Chapter 3: Programming with Combinators

Learning targets of this chapter:

1. Thinking in functional components, avoid thinking in
   - control flow (recursion)
   - details of the data structure (pattern matching in inductive definitions)
2. Natural use of arrows in type signatures
3. Knowledge of the central combinators
   - graphical notion of their structure
   - application of combinators
4. Trees: traversal and search
5. `Functor` as type class for abstraction of data structures
6. Skeletons, example: divide and conquer

Programming with Combinators

- Combinator: function without free variables
- Context-free semantics
- Clear functionality
- High reusability
- Suitable for special, efficient implementation
- Skeleton: combinator with substantial functionality

Combinators in Place of Explicit Recursion

Example problem: revert a list

- Bad solution: explicit recursion

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

Combinators in Place of Explicit Recursion

Example problem: revert a list

- Same bad, explicitly recursive solution, but at least better readable due to constructor style with pattern matching

```
revert []     = []
revert (x:xs) = appendX2 (revert xs)
         where  appendX2 [] = [x]
                appendX2 (y:ys) = y: (appendX2 ys)
```

- Also explicitly recursive (and also with square execution time)

```
revert []     = []
revert (x:xs) = revert xs ++ [x]
```

- Good solution: combinators

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

Issue of Programming Style

Some useful properties ...

1. avoidance of implicit definitions (recursion)
2. avoidance of case analyses (if-then-else)
3. minimal reference to data structures
4. if data structures, then pattern matching rather than selectors ("constructor style")
5. equations between functions, without reference to arguments ("point-free style")

...and why they are desirable:

- (1),(2),(4),(5): simplification of proofs and program transformations
- (1),(2),(3),(4): concise source code $\Rightarrow$ improved readability
- (3): exchangeability of parts of the implementation

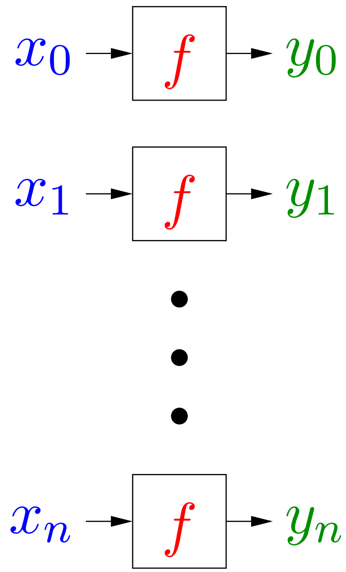
The Benefit of Combinators

- Provision of frequently arising program schemata (`map`, `foldl`, ...)
- Encapsulation of recursion and case analyses
- Adaptation of representations (`flip`, `concat`, ...)
- Skeletons: factorisation of problem solutions into
  - algorithmic part (depth-first search, divide and conquer, ...)
  - application-specific operations
- Simple rules for program transformation,
  e.g., `map g . map f = map (g . f)`

Efficient Implementation of Combinators

The compiler can leverage knowledge about the combinator.

