Kapitel 3: Programmierung mit Kombinatoren

Lernziele dieses Kapitels

1. Denken in funktionalen Komponenten, Vermeidung von
   - Kontrollflussdenken (Rekursion)
   - Details der Datenstruktur (Pattern Matching in induktiven Definitionen)
2. Selbstverständlicher Umgang mit vielen "$\rightarrow$" in den Typsignaturen
3. Kenntnis der wichtigsten Kombinatoren
   - Bildliche Vorstellung ihrer Struktur
   - Anwendung von Kombinatoren
4. Bäume: Durchläufe und Suche
5. `Functor` als Typklasse für Abstraktion von Datenstrukturen
6. Skelette, Beispiel Divide-and-Conquer

Programmierung mit Kombinatoren

- Kombinator: Funktion ohne freie Variablen
- Bedeutung unabhängig von der Umgebung
- klare Funktionalität
- hohe Wiederverwendbarkeit
- geeignet für spezielle effiziente Implementierung
- Skelett: Kombinator mit umfangreicher Funktionalität

Kombinatoren statt expliziter Rekursion

Beispielaufgabe: revertieren einer Liste

- schlechte Lösung mit expliziter Rekursion

```
revertiere l
  = if null l
       then []
       else let rest       = revertiere (tail l)
                ansEnde l1 = if null l1
                                then [head l]
                                else head l1 : ansEnde (tail l1)
            in ansEnde rest
```

- gute Lösung mit Kombinatoren

```
revertiere = foldl (flip (:)) []
```

Aspekt Programmierstil

einige der wünschenswerten Eigenschaften ...

1. Vermeidung impliziter Definitionen (Rekursion)
2. Vermeidung von Fallunterscheidungen (if-then-else)
3. Bezug auf Datenstrukturen so gering wie möglich halten
4. Wenn Datenstrukturen, dann Pattern Matching statt Selektoren
5. Funktionsgleichungen ohne Bezug auf Argumente ($\rightarrow$ pointfree style)

... und warum diese wünschenswert sind:

- (1),(2),(4),(5): Vereinfachung von Beweisen und Programmtransformationen
- (1),(2),(3),(4): Verkürzung des Quelltextumfangs $\Rightarrow$ bessere Lesbarkeit
- (3): Austauschbarkeit von Teilen der Implementierung

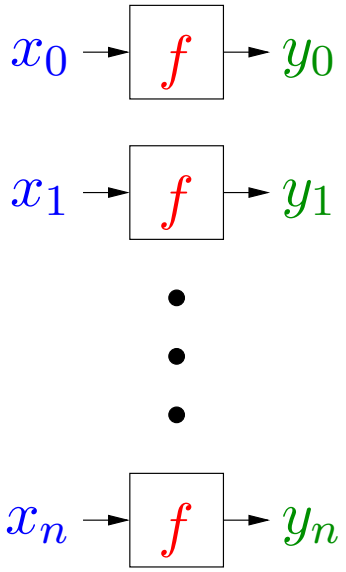
Der Nutzen von Kombinatoren

- Bereitstellung häufig vorkommender Programmschemata (`map`, `foldl`, ...)
- Kapselung von Rekursion und Fallunterscheidungen
- Anpassung von Repräsentationen (`flip`, `concat`, ...)
- Skelette: Faktorisierung einer Problemlösung in
  - algorithmischen Anteil (Depth-First-Suche, Divide-and-Conquer, ...)
  - Anwendungs-spezifische Operationen
- Einfache Regeln für Programmtransformationen
  z.B. `map g . map f = map (g . f)`

Effiziente Implementierung für Kombinatoren

Der Compiler kann Wissen über den Kombinator ausnutzen.

Bsp. `map` $f\ [x_0, ..., x_n] \rightsquigarrow [y_0, ..., y_n]$:



