Funktionale Programmierung mit Haskell

Rekursion

```
-- normal recursion
fak n = if (n==0) then 1 else n * fak (n-1)

-- linear recursive, but not end recursive
fak' n = if (n==0) then 1 else (n * fak (n-1))

--using accumulator. end recursive.
fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n*acc)
fak'' n = fakAcc n 1

-- fibbonacci
fib n
| (n == 0) = 0
| (n == 1) = 1
| otherwise = fib (n - 1) + fib (n - 2)

-- fibbonacci end recursive
fibAcc n n1 n2
| (n == 0) = n1
| (n == 1) = n2
| otherwise = fibAcc (n - 1) n2 (n1 + n2)
fib n = fibAcc n 0 1
```

Listen

```
-- length of a list
length l = if (null l) then 0 else 1 + (length (tail l))

-- is the element y in the list?
isIn [] y = False
isIn (x:xs) y = if (x == y) then True else isIn xs y

-- get the maximum element in the list
maximum [] = error "empty"

maximum (x:[]) = x
```

```
maximum (x:xs) = max x (maximum xs)

-- Append (Infix: a++b), O(length left)
app [] r = r
app (x:xs) r = x:(app xs r)

-- reverse list
rev [] = []
rev (x:xs) = app (rev xs) [x]

-- other functions
head l -- first element of list
tail l -- list without first element
take n l -- first n elements of l
drop n l -- l without first n elemenets

-- map apply a function to all list members
map :: (s -> t) -> [s] -> [t]
map f [] = []
map f (x:xs) = (f x) : map f xs

-- filter a list
filter :: (t -> Bool) -> [t] -> [t]
filter pred [] = []
filter pred (x:xs) = if pred x then x:(filter pred xs) else filter pred xs
```

```
-- intervals
[a..b] = [a, a+1, ..., b-1, b]

-- list comprehensions
[e|q1, ..., qn]
[f x | x<-l] <=> map (\x->f x) l <=> map f l
[x | x<-l, pred x] <=> filter (\x -> pred x) l <=> filter pred l
[f x | x<-l, pred x] <=> map f (filter pred l)
squares n = [ x*x | x <- [0..n]]
primepowers n = [pow2 p i | p <- primes, i <- [1..n]] -- qualifiers are being
applied to from right to left.

-- prime sieve
primes :: Integer -> [Integer]
primes n = sieve [2..n]
    where sieve [] = []
    sieve (p:xs) = p : sieve (filter (not . multipleOf p) xs)
    multipleOf p x = x `mod` p == 0
```

Kombinatoren

foldr, foldl

```
foldr :: (s -> t -> t) -> t -> [s] -> t
foldr op i [] = i
foldr op i (x:xs) = op x (foldr op i xs)

foldl :: (t -> s -> t) -> t -> [s] -> t
foldl op i [] = i
foldl op i (x:xs) = foldl op (op i x) xs

-- examples
sum :: [Int] -> Int
sum = foldr (+) 0

product :: [Int] -> Int
product = foldr (*) 1

listlength :: [t] -> Int
listlength = foldr (\x n -> n + 1) 0

sentenceLength :: [String] -> Int
sentenceLength = foldr (\l n -> length l + n) 0

app :: [t] -> [t] -> [t]
app left right = foldr (:) right left

rev :: [t] -> [t]
rev = foldl cons []
    where cons xs x = x:xs

flatten :: [[t]] -> [t]
flatten = foldr app []

map :: (s -> t) -> [s] -> [t]
map f = foldr (\x l -> f x : l) []
```

Zip

```
-- combine two lists element-wise with a combinator function
zipWith :: (s -> t -> u) -> [s] -> [t] -> [u]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs ys = []
zip = zipWith (,) -- combine two lists into list of two-tuples

-- Example: Hamming distance
hamming l r = sum (zipWith differs l r)
    where differs x y = if (x == y) then 0 else 1
```