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



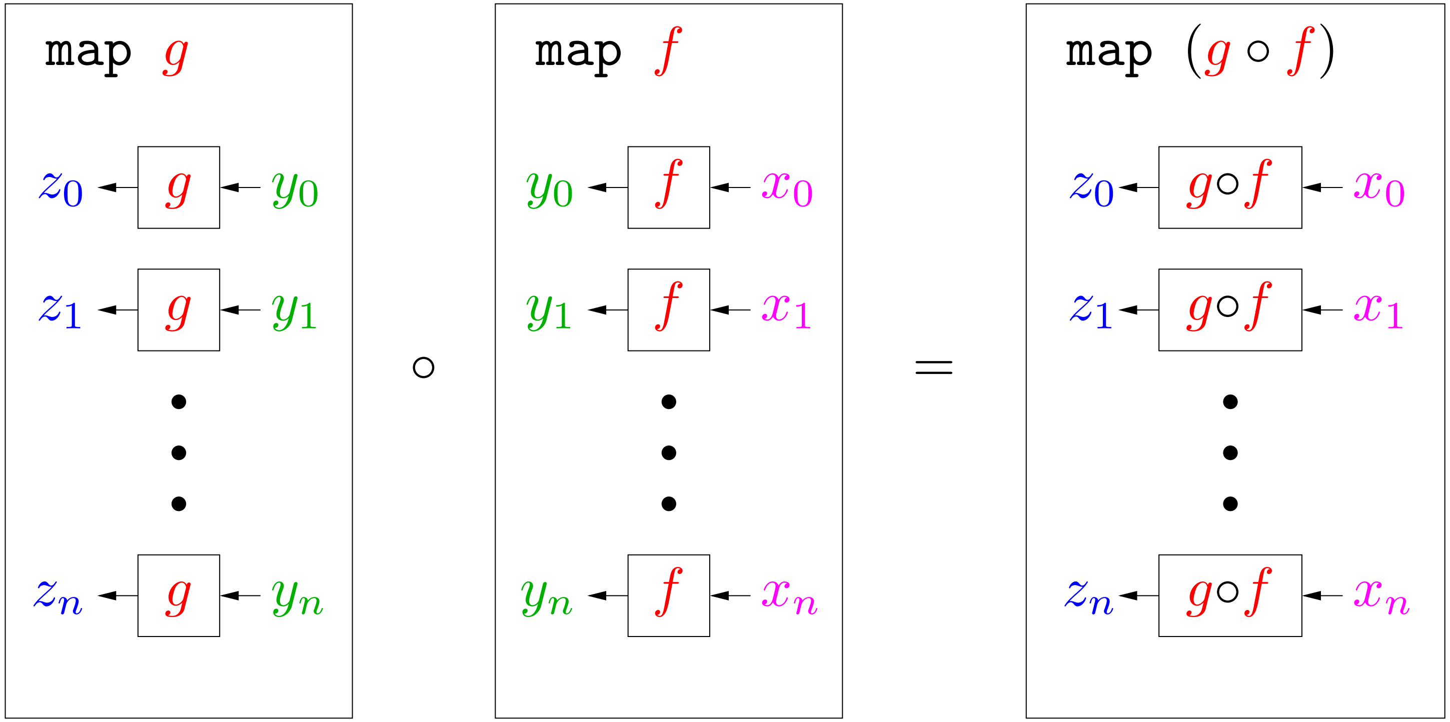
Thinking in functional blocks, not in recursion

Result: all applications of f are independent and could, e.g., be executed simultaneously by a parallel computer.

Program Transformations

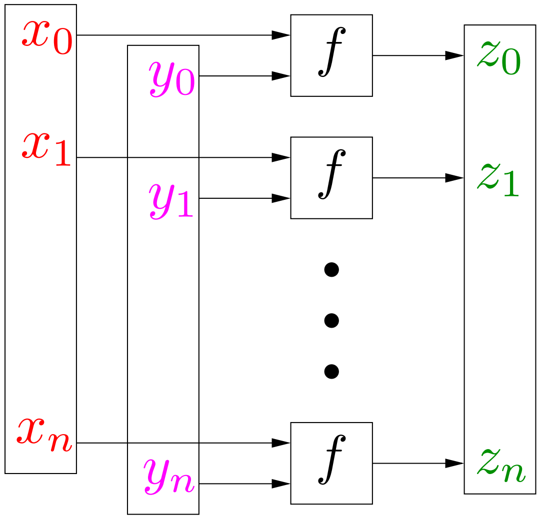
Ex.: $\boxed{\texttt{map } g \texttt{ . map } f \texttt{ == map (} g \texttt{ . } f\texttt{)}}$

- In sequence: avoidance of iterations and temporary data structures (y)
- In parallel: avoidance of synchronization and communication