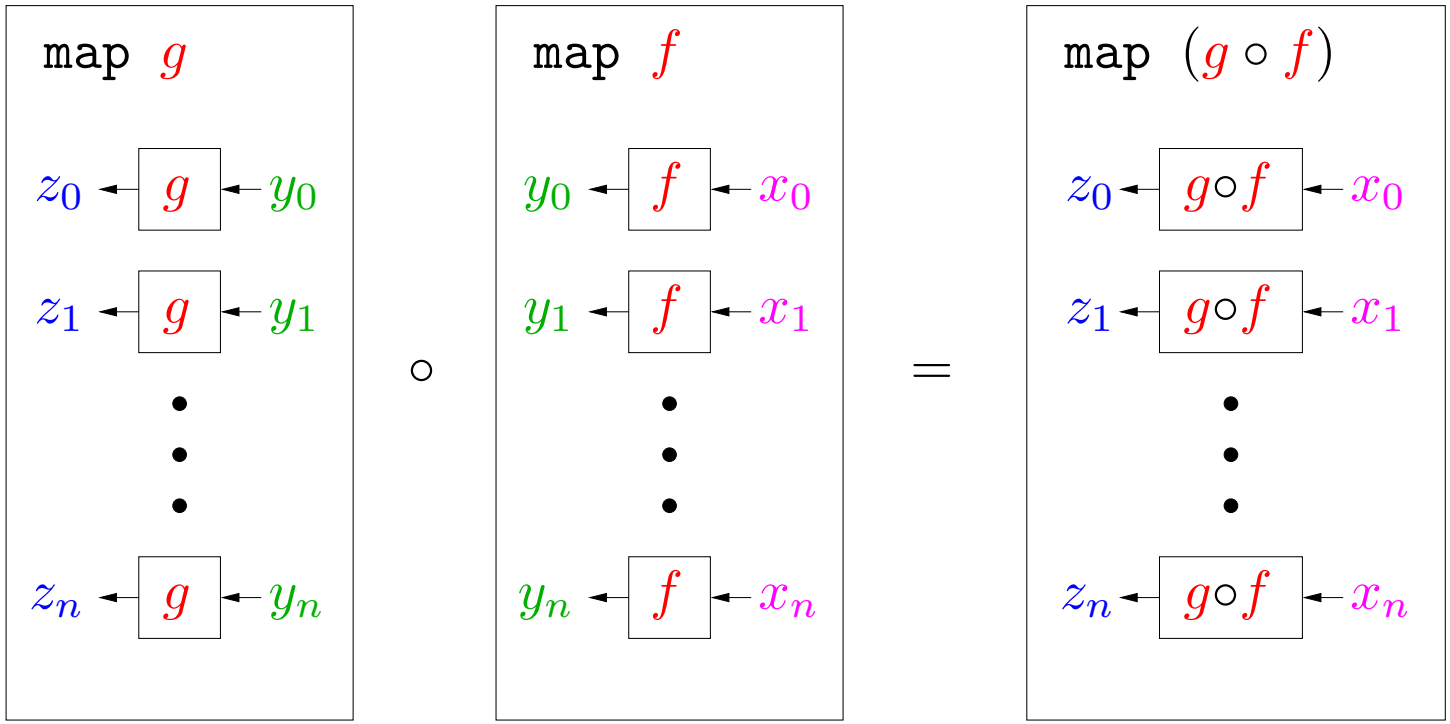
Denken in funktionalen Blöcken,
nicht in Rekursion

Fazit: alle Anwendungen von f sind unabhängig und könnten,
z.B. von einem Parallelrechner gleichzeitig ausgeführt werden.

Programmtransformationen

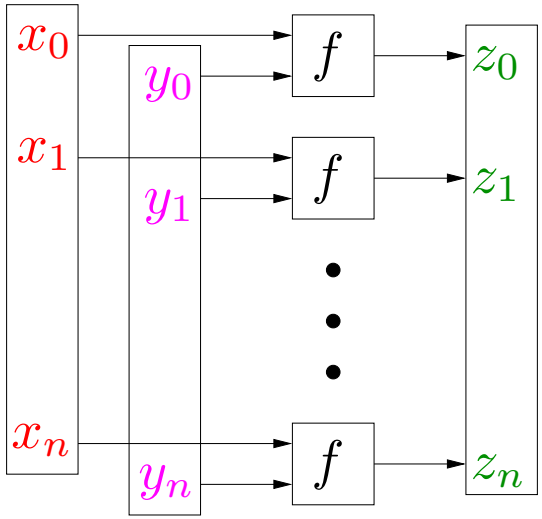
Bsp.: `map` g `.` `map` f `==` `map` `(`g `.` f`)`

- sequenziell: Vermeidung von Iterationen und temporären Datenstrukturen (y)
- parallel: Vermeidung von Kommunikationen



