Haskell

Contents

Befehle aus Prelude

Alle weitere Befehle befinden sich auf einem anderen CheatSheet (nicht von mir).

map

```
map func list
map (2*) [1,2,3] = [2, 4, 6]
```

fold

foldr (right)

```
foldr :: (s -> t -> t) -> t -> [s] -> t
foldr op i [] = i
foldr op i (x:xs) = op x (foldr op i xs)
foldr (+) 0 [1,2,3,4] :=> (1+(2+(3+(4+0))))
```

foldl (left)

```
foldl :: (t -> s -> t) -> t -> [s] -> t
foldl op i [] = i
foldl op i (x:xs) = foldl op (op i x) xs
foldl (+) 0 [1,2,3,4] :=> ((((0+1)+2)+3)+4
```

Beispiele

```
Input: foldl (/) 64 [4,2,4]
Output: 2.0

Input: foldl (\x y -> 2*x + y) 4 [1,2,3]
Output: 43
```

Foldl ist ggf effizienter, da endrekursiv (→Lazy Evaluation)

filter

```
filter predikat list
filter (>5) [3,4,5,3,6,7] = [6, 7]
filter odd [1,2,3,4,5] = [1, 3, 5]
filter (=='x') "xxabcdx" = "xxx"

filter isLeaf [Lead, Leaf, Node, Leaf] = [Leaf, Leaf, Leaf]
isLeaf Leaf = True
isLeaf Node = False
```

iterate

""creates an infinite list where the first item is calculated by applying the function on the secod argument, the second item by applying the function on the previous result and so on.""

```
iterate func list
iterate (2*) 1
iterate f a = f ( f a )

Input: take 10 (iterate (2*) 1)
Output: [1,2,4,8,16,32,64,128,256,512]

Input: take 10 (iterate (\x -> (x+3)*2) 1)
Output: [1,8,22,50,106,218,442,890,1786,3578]
```

iterate beginnt mit dem Initial-Wert (Identität)!
(Schaue bei Zufallzahlen)

elem

returns True if the list contains an item equal to the first argument

```
Input: elem 1 [1,2,3,4,5]
Output: True
```

```
Input: 'o' `elem` "Zvon"
Output: True
```

zipWith

makes a list, its elements are calculated from the function and the elements of input lists occuring at the same position in both lists

```
Input: zipWith (+) [1,2,3] [3,2,1]
Output: [4,4,4]

Input: zipWith (\x y -> 2*x + y) [1..4] [5..8]
Output: [7,10,13,16]
```

replicate

replicate n m : list mit dem m, n mal Wiederholt

```
replicate n m = take n (repeat m)
replicate 10 5 → [5,5,5,5,5,5,5,5,5,5]
```

flatten==concat

flatten aus Vorlesung, concat aus Prelude!

```
Input: concat [[1,2,3], [7,2,39]]
Output: [1,2,3,7,2,39]

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

not

Boolean not

```
Input: not (1>2) -> Output: True
```

odd und even

Ungerade Zahlen

```
Input: odd 13  -> Output: True
```

Gerade Zahlen

```
Input: even 12 -> Output: True
```

mod und div

```
3 `mod` 12 = mod 3 12 = 3
6 `div` 2 = div 6 2 = 3
```

takeWhile und dropWhile

```
takeWhile predicate list
```

creates a list from another one, it inspects the original list and takes from it its elements to the moment when the condition fails, **then it stops processing**

```
Input: takeWhile odd [1,3,5,7,9,10,11,13,15,17]
Output: [1,3,5,7,9]
```

```
dropWhile predicate list
```

Drop elements until predicate is true

```
Input: dropWhile even [2,4,6,7,9,11,12,13,14]
Output: [7,9,11,12,13,14]
```

fst und snd

returns the first/second item in a tuple

```
fst (1,2) -> 1
snd („a“, „b“) -> „b“
```

concatMap

creates a list from a list generating function by application of this function on all elements in a list passed as the second argument

```
Input: concatMap (enumFromTo 1) [1,3,5]
Output: [1,1,2,3,1,2,3,4,5]
```

lookup

get snd from tupel with el as fst

```
Eq a => a -> [(a,b)] -> Maybe b
lookup el tuples
Input: lookup 'c' [('a',0),('b',1),('c',2)]
Output: Just 2
```

splitAt

splitAt n xs returns a tuple where first element is xs prefix of length n and second element is the remainder of the list

```
Input: splitAt 5 [1,2,3,4,5,6,7,8,9,10]
Output: ([1,2,3,4,5],[6,7,8,9,10])
```

last

returns the last item of a list

```
Input: last [1,2,3]
Output: 3
```

Befehle außer Prelude

sort...

delete...

Modul Data.Char

ord

Symbol -> Position in Enum

```
Input: ord 'a'
Output: 97
```

chr

Position in Enum -> Symbol

```
Input: chr 97
Output: 'a'
```

Eigene nützliche Befehle

list.get(index)

get list index = element by index or error

```
get :: [a] -> Int -> a
get [] i = error "invalid index"
get (x:xs) 0 = x
get (x:xs) i = get xs (i-1)
```

BESSER: (!!)-Operator

change list at position i with funtion f

changeListAt func index list = modified list

```
changeListAt :: (a -> a) -> Int -> [a] -> [a]
changeListAt f _ [] = []
changeListAt f 0 (x:xs) = (f x : xs)
changeListAt f n (x:xs) = x : changeListAt f (n-1) xs
```

transponieren die Matrix

```
transpose lists = if length (concat lists) == 0
                  then []
                  else (map head lists) : transpose (map tail lists)

transpose [[1,2,3], [4,5,6], [7,8,9]] --> [[1,4,7],[2,5,8],[3,6,9]]
```

duplicates

duplicates xs = True gilt, wenn in der Liste xs mindestens ein Element doppelt vorkommt.
Geben Sie einen möglichst allgemeinen Typ für duplicates an.

