “;”

je nach Einrückung, Konstrukt und entsprechender Layout-Regel

Ergänzung von Patterns durch **Wächter (guards)**

Fakultät mit guards:

```
fac 0          = 1
fac n | n > 0 = n * fac (n - 1)
```

$n > 0$ ist hier ein Wächter (guard).

Es gibt eine eingebaute Funktion `show`,
die alle Ausdrücke (der Typklasse `Show`) auf Strings abbildet.

d.h. $show : a \rightarrow \text{String}$.

Umkehrfunktion `read`

der Typ muss ermittelbar sein!:

Beispiele:

```
>read "1.23"  
*** Exception: Prelude.read: no parse  
Prelude> (read "1.23")::Double  
1.23  
> (read "123")  
123  
> (read "'a'")::Char  
'a'  
> (read "\"abc\"")::String  
"abc"  
(read "()")::()  
()
```

Schachtelung der Pattern ist möglich
Patternvariablen sind alle verschieden
“_” ist Joker-variable

`f [x,y] = x+y` ist möglich.

Beispiel

```
remdup [] = []  
remdup [x] = [x]  
remdup (x:(y: z)) | x == y = remdup (y:z)  
                  | otherwise = x : (remdup (y:z))
```

Listenpattern: $[x, y, z]$ ist eine Liste mit drei Elementen, die für $x : y : z : []$ steht .

markierte Pattern: $p@(x, y)$. p referenziert das ganze Paar

$m + 1$ -**Pattern** matcht positive Werte und berechnet den um 1 kleineren Wert

Haskell kann bei Laufzeitfehlern Meldungen ausgeben:
Das sind **keine** Werte:

Beispiele

- `1/0` arithmetischer Fehler
- `hd []` Pattern fehlt
- `(error "falsche")` benutzerdefinierter Error
- erkannte Nichtterminierung (black hole)

Semantisch sind alle diese Fehler äquivalent ($= \perp$).

Transformation von Haskell nach KFPTS :

- arithmetische Datentypen als Peanozahlen und Character algebraischen Datentyp
- `show` und `read`: (kommt nach Typklassen)
- Tupel, Listen, `if-then-else`: leicht transformierbar
- Kombinatordefinition mit mehreren Gleichungen:
Patterns von oben nach unten: sequentielles Matchen
Innerhalb eines Pattern von links nach rechts.

Übersetzung mittels `case`; Evtl. geschachtelt.

Die Strategie der Patternverarbeitung muss eindeutig festgelegt sein:

Die Schachtelung der cases legt die Semantik fest!.

Beispiel

```
f x [] = ...  
f x (y:ys) = ...
```

case nur über das 2.te Argument, nicht über das erste.

- Guards: kein Problem
zu beachten Beim Fehlschlagen eines guards wird das nächste Pattern versucht.
- Rekursive Definitionen mittels `let` oder rekursive Superkombinatoren:
Hierzu kann man Fixpunktkombinatoren verwenden, z.B.
$$Y = \lambda f. (\lambda x. f(x\ x)) (\lambda x. f(x\ x))$$

(Siehe ein späteres Kapitel)
- Ein nichtrekursives `let` ist übersetzbar:
$$\text{let } x = s \text{ in } t \equiv ((\lambda x. t) s).$$
- `strict`: kann nach $\text{KFPTS} + \text{seq}$, nach $\text{KFPT} + \text{seq}$, bzw. nach KFP übersetzt werden.

Diese Transformationsregeln nach KFPTS legen in Haskell fest:

1. Die Semantik der Funktionen und Konstrukte in Haskell
2. Die Gültigkeit der kontextuellen Gleichheit und die Korrektheit von Programmtransformationen.

List Comprehensions auch ZF-Expressions (Zermelo & Fränkel).

Analogie:

$\{x * x \mid x \in [1..100], x > 5\}$	Menge; ohne Reihenfolge
$[x * x \mid x \leftarrow [1..100], x > 5]$	Liste; mit Reihenfolge

Zwei Äquivalenzen:

$$[f\ x \mid x \leftarrow l] \equiv \text{map } f\ l$$

$$[x \mid x \leftarrow l, p\ x] \equiv \text{filter } p\ l$$

$[exp \mid g_1, \dots, p_j, \dots]$

exp ist Ausdruck für die Elemente der erzeugten Liste

g_i Generator $x \leftarrow t$

p_j Guard; ist Ausdruck vom Typ Bool.

Beachte Geltungsbereich der Variablen
ist rechts vom Generator in der Generator/ Guard-Liste
und im Ausdruck exp .