Vektoroperationen: zipWith

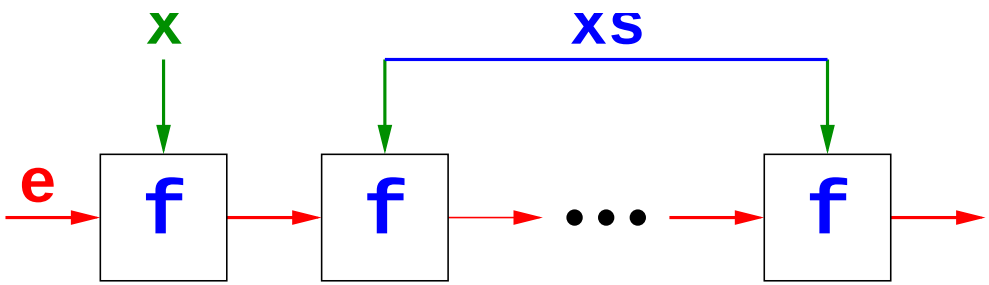
```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _ = []
```




```
innerProd :: Num a => [a] -> [a] -> a
innerProd xs ys = sum (zipWith (*) xs ys)
```

Der `foldl`-Kombinator

```
foldl :: (a->b->a) -> a -> [b] -> a
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```



Beispiele:

```
sum  [1,2,3,4] = foldl (+) 0 [1,2,3,4] = (((0+1)+2)+3)+4
prod [1,2,3,4] = foldl (*) 1 [1,2,3,4] = (((1*1)*2)*3)*4
```

Wiederverwendbarkeit von Kombinatoren

naive Definition

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

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

reverse xs = rev xs []
 where
  rev []     acc = acc
  rev (x:xs) acc = rev xs (x:acc)
```

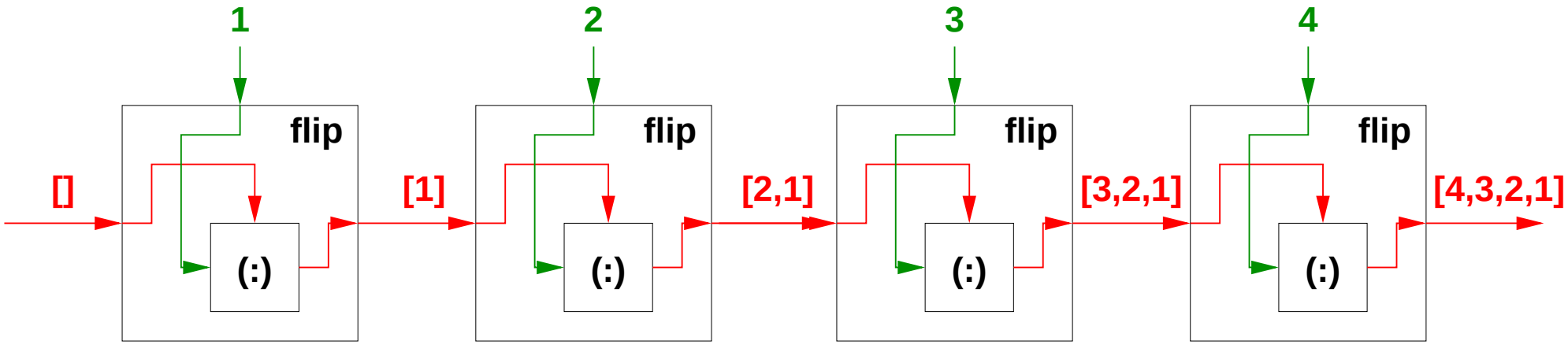
elegante Definition

```
sum = foldl (+) 0

product = foldl (*) 1

reverse
 = foldl (flip (:)) []
```

Listenrevertierung mittels `foldl`

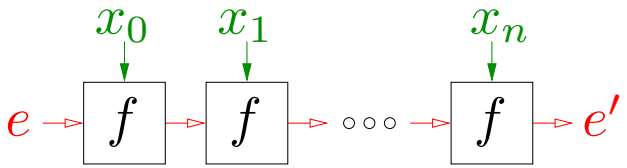
```
reverse = foldl (flip (:)) []
```

```
reverse [1,2,3,4] = foldl (flip (:)) [] [1,2,3,4]
                  = foldl (flip (:)) [1]  [2,3,4]
                  = foldl (flip (:)) [2,1]  [3,4]
                  = foldl (flip (:)) [3,2,1]  [4]
                  = foldl (flip (:)) [4,3,2,1] []
                  = [4,3,2,1]
```