```
duplicates :: Eq a => [a] -> Bool
duplicates [] = False
duplicates (x:xs) = x ‘elem‘ xs || duplicates xs
```

rotations

```
rotations :: [a] -> [[a]]
rotations xs = take (length xs) (iterate rot xs)
where
     rot (x:xs) = xs ++ [x]
rotations "abcde" -> ["abcde","bcdea","cdeab", “deabc”, “eabcd”]
```

permutations

```
permutations :: [a] -> [[a]]
permutations []     = [[]]
permutations (x:xs) = concatMap (rotations.(x:)) (permutations xs)

permutations "abc" -> ["abc", "acb", "bac", "bca", "cab", "cba"]
```

delete

removes the first occurrence of the specified element from its list argument

```
delete 2 [1,2,3] = [1,3]
delete x [] = []
delete x (y:ys) = if (x==y) then ys else y:(delete x ys)
```

FAQ

Allgemein

Ein Intervall erstellen

```
[a..b] = [a,a+1,a+2,...,b]
```

Head of emtpy list

```
head [] = Prelude error
```

cons (:) vs append (++)

```
x:y:[] = [x,y]
[x] ++ [y] = [x,y]
```

“The : operator is known as the "cons" operator and is used to prepend a head element to a list. So [] is a list and x:[] is prepending x to the empty list making a the list [x]. If you then cons y:[x] you end up with the list [y, x] which is the same as y:x:[]”

The ++ operator is the list concatenation operator which takes two lists as operands and "combine" them into a single list. So if you have the list [x] and the list [y] then you can concatenate them like this: [x]++[y] to get [x, y].

Notice that : takes an element and a list while ++ takes two lists.

Wie konvertiert man Zahl nach String?

```
-> mit show
a = 15
show a = „15“ = [„1“, „5“]
```

Wann wird ein Akkumulator benötight?

Um die lineare, aber nicht endrekursive funktionen endrekursiv zu machen

Was ist Prelude?

The Prelude: a **standard module of Haskell**. The Prelude is imported by default into all Haskell modules unless either there is an explicit import statement for it, or the NoImplicitPrelude extension is enabled.

Wie schreibt man in Haskell != (nicht gleich) ?

--> x \= y

```
x /= y === not (x == y)
```

Power: ** oder ^ ?

Beide Varianten sind richtig, verschiedene interne definitionen.
Verwende ^ für Integers, ** für Floats.

Was macht Dot-Operator (.)?

```
The . operator composes functions. For example, a . b
Code sumEuler = sum . (map euler) . mkList
is exactly the same as:
sumEuler myArgument = sum (map euler (mkList myArgument))
```

Wie kann man ein Predikat p negieren?

Mit not und Dot-Operator:

```
filter (not . p) list
```

map vs iterationen mit (x:xs)

```
caesar n s = map (\c -> chr (z + (ord c - z + n) `mod` 26)) s
       where z = ord 'A'

caesar n [] = []
caesar n (x:xs) = (caesar_one n x) : (caesar n xs)
caesar_one n = \c -> chr (z + (ord c - z + n) `mod` 26)
       where z = ord 'A'
```

Was macht @ Operator?

Alias für Variable: list@(x:xs)

Yes, it's just syntactic sugar, with `@` read aloud as "as". `ps@(p:pt)` gives you names for

1. the list: `ps`
2. the list's head : `p`
3. the list's tail: `pt`

Without the `@`, you'd have to choose between (1) or (2):(3).

This syntax actually works for any constructor; if you have `data Tree a = Tree a [Tree a]`, then `t@(Tree _ kids)` gives you access to both the tree and its children.

Was macht !! Opeartor?

list !! n -> get n-te element from list

The index must be smaller than the length of the list, otherwise the result is undefined.

```
[1,2,3] !! 1 = 2
```

Ist die Reihenfolge bei where wichtig?

Nein:

```
x = a*a
  where a = b
        b = 12
```

Kein Fehler, x ist immer 144

data vs type

data allows you to introduce a new algebraic data type, while **type** just makes a type synonym.

```
data Tree a = Node Tree a Tree | Leaf a
type String = [Char]
```

Was ist allgemeinste Typ von X?

```
duplicates :: Eq a => [a] -> Bool
duplicates [] = False
duplicates (x:xs) = x `elem` xs || duplicates xs
```

Tuple in Pattern Matching

Intuitiv:

```
shuffle ((i, j):rest) list = shuffle rest (swap i j list)
```

Unendliche Listen

Wie erstellt man am besten eine unendliche Liste mit einer Zahl/Symbol?