Beispiel

$[(x, y) \mid x \leftarrow [1..6], y \leftarrow [1..6], x + y < 5]$

→

$[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (3, 1)]$

Beispiele

```
nblanks n      = [' ' | x <- [1..n]]
```

```
kartprod xs ys = [(x,y) | x <- xs, y <-ys]
```

kartprod [1,2] ['a', 'b'] ergibt

```
[(1,'a'), (1,'b'), (2,'a'), (2,'b')].
```

Transformation von List Comprehensions nach
Haskell ohne List Comprehensions

Regeln zur Transformation (auf der Syntax)

ZFgen: $[e|x \leftarrow xs, Q]$ = `concatMap (\x -> [e|Q]) xs`
ZFguard: $[e|p, Q]$ = `if p then [e|Q] else []`
ZFnil: $[e|]$ = `[e]`

Diese Regeln werden wiederholt auf $[e|Q]$ angewendet

`[e|x <- xs] → concatMap (\x -> [e]) xs`

das ist äquivalent zu `(map (\x -> e) xs)`.

optimierte Basisfälle mit `map` und `filter`

`[e|x <- xs] = (map (\x -> e) xs)`

`[e|x <- xs,p] = filter p xs`

kartprod: $[(x,y) \mid x \leftarrow xs, y \leftarrow ys]$.

$$\begin{aligned} & [(x,y) \mid x \leftarrow xs, y \leftarrow ys] \\ \equiv & \text{concatMap } (\backslash x \rightarrow [(x,y) \mid y \leftarrow ys]) \text{ } xs \\ \equiv & \text{concatMap } (\backslash x \rightarrow (\text{map } (\backslash y \rightarrow (x,y)) \text{ } ys)) \text{ } xs \end{aligned}$$

```
concatMap (\x -> (map (\y -> (x,y)) [4,5,6])) [1,2,3]
```

und

```
[(x,y) | x<-[1,2,3], y<-[4,5,6]]
```

ergeben dieselbe Liste

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

Die List-Comprehension-Übersetzung terminiert

Sie kann auch als Grundlage für den Typcheck verwendet werden

Alternative: eigene Typregeln für List Comprehensions

Beispiel: $[(x,y) \mid x \leftarrow xs, y \leftarrow ys]$

und x kommt nicht in ys vor

Eine direktere, (vermutlich effizientere) Übersetzung:

```
kp xs ys = kpr xs ys ys
```

```
kpr [] ys zs = []
```

```
kpr (x:xs) [] zs = kpr xs zs zs
```

```
kpr xs@(x:_) (y:ys) zs = (x,y):(kpr xs ys zs)
```

Pattern in Generatoren:

z.B. `[x | (x:h) <- [[], [1]]]` ergibt `[1]`.

Die Übersetzung für den einfachsten Fall ist:

ZF1* `[exp | pat <- lst]`:

```
filterPat [] = []
```

```
filterPat ((x:h):xs) = (x*x):filterPat xs
```

```
filterPat (_:xs) = filterPat xs
```

Wobei `pat = (x:h)` und `exp = x*x` im Bspl.

Allgemeine Übersetzung:

```
[e|pat <- xs,Q] =
```

```
concat (filterPat xs)
```

```
  where filterPat []           = [ ]
```

```
        filterPat (pat:xs)    = [e|Q]:filterPat xs
```

```
        filterPat (_:xs)      = filterPat xs
```

```
zfex1 = [x*x | (x:_) <- [[], [1], [2], [3], []], x >= 2]
```

```
zfex2 = concat (filterPat [[], [1], [2], [3], []])  
  where filterPat [] = []  
        filterPat ((x:_) : xs) = [x*x | x >= 2] : filterPat xs  
        filterPat (_ : xs) = filterPat xs
```

```
*Main> zfex1
```

```
[4,9]
```

```
*Main> zfex2
```

```
[4,9]
```

Vorstellung: Suchalgorithmen, die baumartig arbeiten
kein Ergebnis: [] als Resultat
ein / mehrere Ergebnisse:
 Liste der Ergebnisse als Resultat
Diese werden auf jeder Stufe der Suche
wieder kombiniert zur neuen Ergebnisliste

Beispiel concatMap (flache Suche)

Beispiel Parser mit Backtracking (tiefe Suche)

Kombination von Parsern:

Typ Parser s a :

- s Eingabetyp (zB Token)
- a Ausgabetyt der Elemente

Alternativ-Kombinator : Muss Ergebnisse vereinigen:

$(\langle | \rangle) :: \text{Parser } s \ a \ \rightarrow \ \text{Parser } s \ a \ \rightarrow \ \text{Parser } s \ a$
 $(p1 \ \langle | \rangle \ p2) \ xs = p1 \ xs \ ++ \ p2 \ xs$

Sequenz-Kombinator von Parsern:

Muss auf allen Ergebnissen des ersten Parsers
den Folgeparser anwenden:

Ergibt Kreuzprodukt: Liste der Paare der Ergebnisse

```
(<*>) :: Parser s a -> Parser s b -> Parser s (a,b)
(p1 <*> p2) xs = [(xs2, (v1,v2))           -- Reststring und Ergebnis
                  | (xs1,v1) <- p1 xs,     -- Aufruf erster Parser
                    (xs2,v2) <- p2 xs1]   -- Aufruf zweiter Parser
```