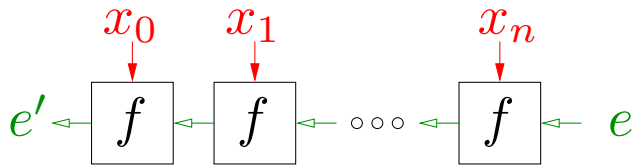


Varianten des `fold`-Kombinators

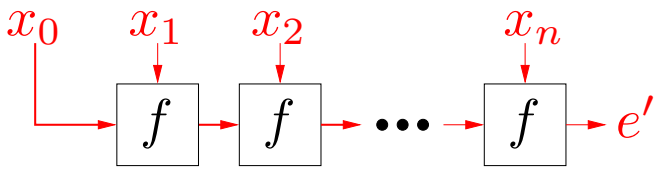
```
foldl :: (a->b->a)->a->[b]->a
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```



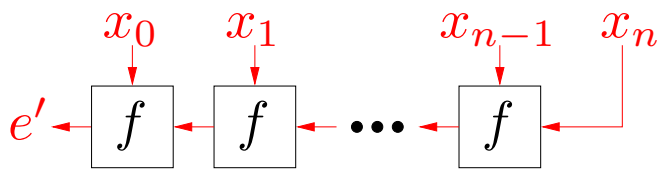

```
foldr :: (a->b->b)->b->[a]->b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```



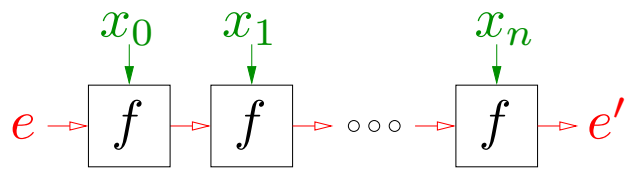

```
foldl1 f (x:xs) = foldl f x xs
```




```
foldr1 f [x]     =  x
foldr1 f (x:xs)  = f x (foldr1 f xs)
```



Anwendung von `foldl`, Bsp. `sum`

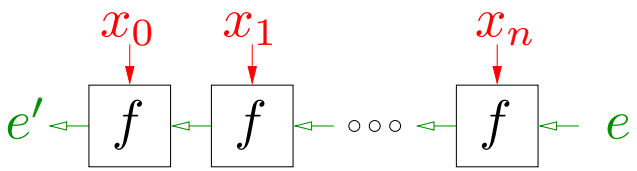



```
foldl :: (a->b->a) -> a -> [b] -> a
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

```
sum = foldl (+) 0
sum [a,b,c,d,e]  ⇝  ((((0+a)+b)+c)+d)+e
```

- Da `+` assoziativ ist, wäre auch `foldr` möglich.
- Für strikte Operatoren wie `+` bevorzugt man `foldl`, weil
  1. die Zahlen in der Liste nicht extra gespeichert werden müssen und
  2. sowieso die ganze Liste durchgegangen werden muss.

Anwendung von `foldr`, Bsp. `and`

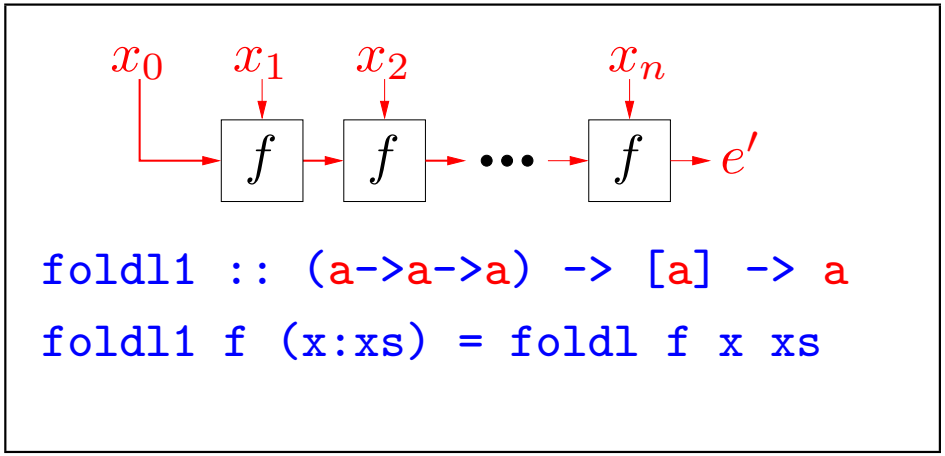



```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f e [] = []
foldr f e (x:xs) = f x (foldr f e xs)
```

```
and = foldr (&&) True
and [a,b,c] ⇝  a && (b && (c && True))
```

- Da `&&` assoziativ ist, wäre auch `foldl` möglich.
- Für im zweiten Argument nicht-strikte Operatoren wie `&&` bevorzugt man `foldr` weil
  1. `foldl` die ganze Liste durchlaufen würde, aber
  2. das Ergebnis bereits früher feststehen kann.

Anwendung von `foldl1`, Bsp. `maximum`



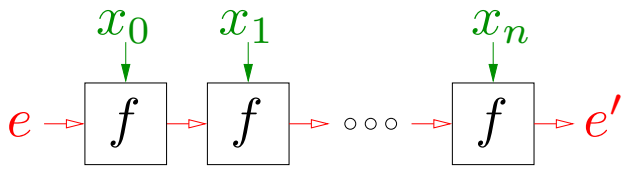