```
-> mit repeat
repeat 3 = [3, 3, 3, 3, 3 ...]
```

Wie erstellt man eine unedliche Liste als Folge?

```
-> List Comprehensions
[2,4..] = [2, 4, 6, 8, 10...]
[5, 10..] = [5, 10, 15, 20...]

-> mit iterate
iterate (2*) 1 = [1, 2, 4, 8, 16, 32, 64..]
```

Theorie

Kontrollflus

If‘s

```
absolute x = if (x<0) then (-x) else x
```

Guard

```
absolute x
 | x < 0 = -x
 | otherwise = x
```

Pattern Matching

Mit _ ist die Variable gemeint, die nicht verwendet wird

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs

length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

Lokale Bindung

let

```
energy m = let c = 299792458 in m * c * c
```

where

```
energy m = m * c * c
where c = 299792458

In where kann man funktionen, if-then-else und Pattern Matching stecken.
f x
  | cond1 x   = a
  | cond2 x   = g a
  | otherwise = f (h x a)
  where
    a = w x
    b = z y
    g x = repeat x
```

Polymorphe Datentypen

Maybe: Optionale Werte

```
data Maybe t = Nothing | Just t
Just True :: Maybe Bool
```

Either: Summen-Typ

```
data Either s t = Left s | Right t
Left 42 :: Either Int String
Right "true" :: Either Int String
```

Algebraische Datentypen

```
data Season = Spring | Summer | Autumn | Winter
```

Shape

```
data Shape = Circle Double| Rectangle Double Double

area :: Shape -> Double
area (Circle r) = pi*r*r
area (Rectangle a b) = a*b
```

Rekursive Datentypen

Stack

```
data Stack t = Empty | Stacked t (Stack t)

pop Empty = error "Empty" push x s = Stacked x s
pop (Stacked x s) = s

top Empty = error "Empty"
top (Stacked x s) = x

someStack :: Stack Integer
someStack = Stacked 3 (Stacked 1 Empty)
```

Binary Tree

```
data Tree t = Leaf | Node (Tree t) t (Tree t)
```

Akkumulator

Explizites Zwischenspeichern von partiellen Ergebnissen

Ohne Akkumulator

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

O(n) Aufrufe, O(n) Speicher

Warum so viel Speicher? -> Aufrufstack wächst

```
fak 3 -> 3 * (fak 2)
-> 3 * (2 * (fak 1))
-> 3 * (2 * (1 * (fak 0)))
-> 3 * (2 * (1 * 1))
-> 3 * (2 * 1)
-> 3
```

Mit Akkumulator

```
fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n*acc)
fak n = fakAcc n 1
```

O(n) Aufrufe, O(1) Speicher

```
fak 3 -> fakAcc 3 1 -> fakAcc 2 3 -> fakAcc 1 6 -> fakAcc 0 6 -> 6
```

Endrekursion

Eine Funktion heißt linear rekursiv, wenn in jedem Definitionszweig nur ein rekursiver Aufruf vorkommt.

-> Nur 1 Aufruf pro Zweig

Eine linear rekursive Funktion heißt endrekursiv (tail recursive), wenn in jedem Zweig der rekursive Aufruf nicht in andere Aufrufe eingebettet ist.

-> Nur 1 Aufruf pro Zweig, und es gibt keine „äußere Aktionen“ mit dem Aufrug

Linear, aber nicht endrekursiv

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

Endrekursiv

```
fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n*acc)
fak n = fakAcc n 1
```

Typklassen

Neue Typlassen erstellen

Man kann eigene Typklassen erstellen mit
`class X where`

Typklassendefinition

Typklassen-Definition: `Eq t` mit Default-Implementierungen

```
class Eq t where
  (==) :: t -> t -> Bool
  (/=) :: t -> t -> Bool

  x /= y  = not (x == y)
  x == y  = not (x /= y)
```

Typklassen-Instanziierung: Gleichheit von `Bool`

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

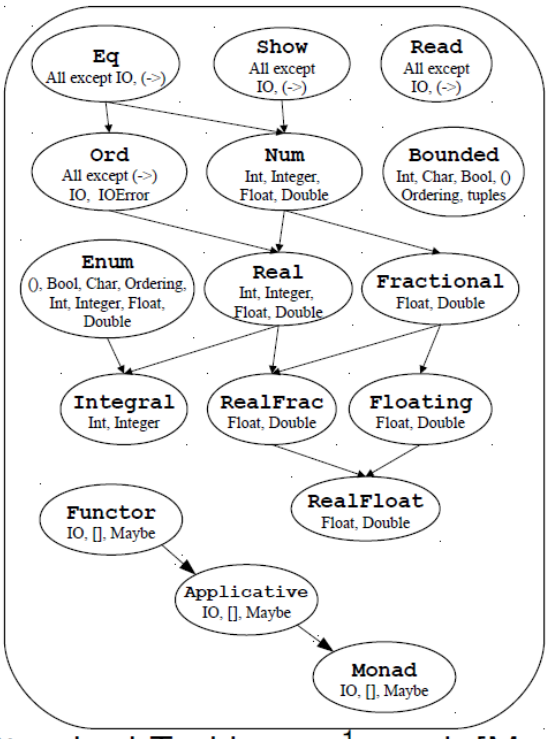
oder