Vector Operations: zipWith

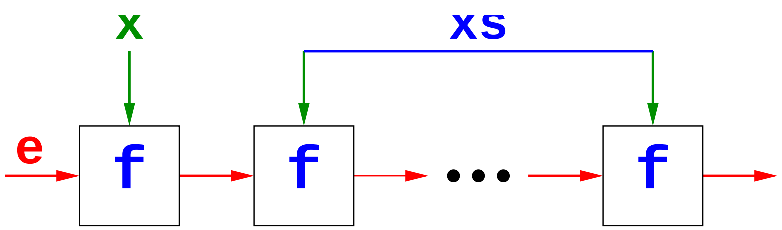
```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _ = []
```




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

The Combinator `foldl`

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



Examples:

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

Reuseability of Combinators

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)
```

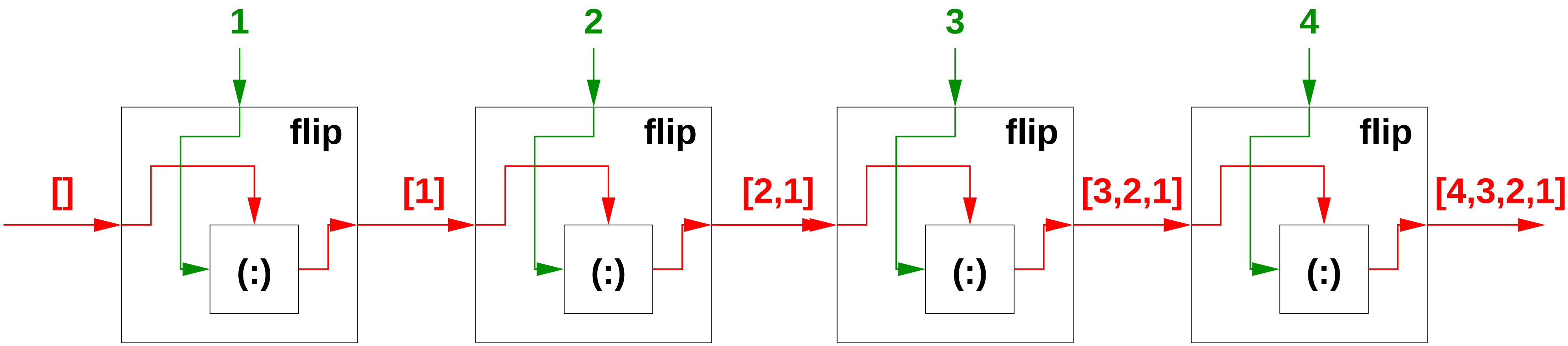
Elegant Definition

```
sum = foldl (+) 0

product = foldl (*) 1

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

List Reversal with `foldl`

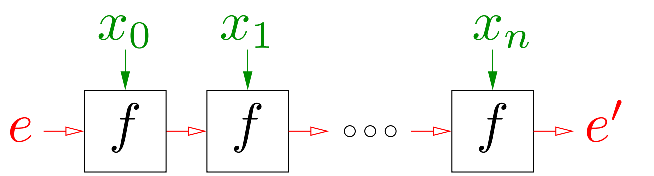
```
reverse = foldl (flip (:)) []
```

```
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]
```

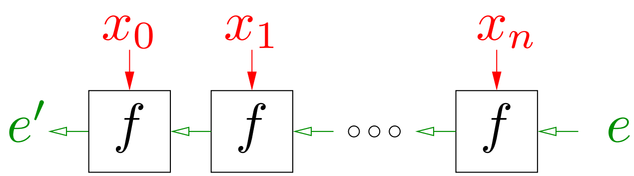


Variants of Combinator `fold`

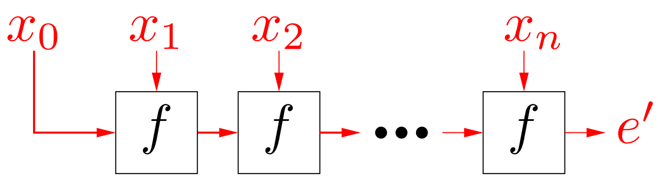
```
foldl :: (a->b->a)->a->[b]->a
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```



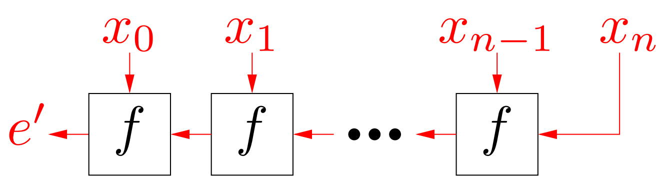