```
foldl1 :: (a->a->a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

```
maximum = foldl1 max
maximum [3,5,7,4] ⇝  max (max (max 3 5) 7) 4
```

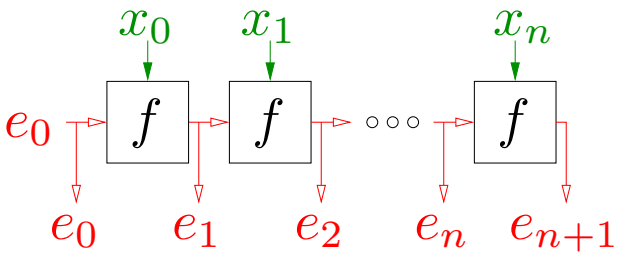
- `foldl1`, weil oft kein neutrales Element vorhanden ist.
- Nur noch eine Variable im Typ.
- Da ‘`max`‘ assoziativ ist, wäre auch `foldr1` möglich.
- Analog zu `sum`: `foldl1`, weil `max` strikt.

Zwischenergebnisse: `scanl` und `mapAccumL`

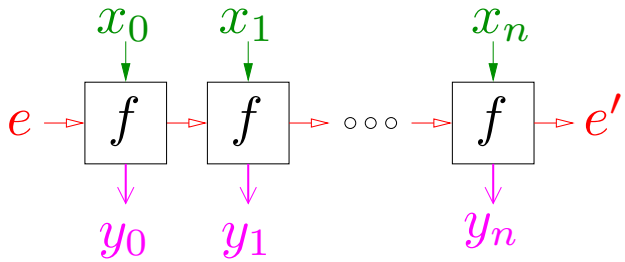
```
foldl :: (a->b->a)
         -> a -> [b] -> a
```




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




```
mapAccumL ::
 (a->x->(a,y))
 -> a -> [x] -> (a,[y])
```



analog: `scanr`, `scanl1`, `scanr1` und `mapAccumR`

Berechnung von Zielpositionen für eine Teilfolge

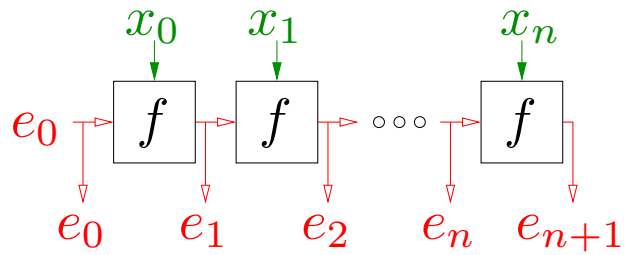
Anwendung: paralleles Partitionieren (paralleler Quicksort)

- gegeben: Prädikat `(>4)` und Liste `[2,7,6,4,8,2]`
- gesucht: Positionen in der Teilfolge der Zahlen, die das Prädikat erfüllen

```
Eingabe:   [2,7,6,4,8,2]
Maske:     [0,1,1,0,1,0]     1 gdw. Prädikat erfüllt
Offsets:   [0,0,1,2,2,3,3]   Präfixsumme von Maske (scanl (+) 0)
Ergebnis:  [0,0,1,2,2,3]
```

- `Ergebnis` ist elementweise Verknüpfung (`zipWith`) aus `Maske` (Farbe) und `Offsets` (Wert), kleinere Liste bestimmt Länge
- Verwende statt Farbe `Nothing` und `Just` im Ergebnis:
  `[Nothing,Just 0,Just 1,Nothing,Just 2,Nothing]`

Anwendung von `scanl`, Bsp.: `positions`

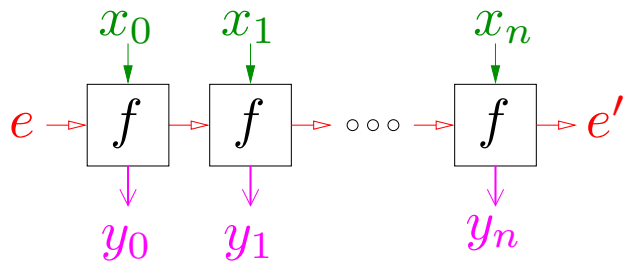



```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
```

```
positions :: (a->Bool) -> [a] -> [Maybe Int]
positions pred xs =
 let mask    = [ if pred x then 1 else 0 | x<-xs ]    [0,1,1,0,1,0]
     offsets = scanl (+) 0 mask                       [0,0,1,2,2,3,3]
     f m o   = if m==0 then Nothing else Just o
 in zipWith f mask offsets
```