```
instance Eq Bool where
  True  /= True  = False
  False /= False = False
  False /= True  = True
  True  /= False = True
```

- Fehlende Implementierungen: Default-Implementierung

⇒ { `(==)` } und { `(/=)` } sind je minimal-vollständig

Hierarchie



Standard-Typklassen[1], nach [Mar]

Automatische Instanziierung

Am Ende der Deklaration einfach deriving Eq/Ord... schreiben

Automatische Instanziierung

Gleichheit für Datentypen: `Eq Shape`

```
data Shape =   Circle Double -- Radius
             | Rectangle Double Double -- Seitenlängen
            deriving Eq
```

Automatische Instanziierung: **`deriving Eq`**

- verschiedene Konstruktoren ⇒ verschiedene Werte
- gleicher Konstruktor, verschiedene Parameter ⇒ verschiedene Werte
- Sowieso: gleicher Konstruktor, gleiche Parameter ⇒ gleicher Wert

```
Circle 1 == Square 1   ⇒   False
Circle 1 == Circle 3   ⇒   False
Square 2 == Square 2   ⇒   True
```

Automatische Instanziierung: Auch für `Show`, `Ord`, `Enum`

Enum

Operationen:

```
succ :: t -> t        succ 5 = 6
pred :: t -> t        pred “b” = “a”

toEnum :: Int -> t fromEnum :: t -> Int
returns the item at argument position from an enumeration
toEnum 35::Char -> „#“

enumFromTo :: t -> t -> [t]
enumFromTo 'c' 'g' = "cdefg"
enumFromTo 12 17 = [12,13,14,15,16,17]
```

[a..b] <=> enumFromTo a b

Funktionen

Infix-Notation und Präfix-Notation

```
Infix: left ‘app‘ right
Präfix: app left right

Infix 0.9999 == 1
Präfix (==) 0.9999 1
```

Currying und Unterversorgung

Currying: Ersetzung einer mehrstelligen Funktion durch Schachtelung einstelliger Funktionen
Funktionsanwendung ist links-assoziativ:

```
f 3 7 ===  (f 3) 7
```

Konsequenz: Funktionstypen -> sind rechts-assoziativ:

```
Int -> Int -> Int === Int -> (Int -> Int)
```

Unterversorgung

Unterversorgung:

- Anwendung "mehrstelliger" Funktionen auf zu wenige Parameter
- Zusammen mit Kombinatoren: kompakte Schreibweise

Beispiel: $(\lambda a.\lambda x.a \cdot x)(42) = \lambda x.42 \cdot x$

Hinweis: auch (arithmetische) Operationen sind grundsätzlich gecurryte Funktionen!
Bsp: $x + y = ((\lambda a.\lambda b.a + b)(x))(y)$
$5 + y = ((\lambda a.\lambda b.a + b)(5))(y) = (\lambda b.5 + b)(y)$

Unterversorgbar: `(5+)` ≡ `\b -> 5+b`

Bei Infixoperatoren: Erhöhe Listeneinträge um 5

```
add5 :: [Integer] -> [Integer]
add5 list = map (5+) list
```

Noch kürzer: Unterversorgung von `map`

```
add5 :: [Integer] -> [Integer]
add5 = map (5+)
```

Anwendungsbeispiel: Erkenne Alphabetzeichen

```
isAlpha :: Char -> Bool
isAlpha = isIn "abcdefghijklmnopqrstuvwxyz"
```

Unterschied Currying/Tupelargumente

Funktion: Berechnung von $a \cdot x^2$

Gecurryt

```
f :: Double -> Double -> Double
f a x = a * x * x
```

Mit Tupeln

```
g :: (Double, Double) -> Double
g (a,x) = a * x * x
```

Definitionen verschieden!

- `f` ist gecurryte Funktion mit zwei Argumenten
- `g` ist Funktion mit einem Tupel als Argument

Bemerkung:

- Haskell bietet eingebaute Tupel, z.B. `(2,5)` oder `(True,[],42)`. Näheres s.u.

Vergleich:

- Tupelschreibweise entspricht mathematischer Schreibweise
- Kann aber nicht unterversorgt werden!

List Comprehension

```
i ∈ [a, b] === i <- [a, b]
```

Der aufeinanderfolgende Generator verfeinert die Ergebnisse des vorherigen Generators:

```
take 10 [ (i,j) | i <- [1,2],
                  j <- [1..] ]
```

```
> [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]
```

Lokal let Deklaration:

```
take 10 [ (i,j) | i <- [1..],
                  let k = i*i,
                  j <- [1..k] ]
> [(1,1),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4),(3,5)]
```

verschachtelte Folge von List Comprehensions:

```
take 5 [ [ (i,j) | i <- [1,2] ] | j <- [1..] ]
> [[(1,1),(2,1)], [(1,2),(2,2)], [(1,3),(2,3)], [(1,4),(2,4)], [(1,5),(2,5)]]
```

Aufgaben

Breitensuche für Binary Tree mit Queue

Queue front back

Q front back stellt die Liste dar, die durch Konkatenation der *Vorderseite* front und der Umkehrung der *Rückseite* back entsteht. So stellen diese drei Haskell-Werte die gleiche Liste dar:

```
Q [1,2,3,4,5] []
Q [1,2,3] [5,4]
Q [] [5,4,3,2,1]
```

enqueue - add element to the end of queue

```
enqueue :: a -> Queue a -> Queue a
enqueue x (Q front back) = Q front (x:back)
```

Ist die von q dargestellte Liste leer, gilt dequeue q == Nothing. Ansonsten teilt sich q in das vorderste Element x und den Rest q', und dequeue q == Just (x,q').