```
foldr :: (a->b->b)->b->[a]->b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```



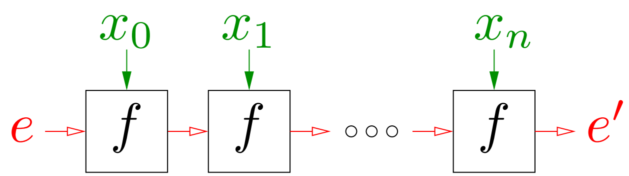

```
foldl1 f (x:xs) = foldl f x xs
```




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



Use of `foldl`, Ex.: `sum`

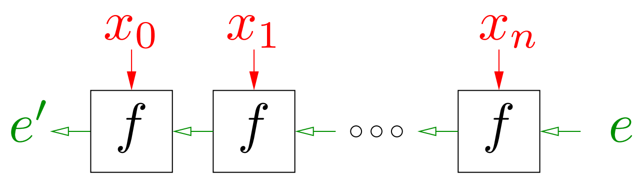



```
foldl :: (a->b->a) -> a -> [b] -> a
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

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

- Since `+` is associative, `foldr` applies as well.
- For strict operators, like `+`, the preferable choice is `foldl`, since
  1. there is no need to store the numbers of the list and
  2. the entire list must be traversed anyway.

Use of `foldr`, Ex.: `and`

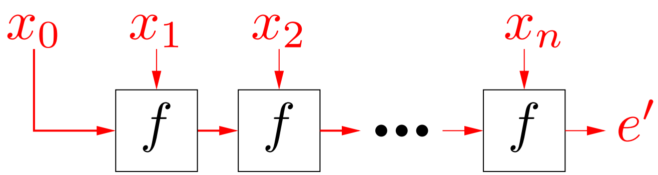



```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f e [] = []
foldr f e (x:xs) = f x (foldr f e xs)
```

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

- Since `&&` is associative, `foldl` applies as well.
- For operators that are non-strict in the second argument, like `&&`, the preferable choice is `foldr`, since
  1. `foldl` would traverse the entire list, but
  2. the result may be reached earlier on.

Use of `foldl1`, Ex.: `maximum`



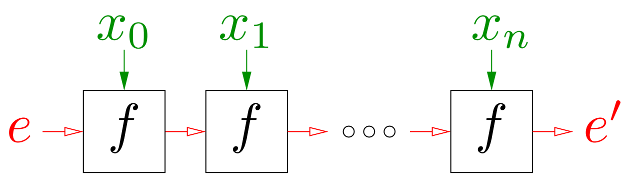

```
foldl1 :: (a->a->a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

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

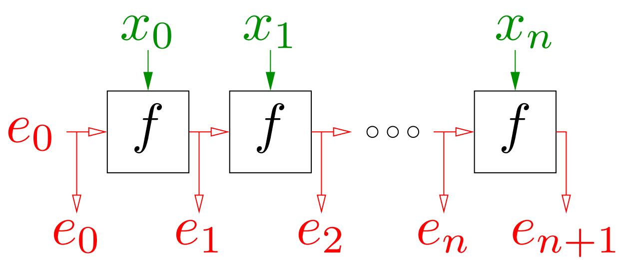
- `foldl1`, since there is no neutral element.
- Only one variable in the signature.
- Since ‘`max`‘ is associative, `foldr1` applies as well.
- Analogously to `sum`: `foldl1`, since `max` is strict.