Anwendung von `mapAccumL`, nochmal: `positions`

Vorteil: nur ein Listendurchlauf, keine temporären Datenstrukturen




```
mapAccumL :: (a->x->(a,y)) -> a -> [x] -> (a,[y])
```

```
positions :: (a->Bool) -> [a] -> [Maybe Int]
positions pred xs
 = let f acc x = if pred x then (acc+1,Just acc)
                           else (acc  ,Nothing)
   in snd (mapAccumL f 0 xs)
```

Kombinatoren auf Bäumen

- Baumtyp (eine Art): `data Tree element = Node element [Tree element]`
- Beispiel:

```
tree1 :: Tree String
tree1
  = Node "" [Node "a" [Node "aa" [],
                       Node "ab" [Node "aba" [],
                                  Node "abb" [Node "abba" [],
                                              Node "abbb" [],
                                              Node "abbc" []],
                                  Node "abc" []]
                      ],
             Node "b" [Node "ba" [],
                       Node "bb" [Node "bba" []]],
             Node "c" [Node "ca" [Node "caa" []]]]
```

Anwendung einer Funktion auf alle Knoten

- Definition

```
mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Node x subtrees) = Node (f x) (map (mapTree f) subtrees)
```

- Beispiel

```
*Tree> mapTree length tree1
Node 0 [Node 1 [Node 2 [],
                Node 2 [Node 3 [],
                        Node 3 [Node 4 [],
                                Node 4 [],
                                Node 4 []],
                        Node 3 []]],
        Node 1 [Node 2 [],
                Node 2 [Node 3 []]],
        Node 1 [Node 2 [Node 3 []]]]
```

Reduktion (fold) auf Bäumen

- Definition

```
foldTree :: (a->[b]->b) -> Tree a -> b
foldTree f (Node x subtrees) = f x (map (foldTree f) subtrees)
```

- Bsp.: Summierung aller Zahlen in einem Baum

```
sumElems :: Num a => Tree a -> a
sumElems = let f nodeVal subTreeSums = nodeVal + sum subTreeSums
           in foldTree f

*Tree> let tree2 = mapTree length tree1
*Tree> sumElems tree2
40
```

Tiefendurchlauf von Bäumen

```
preOrder, inOrder, postOrder ::  Tree a -> [a]

preOrder = let pre x xss = x : concat xss
           in foldTree pre

postOrder = let post x xss = concat xss ++ [x]
            in foldTree post

inOrder = let second x [] = [x]
              second x (xs:xss) = xs ++ (x : concat xss)
          in foldTree second
```

Breitendurchlauf von Bäumen

Hilfsmittel: algorithmisches Skelett `workQueue`

```
workQueue :: (a->([b],[a])) -> [a] -> [b]
workQueue f xs = wQ f xs [] where
 wQ f []     acc = acc
 wQ f (x:xs) acc = let (out,app) = f x
                   in wQ f (xs++app) (acc++out)
```

Implementierung des Breitendurchlaufs `breadthOrder`

- hänge aktuellen Knotenwert an Ergebnisliste `acc` an
- hänge Unterbäume an die Warteschlange `xs` an

```
breadthOrder :: Tree a -> [a]
breadthOrder t = workQueue f [t]
 where f (Node x subtrees) = ([x],subtrees)
```

Suche in Bäumen

1. erzeuge eine Liste mit den Elementen in der gewünschten Ordnung
2. suche das erste Element in der Liste, welches das Prädikat `pred` erfüllt

```
depthFirstSearch, breadthFirstSearch :: (a->Bool) -> Tree a -> Maybe a

depthFirstSearch   pred = maybeHead . filter pred . preOrder
breadthFirstSearch pred = maybeHead . filter pred . breadthOrder
```

> Wegen der Laziness von Haskell wird die Datenstruktur nur soweit durchlaufen, bis das entsprechende Element gefunden wurde.

```
maybeHead :: [a] -> Maybe a
maybeHead []    = Nothing
maybeHead (x:_) = Just x
```

```
*Tree> preOrder tree1
["","a","aa","ab","aba","abb","abba","abbb","abbc","abc",
 "b","ba","bb","bba","c","ca","caa"]
*Tree> postOrder tree1
["aa","aba","abba","abbb","abbc","abb","abc","ab","a","ba",
 "bba","bb","b","caa","ca","c",""]
*Tree> inOrder tree1
["aa","a","aba","ab","abba","abb","abbb","abbc","abc","",
 "ba","b","bba","bb","caa","ca","c"]
*Tree> breadthOrder tree1
["","a","b","c","aa","ab","ba","bb","ca","aba","abb","abc",
 "bba","caa","abba","abbb","abbc"]
*Tree> let pred x = length x > 0 && last x == 'b'
*Tree> depthFirstSearch pred tree1
Just "ab"
*Tree> breadthFirstSearch pred tree1
Just "b"
```