```
dequeue :: Queue a -> Maybe (a, Queue a)
dequeue (Q [] []) = Nothing
dequeue (Q [] back) = dequeue (Q (reverse back) [])
dequeue (Q (x:front) back) = Just (x, Q front back)
```

Breitensuche

```
bfs (N (N (N L 4 L) 2 L) 1 (N L 3 L))
> [1, 2, 3, 4]
```

```
bfs :: Tree a -> [a]
bfs t = go (fromList [t])
where
  go q = go2 (dequeue q)
  go2 Nothing = []
  go2 (Just (Leaf, q')) = go q'
  go2 (Just (Node l x r, q')) = x : go (enqueue r (enqueue l q'))
```

Merge Listen

Das einfache merge

merge kann zwei (potenziell unendliche) sortierte Listen zu einer sortierten Liste verschmelzen

```
merge [] bs = bs
merge as [] = as
merge (a:as) (b:bs)
  | a <= b = a : merge as (b:bs)
  | otherwise = b : merge (a:as) bs
```

Undendliche merge

mergeAll fügt eine Liste von Listen zu einer einzelnen sortierten Liste zusammen

```
> mergeAll [ [6, 8, 12],
             [8, 10, 14],
             [105,107,111] ]
> [6,8,8,10,12,14,105,107,111]

mergeAll (x:y:rest) = mergeAll (merge x y : rest)
mergeAll [rest] = rest
mergeAll [] = []
```

Alternative Konstruktion:

```
mergeAll ((f:fs):gs:rest) = f : mergeAll (merge fs gs : rest)
```

Pseudozufallzahlen

```
b = 13849
a = 25173
m = 2^16
x0 = 32
```

Mit tail

```
rand :: [Integer]
rand = tail (iterate (\x -> ((a * x + b) `mod` m )) x0)
```

show (take 5 rand):
`[32953,50566,6039,55612,20229]`

Ohne tail

```
rand :: [Integer]
rand = iterate (\x -> ((a * x + b) `mod` m )) x0
```

show (take 5 rand):
`[32,32953,50566,6039,55612,20229]`

-> tail ist hier wichtig!

Quick Sort

Pattern Matching

```
qsort:: (Ord t) => [t] a [t]
qsort [] = []
qsort (p:ps) = (qsort (filter (\x -> x <= p) ps))
        ++ p : (qsort (filter (\x -> x > p) ps))
```

List Comprehension

```
qsort [] = []
qsort (p:ps) = (qsort [x | x <- ps, x <= p])
         ++ p: (qsort [x | x <- ps, x > p])
```

SplitWhen

```
splitWhen even [1,2,3] ==>([1],[2,3])
splitWhen (=='o') "Hello, World!" ==>("Hell","o, World!")

splitWhen :: (a -> Bool) -> [a] -> ([a],[a])
splitWhen _ [] = ([],[])
splitWhen p (x:xs)
  | p x = ([],x:xs)
  | otherwise = let (ys,zs) = splitWhen p xs in (x:ys,zs)
```

```
Alternativ:
splitWhen' p xs = (takeWhile (not . p) xs, dropWhile (not . p) xs)
```

Generator von Pseudozufallszahlen

Generator

```
gen a b m x0 = iterate (\x -> (a * x + b) `mod` m) x0
rand = gen 25173 13849 (2^16) 32
```

Münze

Funktion toCoin wandelt den Stream rand in einen Stream von „Münzwürfen“ vom Typ [Bool] um.

```
toCoin :: [Integer] -> [Bool]
```

```
-- 1. Idee: Betrachte das niedrigstwertige Bit.
--
-- Beobachtung:
-- Man erhält eine alternierende Folge von True und False.
-- Dies liegt daran, dass bei linearen Kongruenzgeneratoren,
-- die für den Parameter m eine Zweierpotenz und für a,b
-- ungerade Zahlen verwenden das niedrigstwertige
-- Bit der erzeugten Zahlen immer alterniert.
toCoin' = map odd

-- Für rand geeignet: Betrachte das höchstwertige Bit.
toCoin = map (>= 2^15)
```

Hamming-Stream

Definieren Sie eine unendliche Liste hamming :: [Integer] aller Hamming-Zahlen, also aller Zahlen h der Gestalt

```
h = 2^i · 3^j · 5^k mit i, j, k >= 0
```

```
hamming = 1 : (map (2*) hamming) `merge`
              (map (3*) hamming) `merge`
              (map (5*) hamming)

hamming = 1 : concat [[2*h,3*h,5*h] | h <- hamming]
```

AST

Primzahlen

Primzahlensieb: Liste aller Primzahlen <= n

Rekursiv