With Intermediate Results: `scanl` and `mapAccumL`

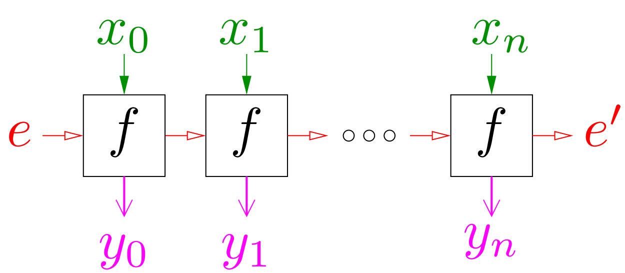
```
foldl :: (a->b->a)
         -> a -> [b] -> a
```




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




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



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

Computation of Target Positions for a Subsequence

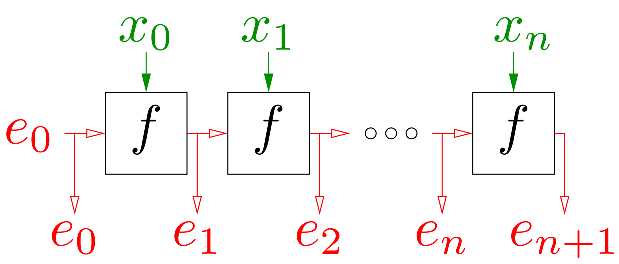
Use: parallel partitioning (in parallel Quicksort)

- Given: condition `(>4)` and list `[2,7,6,4,8,2]`
- Goal: positions in the subsequence of numbers that satisfy the condition

```
input:   [2,7,6,4,8,2]
mask:     [0,1,1,0,1,0]      1 <=> condition satisfied
offsets:   [0,0,1,2,2,3,3]  prefix sum of mask (scanl (+) 0)
result:  [0,0,1,2,2,3]
```

- `Result`: elementwise composition (`zipWith`) of `mask` (color) and `offsets` (value), shorter list determines length
- `Nothing` and `Just` in place of the colors:
  `[Nothing,Just 0,Just 1,Nothing,Just 2,Nothing]`

Use of `scanl`, Ex.: `positions`

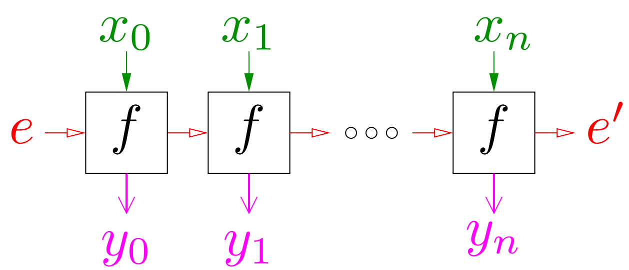



```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
```

```
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
```

Use of `mapAccumL`, Ex.: `positions`

Benefit: a single list traversal, no intermediate data structures




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

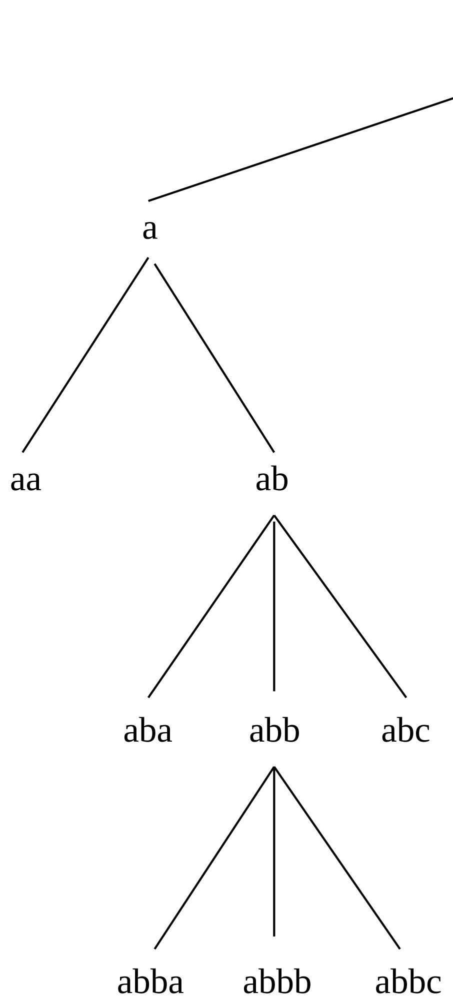
```
positions :: (a->Bool) -> [a] -> [Maybe Int]
positions pred xs
 = let f acc x = if pred x then (acc+1,Just acc)
                           else (acc  ,Nothing)
   in snd (mapAccumL f 0 xs)
```