Verallgemeinerung von `map`: `fmap`

- `map` und `mapTree` unterscheiden sich nur in der Datenstruktur (Liste/Baum).
- Typkonstruktoren für Listen (`[]`), Bäume (`Tree`) und andere Datenstrukturen sind sogenannte Funktoren ($\rightarrow$ Kategorientheorie).
- in Haskell gibt es eine Typklasse für Funktoren:

```
class Functor f where
   fmap :: (a -> b) -> (f a -> f b)
```

- Instanzen von `Functor` sollten folgende Bedingungen erfüllen:

(1) `fmap id = id`

(2) `fmap (f . g) = fmap f . fmap g`

Instanzen von Functor

- Listen

```
instance Functor [] where
   fmap = map
Prelude> fmap (^2) [2,3,4]
[4,9,16]
```

- Bäume

```
instance Functor Tree where
   fmap = mapTree
*Tree> fmap (^2) (Node 3 [Node 4 [], Node 5 []])
Node 9 [Node 16 [],Node 25 []]
```

- Maybe

```
instance Functor Maybe where
   fmap f Nothing  = Nothing
   fmap f (Just x) = Just (f x)
Prelude> fmap (^2) (Just 3)
Just 9
```

- Arrays

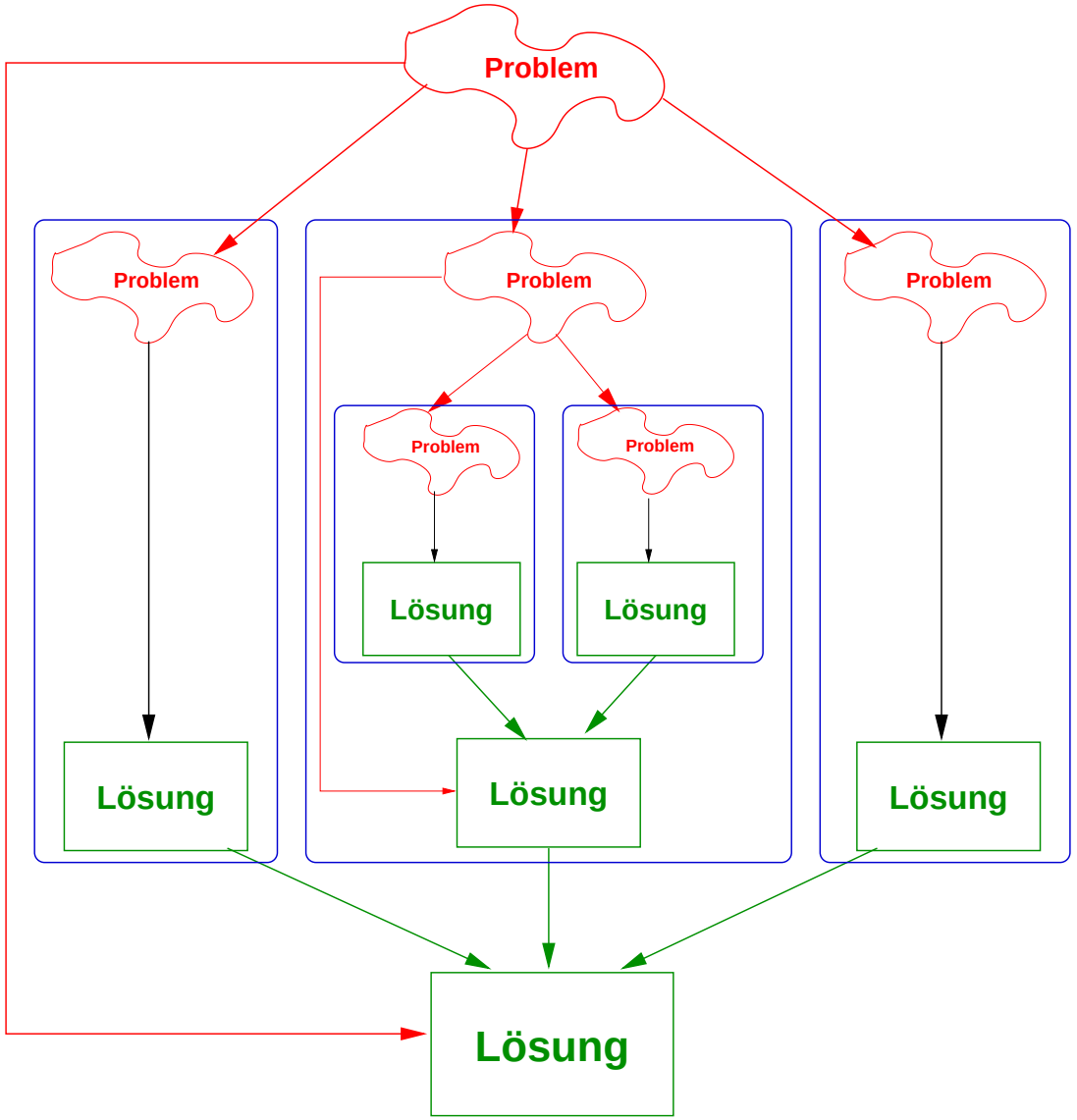
```
instance (Ix a) => Functor (Array a) where        (a:Indexmenge)
   fmap fn (MkArray b f) = MkArray b (fn . f)
Prelude Array> fmap (^2) (array ((0,0),(1,1))
                        [((0,0),3),((0,1),6), ((1,0),5), ((1,1),2)])
array ((0,0),(1,1)) [((0,0),9),((0,1),36),((1,0),25),((1,1),4)]
```