```
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
```

Mit List Comprehension

```
oddPrimes (p : ps) = p : (oddPrimes [p' | p' <- ps, p' `mod` p /= 0])
primes = 2 : oddPrimes (tail odds)
```

Primepowers

Funktion primepowers, die für einen gegebenen Parameter n die unendliche Liste der ersten n Potenzen aller Primzahlen
berechnet, aufsteigend sortiert. D.h., primepowers n enthält in aufsteigender Reihenfolge genau die Elemente der Menge `{p^i | p Primzahl, 1 <= i <= n}`

```
primepowers :: Integer -> [Integer]
primepowers n = foldr merge [] [map (^i) primes | i <- [1..n]]
```

List Builder

```
data ListBuilder a
= Nil
| Cons a (ListBuilder a)
| Append (ListBuilder a) (ListBuilder a)
```

fromList überführt die eine Liste in eine gültige Darstellung als ListBuilder

```
fromList :: [a] -> ListBuilder a
fromList = foldr Cons Nil
```

rev lb soll die umgedrehte von lb dargestellte Liste darstellen.

```
rev :: ListBuilder a -> ListBuilder a
rev Nil = Nil
rev (Cons x xs) = Append (rev xs) (Cons x Nil)
rev (Append xs ys) = Append (rev ys) (rev xs)
```

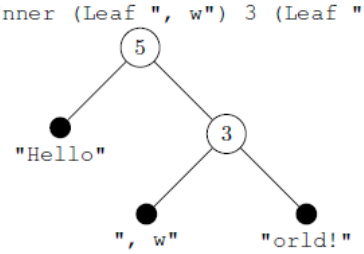
toList lb = toListAcc lb [] gibt die von einem ListBuilder dargestellte Liste zurück

```
toList :: ListBuilder a -> [a]
toList lb = toListAcc lb []
toListAcc Nil acc = acc
toListAcc (Cons x xs) acc = x : toListAcc xs acc
toListAcc (Append xs ys) acc = toListAcc xs (toListAcc ys acc)
```

Ropes

```
data Rope
  = Leaf String
  | Inner Rope Int Rope
```

```
Inner (Leaf "Hello") 5
      (Inner (Leaf ", w") 3 (Leaf "orld!"))
```



(a) Implementieren Sie die Funktion [5 Punkte]

`ropeLength :: Rope -> Int`

die die Länge der durch das Rope dargestellten Zeichenkette berechnet. Nutzen Sie das Gewicht innerer Knoten, um möglichst wenige Knoten zu besuchen.

(b) Implementieren Sie die Funktion [2 Punkte]

`ropeConcat :: Rope -> Rope -> Rope`

die die übergebenen Ropes verkettet. Benutzen Sie `ropeLength` zur Berechnung des Gewichts.

(c) Implementieren Sie die Funktion [11 Punkte]

`ropeSplitAt :: Int -> Rope -> (Rope, Rope)`

`ropeSplitAt i r` zerlegt das Rope `r` der Länge n, das den String $c_0 ... c_{n-1}$ darstellt, an Index `i` in zwei Teilropes: Das erste Teilrope stellt den String $c_0 ... c_{i-1}$ dar, das zweite Teilrope $c_i ... c_{n-1}$. Für Indexwerte außerhalb des Intervalls $[0, n]$ ist die Funktion unspezifiert.

Verwenden Sie das Gewicht innerer Knoten, um die Spaltposition zu finden. Die Listenfunktionen `drop, take :: Int -> [a] -> [a]` sind hilfreich, um die Strings in den Blattknoten zu zerlegen.

Beispiel[1]:

```
> let (l, r) = ropeSplitAt 6 (fromString "Hello, world!")
> toString l
"Hello,"
> toString r
" world!"
```

```
ropeLength :: Rope -> Int
ropeLength (Leaf s) = length s
ropeLength (Inner _ w r) = w + ropeLength r
```

```
ropeConcat :: Rope -> Rope -> Rope
```

```
ropeConcat l r = Inner l (ropeLength l) r
```

Idee: falls i < w: schneide linke Teilbaum, konkateniere lr und r
falls i > w: schneide rechte Teilbaum...
sonst sind die Bäume schon richtig geschnitten

```
ropeSplitAt :: Int -> Rope -> (Rope, Rope)
ropeSplitAt i (Leaf s) = (Leaf (take i s), Leaf (drop i s))
ropeSplitAt i (Inner l w r)
| i < w
  = let (ll, lr) = ropeSplitAt i l
  in (ll, ropeConcat lr r)
| i > w
  = let (rl, rr) = ropeSplitAt (i-w) r
  in (ropeConcat l rl, rr)
| otherwise
  = (l, r)
```

Proper Divisors

Aufgabe 1 (Haskell: Vollkommene Zahlen) [15 Punkte]

Eine Zahl $n \geq 2$ heißt vollkommen, wenn die Summe ihrer echten Teiler (Teiler außer n selbst) wieder n ergibt.
Beispielsweise sind 6 und 28 vollkommene Zahlen, da $1 + 2 + 3 = 6$ und $1 + 2 + 4 + 7 + 14 = 28$.

(a) Implementieren Sie die Funktionen [5 Punkte]

