Haskell

Contents

[Befehle aus Prelude 3](#_Toc67929280)

[map 3](#_Toc67929281)

[fold 4](#_Toc67929282)

[foldr (right) 4](#_Toc67929283)

[foldl (left) 4](#_Toc67929284)

[Beispiele 4](#_Toc67929285)

[filter 4](#_Toc67929286)

[iterate 4](#_Toc67929287)

[zipWith 5](#_Toc67929288)

[replicate 5](#_Toc67929289)

[flatten==concat 5](#_Toc67929290)

[not 6](#_Toc67929291)

[odd und even 6](#_Toc67929292)

[mod und div 6](#_Toc67929293)

[takeWhile und dropWhile 6](#_Toc67929294)

[fst und snd 6](#_Toc67929295)

[Befehle außer Prelude 6](#_Toc67929296)

[Modul Data.Char 6](#_Toc67929297)

[ord 6](#_Toc67929298)

[chr 7](#_Toc67929299)

[FAQ 7](#_Toc67929300)

[Allgemein 7](#_Toc67929301)

[Ein Intervall erstellen 7](#_Toc67929302)

[Head of emtpy list 7](#_Toc67929303)

[cons (:) vs append (++) 7](#_Toc67929304)

[Wie konvertiert man Zahl nach String? 7](#_Toc67929305)

[Wann wird ein Akkumulator benötight? 8](#_Toc67929306)

[Was ist Prelude? 8](#_Toc67929307)

[Wie schreibt man in Haskell != (nicht gleich) ? 8](#_Toc67929308)

[Power: \*\* oder ^ ? 8](#_Toc67929309)

[Was macht Dot-Operator (.)? 8](#_Toc67929310)

[Wie kann man ein Predikat p negieren? 8](#_Toc67929311)

[map vs iterationen mit (x:xs) 8](#_Toc67929312)

[Unendliche Listen 9](#_Toc67929313)

[Wie erstellt man am besten eine unendliche Liste mit einer Zahl/Symbol? 9](#_Toc67929314)

[Wie erstellt man eine unedliche Liste als Folge? 9](#_Toc67929315)

[Theorie 9](#_Toc67929316)

[Kontrollflus 9](#_Toc67929317)

[If‘s 9](#_Toc67929318)

[Guard 9](#_Toc67929319)

[Pattern Matching 9](#_Toc67929320)

[Lokale Bindung 10](#_Toc67929321)

[let 10](#_Toc67929322)

[where 10](#_Toc67929323)

[Polymorphe Datentypen 10](#_Toc67929324)

[Maybe: Optionale Werte 10](#_Toc67929325)

[Either: Summen-Typ 10](#_Toc67929326)

[Algebraische Datentypen 10](#_Toc67929327)

[Shape 10](#_Toc67929328)

[Rekursive Datentypen 11](#_Toc67929329)

[Stack 11](#_Toc67929330)

[Binary Tree 11](#_Toc67929331)

[Akkumulator 11](#_Toc67929332)

[Ohne Akkumulator 11](#_Toc67929333)

[Mit Akkumulator 12](#_Toc67929334)

[Endrekursion 12](#_Toc67929335)

[Linear, aber nicht endrekursiv 12](#_Toc67929336)

[Endrekursiv 12](#_Toc67929337)

[Typklassen 12](#_Toc67929338)

[Neue Typlassen erstellen 12](#_Toc67929339)

[Hierarchie 13](#_Toc67929340)

[Automatische Instanziierung 13](#_Toc67929341)

[Enum 14](#_Toc67929342)

[Funktionen 14](#_Toc67929343)

[Infix-Notation und Präfix-Notation 14](#_Toc67929344)

[Currying und Unterversorgung 15](#_Toc67929345)

[List Comprehension 15](#_Toc67929346)

[Aufgaben 16](#_Toc67929347)

[Breitensuche für Binary Tree mit Queue 16](#_Toc67929348)

[Queue front back 16](#_Toc67929349)

[Breitensuche 17](#_Toc67929350)

[Merge Listen 17](#_Toc67929351)

[Das einfache merge 17](#_Toc67929352)

[Undendliche merge 17](#_Toc67929353)

[Pseudozufallzahlen 18](#_Toc67929354)

[Mit tail 18](#_Toc67929355)

[Ohne tail 18](#_Toc67929356)

[Quick Sort 18](#_Toc67929357)

[Pattern Matching 18](#_Toc67929358)

[List Comprehension 18](#_Toc67929359)

[SplitWhen 18](#_Toc67929360)

[Generator von Pseudozufallszahlen 19](#_Toc67929361)

[Generator 19](#_Toc67929362)

[Münze 19](#_Toc67929363)

[Hamming-Stream 19](#_Toc67929364)

[AST 20](#_Toc67929365)

[Primzahlen 20](#_Toc67929366)

[Rekursiv 20](#_Toc67929367)

[Mit List Comprehension 20](#_Toc67929368)

[Primepowers 20](#_Toc67929369)