Das Skelett Divide-and-Conquer (1)

Paradigma Divide-and-Conquer:

- falls das Problem einfach ist, löse es direkt
- sonst
  1. teile das Problem in unabhängige Teile gleichen Typs
  2. wende das Verfahren rekursiv auf jedes Teilproblem an
  3. kombiniere die Lösungen der Teile zur Gesamtlösung

Das Skelett Divide-and-Conquer (2)



Das Skelett Divide-and-Conquer (3)

```
dc :: (a->Bool)->(a->b)->(a->[a])->(a->[b]->b)->a->b
dc p b d c = dcprg
  where dcprg x = if p x
                  then b x
                  else c x (map dcprg (d x))
```

Instanzierung mit problemspezifischen Funktionen:

1. `p::(a->Bool)` stellt fest, ob das Problem einfach ist
2. `b::(a->b)` löst das Problem direkt
3. `d::(a->[a])` teilt das Problem in eine Liste von Teilproblemen
4. `c::(a->[b]->b)` verbindet die Teillösungen

`dcprg::(a->b)` ist das resultierende Programm

Anwendung von dc (1): funktionales "quicksort"

```
quicksort :: Ord a => [a] -> [a]
quicksort = dc p b d c
 where p xs     = length xs < 2
       b xs     = xs
       d (x:xs) = let (a,b) = partition (<x) xs
                  in [a,b]
       c (x:_) [a,b] = a ++ (x : b)
```

Anwendung von dc (2): Damenproblem

- Eingabe: Anzahl der Zeilen/Spalten des "Schach"bretts
- Ausgabe: Liste aller Lösungen, bei jeder Lösung: Element i der Liste enthält Spaltenposition der Dame, die in Zeile i steht

```
queens :: Int -> [[Int]]
queens n = dc p b d c ([],[0..n-1]) where
   p (_,remain) = null remain
   b (placed,_) = [placed]
   d (placed,remain)
     = [ (placed++[i],filter (/=i) remain)
       | i <- remain, not (diagonal_attack i) ]
       where diagonal_attack i
                = or [ length placed - j  ==  abs (i - placed!!j)
                     | j<-[0..length placed -1] ]
   c _ = concat
```

Skelett ramp (reduce-and-map-over-pairs)

beschreibt wechselseitigen Einfluß von Objekten

```
ramp :: (a->a->b) -> (b->b->b) -> [a] -> [b]
ramp f g xs = map h xs
              where h x = foldr1 g (map (f x) xs)
```

Anwendung: Simulation eines Systems von Planeten (nBody)

```
nBody :: [Planet] -> [[Planet]]
nBody ps = ps : nBody (map newPos (zip ps (ramp calcF sumFs ps)))

newPos :: (Planet,Force) -> Planet
          neue Beschleunigung, Geschwindigkeit, Position
calcF  :: Planet -> Planet -> Force
          Gravitationskraft zwischen zwei Objekten
sumFs  :: Force -> Force -> Force
          Vektoraddition von Kräften
```