```
properDivisors :: Integer -> [Integer]
perfectNumber :: Integer -> Bool
```

`properDivisors n` berechnet die Liste der echten Teiler von $n \geq 2$, und `perfectNumber n` soll genau dann gelten, wenn n eine vollkommene Zahl ist.

Beispiellösung:

```
properDivisors :: Integer -> [Integer]
properDivisors n = filter (\i -> n `mod` i == 0) [1..n-1]
```

Beispiellösung:

```
perfectNumber :: Integer -> Bool
perfectNumber n = n == sum (properDivisors n)
```

(b) Implementieren Sie eine optimierte Variante von `properDivisors`, die höchstens $\sqrt{n}$ Teilbarkeitsprüfungen durchführt. Dabei gilt: [10 Punkte]

- 1 ist ein echter Teiler von n.
- Ist $2 \leq i < \sqrt{n}$ und ist n durch i teilbar, so sind i und $\frac{n}{i}$ echte Teiler von n.
- Ist $i = \sqrt{n}$ eine natürliche Zahl, so ist i ein echter Teiler von n.

Sie dürfen dafür die Funktion `isqrt :: Integer -> Integer` verwenden, die die abgerundete Quadratwurzel berechnet. Was die Funktion für nicht-positive Eingaben tut, bleibt Ihnen überlassen.

Beispiellösung:

```
properDivisorsOpt :: Integer -> [Integer]
properDivisorsOpt n = 1:concat [ divs i | i <- [2..isqrt n], n `mod` i == 0]
    where divs i | i*i == n = [i]
                 | otherwise = [i, n `div` i]

-- alternativ:
properDivisorsOpt' :: Integer -> [Integer]
properDivisorsOpt' n = 1:properDivisorsHelp 2
   where properDivisorsHelp i
          | i*i < n
             && n `mod` i == 0 = i:(n `div` i):properDivisorsHelp (i+1)
          | i*i < n = properDivisorsHelp (i+1)
          | i*i == n = [i]
          | otherwise = []
```

Acht Damen

Lösungs-Konfigurationen: 8 Damen auf dem Brett

```
solution :: Conf -> Bool
solution board = (length board) == 8
```

Start-Konfiguration: leeres Brett

```
queensSolutions :: [Conf]
queensSolutions = backtrack []
```

Folgekonfigurationen: Platziere weitere Dame in beliebiger Zeile

```
successors :: Conf -> [Conf]
successors board = map (:board) [1..8]
```

Legale Konfigurationen: Neue Dame bedroht keine bestehenden

```
threatens :: Int -> (Int, Int) -> Bool
threatens row1 (diag, row2) = row1 == row2 || abs (row1-row2) == diag


legal :: Conf -> Bool
legal [] = True
legal (row:rest) = not (any (threatens row) (zip [1..] rest))


successors [6,8,5,1] --> [[1,6,8,5,1],[2,6,8,5,1],...,[8,6,8,5,1]]


backtrack :: Conf -> [Conf]
backtrack conf =
```

```
if (solution conf)
then [conf]
else flatten (map backtrack (filter legal (successors conf)))
```

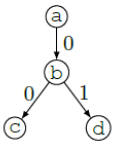
Mehrwegbäume

Gegeben ist der Typ für Mehrwegbäume

```
data Tree a = Node a [Tree a]
```

sowie zwei Beispielbäume

```
t1 = Node "a" [
       Node "b" [
         Node "c" [],
         Node "d" []
       ]
     ]
```




```
t2 = Node "Z" []
```

(a) Der Weg von der Wurzel eines Baumes zu einem Teilbaum kann durch eine Liste natürlicher Zahlen beschrieben werden. Jedes Element solch einer *(gültigen) Position* steht dabei für den Index einer ausgehenden Kante (gezählt ab 0). Alle anderen Listen nennen wir *ungültige Positionen*. [5 Punkte]

Implementieren Sie die Funktion

```
treeIndex :: Tree a -> [Int] -> Tree a
```

sodass der Aufruf `treeIndex t p` den durch die Position `p` angegebenen Teilbaum von `t` zurückgibt. Falls `p` ungültig ist, beenden Sie die Ausführung durch Aufruf von `error`.

Beispiel:

```
          t1          ⇒+ Node "a" [Node "b" [Node "c" [], Node "d" []]]
treeIndex t1 []       ⇒+ Node "a" [Node "b" [Node "c" [], Node "d" []]]
treeIndex t1 [0,1]    ⇒+                                  Node "d" []
treeIndex t1 [1]      ⇒+ ⊥  Exception: invalid position
treeIndex t1 [0,1,0]  ⇒+ ⊥  Exception: invalid position
```

(b) Implementieren Sie eine Funktion [6 Punkte]

```
treePositions :: Tree a -> [[Int]]
```

die eine Liste aller im obigen Sinne gültigen Positionen eines Baumes berechnet.

Beispiel:

```
treePositions t1 == [ [], [0], [0,0], [0,1] ]
treePositions t2 == [ [] ]
```

Hinweis: Sie können List Comprehensions verwenden.

(c) Implementieren Sie eine Funktion [10 Punkte]