Combinators for Trees

- Tree type: `data Tree element = Node element [Tree element]`
- Example

```
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" []]]]
```

`tree1`, graphically



Function map for Type Tree

- Definition

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

- Example

```
*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 []]]]
```

Reduction (fold) on Trees

- Definition

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

- Ex.: sum of all numbers in a tree

```
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
```

Depth-First Traversal of Trees

```
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
```

Breadth-First Traversal of Trees

Utility: algorithmic skeleton `workQueue`

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

Implementation of the breadth-first traversal `breadthOrder`:

- append node value at hand to result list `acc`
- append subtree to queue `xs`

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

Search in Trees

1. Generate a list with the elements in the desired order.
2. Look for the first element in the list that satisfies predicate pred.

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

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

> Due to the laziness of Haskell, the data structure will only be traversed to the point at which the desired element is found.

```
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"
```

Generalization of `map`: `fmap`

- `map` and `mapTree` only differ in the data structure (list/tree).
- Type constructors for lists (`[]`), trees (`Tree`) and other data structures are called functors (→ category theory).
- Haskell provides a type class for functors:

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

- Instances of `Functor` should satisfy the following conditions:

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

Instances of `Functor`

- Lists

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

- Trees

```
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

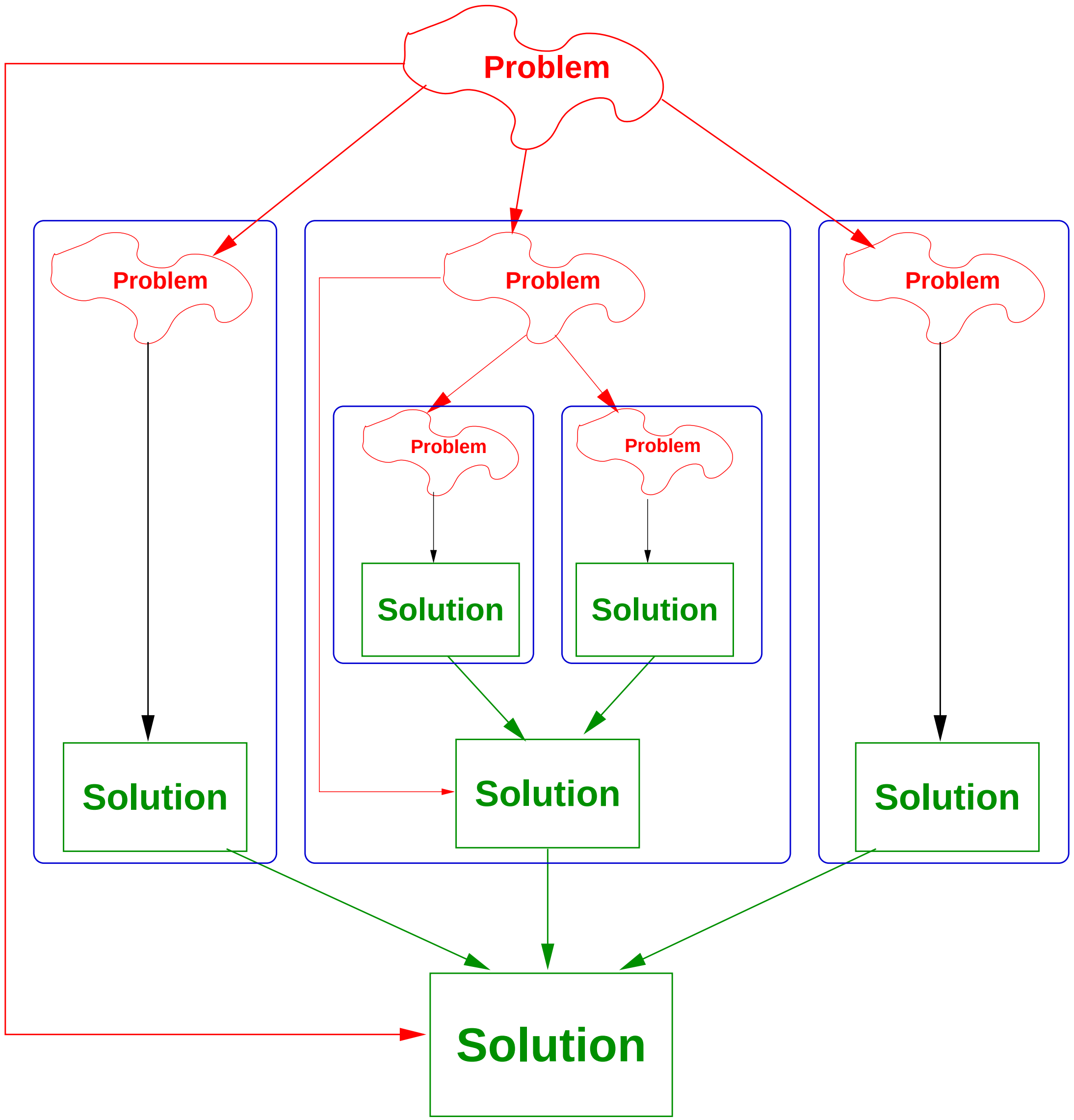
```
instance (Ix a) => Functor (Array a) where        (a: index set)
   fmap fn (MkArray b f) = MkArray b (fn . f)
Prelude Array> fmap (^2) (array ((0,0),(1,1))
                 [((0,0),3),((0,1),6), ((1,0),5), ((1,1),2)])
array ((0,0),(1,1)) [((0,0),9),((0,1),36),((1,0),25),((1,1),4)]
```

A Skeleton for Divide and Conquer (1)

Divide-and-Conquer paradigm:

- If a problem can be solved directly, do it.
- Otherwise
  1. Divide the problem into independent parts of the same type.
  2. Apply the procedure recursively to each subproblem.
  3. Combine the solutions of the subproblems to an encompassing solution.

A Skeleton for Divide and Conquer (2)



A Skeleton for 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))
```

Instantiation with problem-specific functions:

1. `p::(a->Bool)` answers whether the problem can be solved directly.
2. `b::(a->b)` solves the problem directly.
3. `d::(a->[a])` divides the problem into a list of subproblems.
4. `c::(a->[b]->b)` combines the solutions of the subproblems.

`dcprg::(a->b)` is the resulting program.

Use of dc (1): functional "quicksort"