Streams

```
-- iterate loop: (f,a) => [a, f(a), f(f(a)), f(f(f(a))), ...]
iterate :: (a -> a) -> a -> [a]
iterate f a = a : iterate f (f a)
iterate f x !! 23 -- iterate only 23 times

takeWhile :: (a -> Bool) -> [a] -> [a]
-- takeWhile op list supplys elements from the list as long as op is true for
the current list element.
```

Approximation von Pi

```
partialSums :: [Double] -> [Double]
partialSums (x : xs) = pSumsAcc xs x
    where pSumsAcc (x : xs) acc = acc : (pSumsAcc xs (x + acc))
approxPi = map (*4) (partialSums piSeq)
    where piSeq = zipWith (/) (iterate negate 1) odds
```

Sieb des Eratosthenes

```
oddPrimes (p : ps) = p : (oddPrimes [p' | p' <- ps, p' `mod` p /= 0])
primes = 2 : oddPrimes (tail odds)
odds = 1 : map (+2) odds
-- in comprehension: sort out multiples of current value
```

Typen

```
Int -- whole numbers, restricted size
Integer -- whole numbers, arbitrary size
Float -- floating point numbers, single precision
Double -- floating point numbers, double precision
Bool -- boolean value
Char -- unicode character
```

Beispiel für Typen: Mengen

```
type Set t = [t]

insert x s = if (isIn s x) then s else x:s

delete x [] = []
delete x (y:ys) = if (x==y) then ys else y:(delete x ys)

fold = foldr
```

Backtracking, Damenproblem

Vorgehensweise

- Mit zulässiger Anfangskonfigurationbeginnen (leeres Brett)
- Baum von zulässigen Folgekonfigurationen konstruieren und durchsuchen (weitere Damen platzieren, ohne vorhandene zu bedrohen)
- Bis Lösungskonfiguration gefunden (8 Damen auf Brett)

```
type Conf = [Integer] -- Konfiguration = Liste von Zeilenpositionen
successors :: Conf -> [Conf] -- Folgekonfigurationen bestimmen
legal :: Conf -> Bool -- Konfiguration auf Zulässigkeit prüfen
solution :: Conf -> Bool -- Zulässige Konfiguration prüfen, ob korrekte Lösung

solutions = backtrack initial -- liefert alle Lösungen (oder mit head nur
erste)

backtrack :: Conf -> [Conf]
backtrack conf =
    if (solution conf) then [conf]
    else flatten (map backtrack (filter legal (successors conf)))

successors :: Conf -> [Conf] -- Folgekonfigurationen bestimmen
successors board = map (:board) [1..8] -- in jeder Zeile eine Dame platzieren


threatens :: Int -> Int -> Conf -> Bool
```

```
threatens diag row [] = False -- diag = horizontaler abstand zw 1ter und 2ter
Dame
threatens diag row (row2:rest) =
    (row2==row+diag) || (row2==row-diag) ||
    (row2==row) || threatens (diag+1) row rest
legal :: Conf -> Bool -- Konfiguration auf Zulässigkeit prüfen
legal [] = True
legal (row:rest) = (not (threatens 1 row rest))

solution :: Conf -> Bool -- Ist Konfiguration fertige Lösung? Ja wenn 8 Damen.
solution board = (length board) == 8

initial = [] -- Initialkonfiguration: Leeres Brett.
```

Algebraische Datentypen

```
data DataName = ConstructorName Param1 Param2 | ConstructorName2 Param1b
Param2b
data Season = Spring | Summer | Autumn | Winter -- Enumeration
data People =   Person String Int
             | Adult String

jogi = Person "Joachim Löw" 50 -- jogi :: People
isAdult (Person name age) = age >= 18 -- isAdult :: People -> Bool
isAdult (Adult name) = True
```

Polymorphe Datentypen

```
data Maybe t = Nothing | Just t
Just True :: Maybe Bool
Nothing :: Maybe String -- Optional parameters default to String (?)
```

Dicht und dünn besetzte Matrizen:

```
data Matrix t =   Dense [[t]] -- list of rows
               | Sparse [(Integer,Integer,t)] t -- matrix values (i, j, val)
and default value

unit :: Integer -> Matrix Float
unit n = Sparse [(i,i,1.0) | i <- [1..n]] 0

rotation :: Double -> Matrix Double
rotation alpha = Dense [[cos alpha, -sin alpha], [sin alpha, cos alpha]]
```