```
changeTree :: (Tree a -> Tree a) -> [Int] -> Tree a -> Tree a
```

Der Aufruf `changeTree f p t` ändert in `t` den durch `p` beschriebenen Teilbaum ab, indem er `f` auf diesen Teilbaum anwendet. Für ungültiges `p` wird `t` unverändert zurückgegeben.

Beispiel:

```
let f (Node _ xs) = Node "Z" xs
                      t1 ⇒+ Node "a" [Node "b" [Node "c" [], Node "d" []]]
changeTree f []       t1 ⇒+ Node "Z" [Node "b" [Node "c" [], Node "d" []]]
changeTree f [0]      t1 ⇒+ Node "a" [Node "Z" [Node "c" [], Node "d" []]]
changeTree f [0,1]    t1 ⇒+ Node "a" [Node "b" [Node "c" [], Node "Z" []]]
changeTree f [0,1,0]  t1 ⇒+ Node "a" [Node "b" [Node "c" [], Node "d" []]]
```


(d) Verwenden Sie nun `changeTree` zur Implementierung einer Funktion [5 Punkte]

```
overrideTree :: Tree a -> [Int] -> Tree a -> Tree a
```

Der Aufruf `overrideTree t' p t` ersetzt in `t` den durch `p` beschriebenen Teilbaum durch `t'`. Für ungültiges `p` wird `t` unverändert zurückgegeben.

Beispiel:

```
                          t1 ⇒+ Node "a" [Node "b" [Node "c" [], Node "d" []]]
overrideTree t2 []        t1 ⇒+ Node "Z" []
overrideTree t2 [0]       t1 ⇒+ Node "a" [Node "Z" []]
overrideTree t2 [0,1,0]   t1 ⇒+ Node "a" [Node "b" [Node "c" [], Node "d" []]]
```

```
ith :: [a] -> Int -> a
ith [] i = error "invalid index"
ith (x:xs) 0 = x
ith (x:xs) i = ith xs (i-1)

treeIndex :: Tree a -> [Int] -> Tree a
treeIndex t [] = t
treeIndex (Node a ts) (i:is)
| i < 0 = error "invalid position"
| i >= length ts = error "invalid position"
| otherwise = treeIndex (ith ts i) is

treePositions :: Tree a -> [[Int]]
treePositions (Node _ ts) =
      [] : [ (i : is ) | i <- [0..(length ts - 1)],
      is <- treePositions (ith ts i) ]

changeListAt :: (a -> a) -> Int -> [a] -> [a]
changeListAt f _ [] = []
changeListAt f 0 (x:xs) = (f x : xs)
changeListAt f n (x:xs) = x : changeListAt f (n-1) xs
```

```
changeTree :: (Tree a -> Tree a) -> [Int] -> Tree a -> Tree a
changeTree f [] t = f t
changeTree f (i:is) t@(Node a ts)
| i < 0 = t
| i >= length ts = t
| otherwise = Node a (changeListAt (changeTree f is) i ts)

overrideTree :: Tree a -> [Int] -> Tree a -> Tree a
overrideTree t = changeTree (\x -> t)
```

Huffman Tree to List of Tuples [(Char, Bitfolge)]

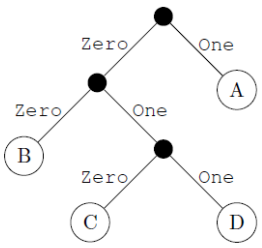
```
data Bit = Zero | One

data HuffmanTree = Node HuffmanTree HuffmanTree
                 | Leaf Char
```

Beispiel:

Der nebenstehende Huffman-Baum repräsentiert folgende Codierung:

```
'A'  ↔  [One]
'B'  ↔  [Zero, Zero]
'C'  ↔  [Zero, One, Zero]
'D'  ↔  [Zero, One, One]
```



Als Haskell-Datum sieht der Baum so aus:

```
Node (Node (Leaf 'B')
           (Node (Leaf 'C')
                 (Leaf 'D')))
     (Leaf 'A')
```

```
treeToList :: HuffmanTree -> [(Char, [Bit])]
treeToList (Leaf c) prefix = [(c, prefix)]
treeToList (Node t1 t2) prefix = (treeToList t1 (prefix ++ [Zero])) ++
                                 (treeToList t2 (prefix ++ [One]))
```

Cut Dead Leaves

Aufgabe 2 (Haskell, Bäume) [7 Punkte]

Der Datentyp für Mehrweg-Bäume mit toten/lebendigen Blättern ist definiert als:

```
data Tree  = Node [Tree] | Leaf State          deriving (Show, Eq)
data State = Dead | Alive                      deriving (Show, Eq)
```

Ein Baum ist lebendig, wenn er ein lebendiges Blatt besitzt. Andernfalls ist er tot.

Geben Sie eine Funktion

```
prune :: Tree -> Tree
```

an, die alle toten Teilbäume „entfernt".

(a)
```
prune :: Tree -> Tree
prune (Leaf s)  = Leaf s
prune (Node ts) = Node removed
  where removed = filter (not . isDead) (map prune ts)
        isDead (Leaf Dead)   = True
        isDead (Node [])     = True
        isDead _             = False
```