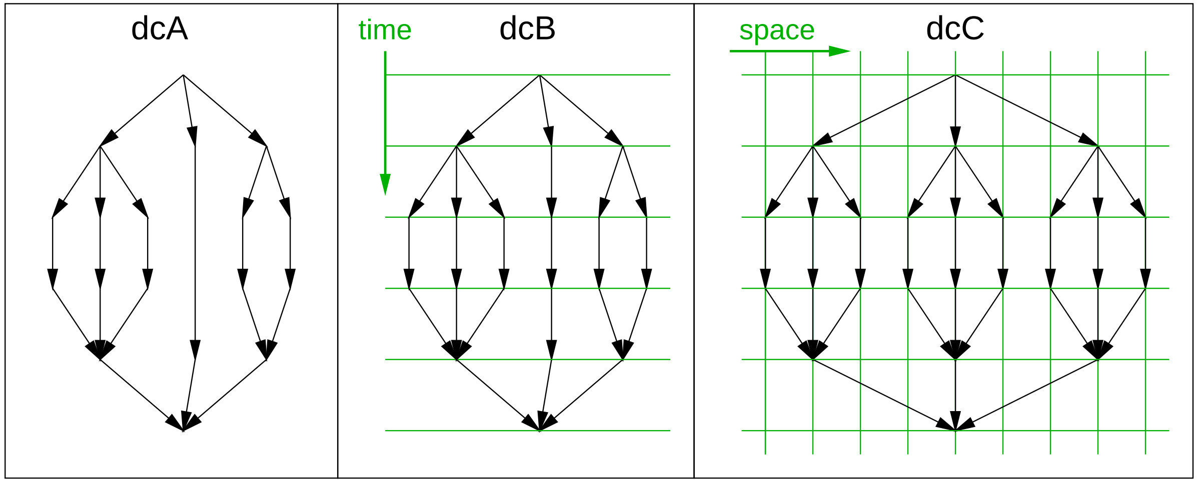
```
quicksort :: Ord a => [a] -> [a]
quicksort = dc p b d c
 where p xs     = length xs < 2
       b xs     = xs
       d (x:xs) = let (a,b) = partition (<x) xs
                  in [a,b]
       c (x:_) [a,b] = a ++ (x : b)
```

Use of dc (2): Queens Problem

- Input: number of rows/columns of the "chess" board
- Output: list of all solutions;
  for every solution, list element i is the column position of the queen in row i

```
queens :: Int -> [[Int]]
queens n = dc p b d c ([],[0..n-1]) where
   p (_,remain) = null remain //remain = column positions still free
   b (placed,_) = [placed]    //placed = rows with queens placed so far
   d (placed,remain)
     = [ (placed++[i],filter (/=i) remain)       //no column conflict
       | i <- remain, not (diagonal_attack i) ] //no diagonal conflict
       where diagonal_attack i
               = or [ length placed - j  ==  abs (i - placed!!j)
                    | j<-[0..length placed -1] ]
   c _ = concat
```

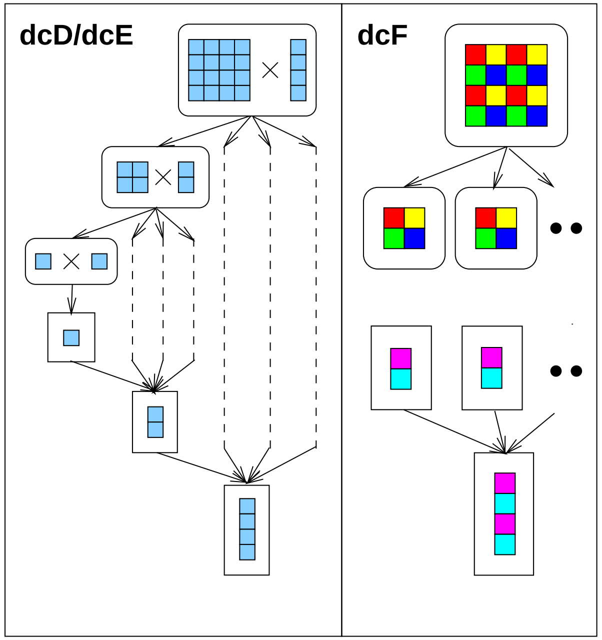
Divide-and-Conquer Skeletons: Task Division



skeleton	restriction	instances
`dcA`	independent subproblems	Quicksort, Maximum Independent Set
`dcB`	fixed recursion depth	n Queens
`dcC`	fixed division degree k	Karatsuba Integer Product (k=3)

The Skeleton-Based Parallelization of Divide-and-Conquer Recursions
(Dissertation of Christoph Herrmann, Universität Passau, June 2000)

Divide-and-Conquer Skeletons: Data Division



`dcD`	block recursion	Triangular Matrix Inversion (k=2), Batcher Sort (k=2)
`dcE`	elementwise operations	Matrix-Vector Product (k=4)
`dcF`	communication between corresponding elements	Karatsuba's Polynomial Product (k=3), Bitonic Merge (k=2), FFT (k=2), Strassen's Matrix Product (k=7)