Binäre Bäume

```
data Tree t = Leaf | Node (Tree t) t (Tree t) -- data is only in nodes, not in
leafs.
someTree = Node (Node Leaf 1 Leaf) 3 Leaf

-- Get amount of elements stored in tree
size :: Tree t -> Int
size Leaf = 0
size (Node left x right) = (size left) + 1 + (size right)

-- Height of the tree
height :: Tree t -> Int
height Leaf = 0
height (Node left x right) = 1 + (max (height left) (height right))

-- With fold. f has three params.
foldT :: (s -> t -> s -> s) -> s -> Tree t -> s
foldT f i Leaf = i
foldT f i (Node left x right) = f (foldT f i left) x (foldT f i right)

size = foldT (\left x right -> left+1+right) 0
height = foldT (\left x right -> 1+(max left right)) 0
paths tree = foldT consAll [[]] tree        -- paths :: Tree t -> [[t]]
    where consAll left x right = map (x:) (left++right) --todo?
```

Rot-Schwarz-Bäume

Knoten sind rot oder schwarz, Blätter sind schwarz.Invarianten:

- Kein roter Knoten hat roten Elternknoten.
- Alle vollständigen Pfade haben dieselbe Anzahl schwarzer Knoten.
- Baum ist sortiert: Elemente linker Teilbaum <= KnotenElement. Elemente rechter Teilbaum >= KnotenEl.

```
data Color = Red | Black
data RedBlackTree t = Leaf | Node Color (RedBlackTree t) t (RedBlackTree t)

fold :: (Color -> s -> t -> s -> s) -> s -> RedBlackTree t -> s
fold f i Leaf = i
fold f i (Node c left x right) = f c (fold f i left) x (fold f i right)

mapRB :: (Color -> s -> t) -> RedBlackTree s -> RedBlackTree t
mapRB f Leaf = Leaf
mapRB f (Node c left x right) = Node c (mapRB f left) (f c x) (mapRB f right)
```

Typklassen

```
Eq t: == :: t->t->Bool, /= :: t->t->Bool
    Instanzen: Eq Integer, Eq Char, Eq Bool, ...

Ord t: <= :: t->t->Bool, <, >, >=
    Instanzen: Ord Float, Ord Integer, Ord Char, ...

Num t: + :: t->t->t, *, -, negate :: t->t, abs :: t->t, signum :: t->t,
fromInteger :: Integer->t
    Instanzen: Num Float, Num Integer, ...

Show t: show :: t->String
    Instanzen: Show Float, Show Bool

Enum t: succ :: t->t, pred :: t->t, toEnum :: Int->t, fromEnum :: t->Int,
enumFromTo :: t->t->[t]
    Instanzen: Enum Bool, Enum Int, Enum Char
```

Beispiel:

```
class (Eq t) where
    (==) :: t -> t -> Bool
    (/=) :: t -> t -> Bool
    x /= y = not (x == y)   -- default implementation
    x == y = not (x /= y)   -- default implementation

instance (Eq Bool) where
    True == True = True
    False == False = True
    False == True = False
    True == False = False
```

Weiteres Beispiel

```
class Unit u where
    plus :: u -> u -> u
    minus :: u -> u -> u
    ntimes :: Double -> u -> u
instance Unit Metre where
    (M x) `plus` (M y) = M (x+y)
    (M x) `minus` (M y) = M (x-y)
    x `ntimes` (M y) = M (x*y)
instance Unit Yard where
    (Yd x) `plus` (Yd y) = Yd (x+y)

    -- ...
```

```
-- conversion
class (Unit u) => Length u where
    toMetre :: u -> Metre
    fromMetre :: Metre -> u
instance Length Metre where
    toMetre = id
    fromMetre = id
instance Length Yard where
    toMetre (Yd x) = M (x*0.9144)
    fromMetre (M x) = Yd (x/0.9144)
```