[List Builder 20](#_Toc67929370)

[Ropes 21](#_Toc67929371)

[Proper Divisors 23](#_Toc67929372)

# 

# 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]

## 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)

## 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 а [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“

# Befehle außer Prelude

## 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)

### 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

# 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’

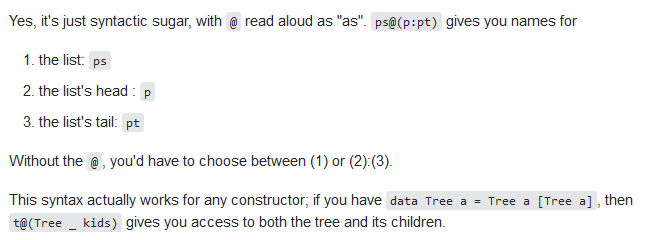
caesar n [] = []  
caesar n (x:xs) = (caesar\_one n x) : (caesar n xs)

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)



## 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 **\_** istdie 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

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)

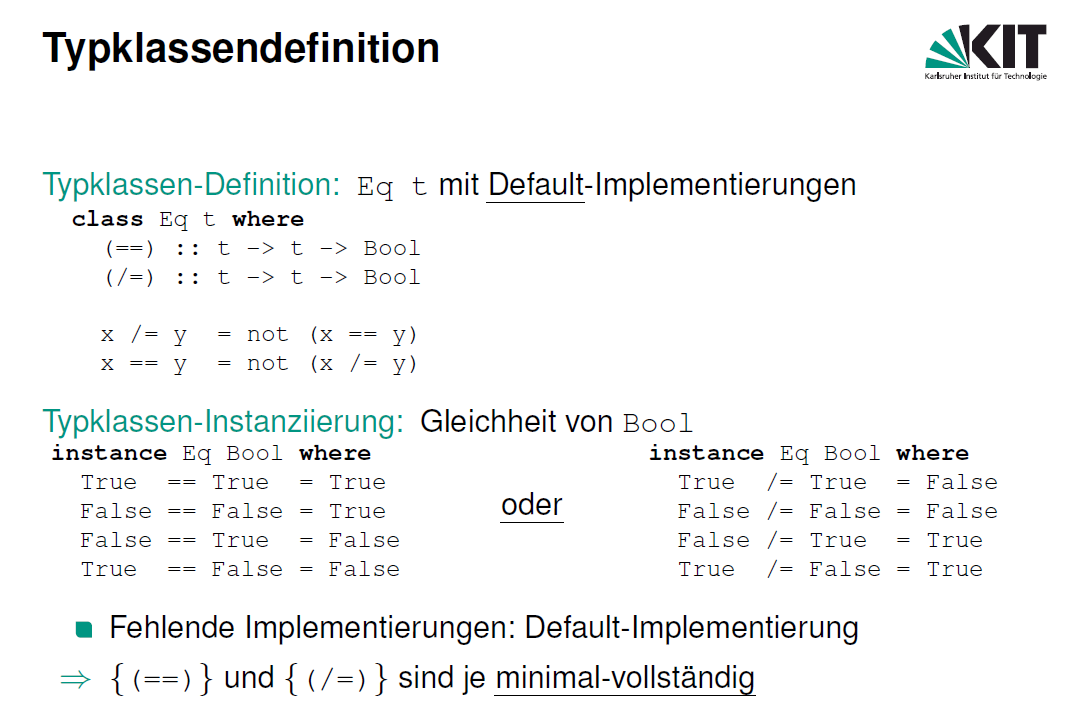
fak n = fakAcc n 1

## Typklassen

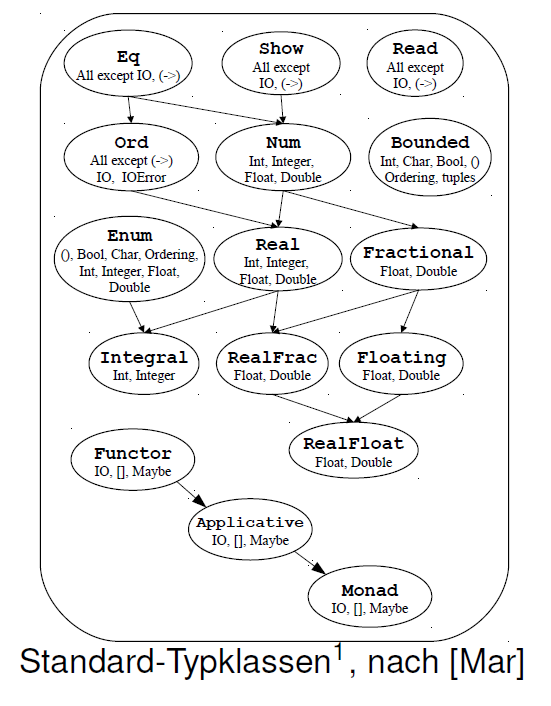
### Neue Typlassen erstellen

Man kann eigene Typklassen erstellen mit

class X where

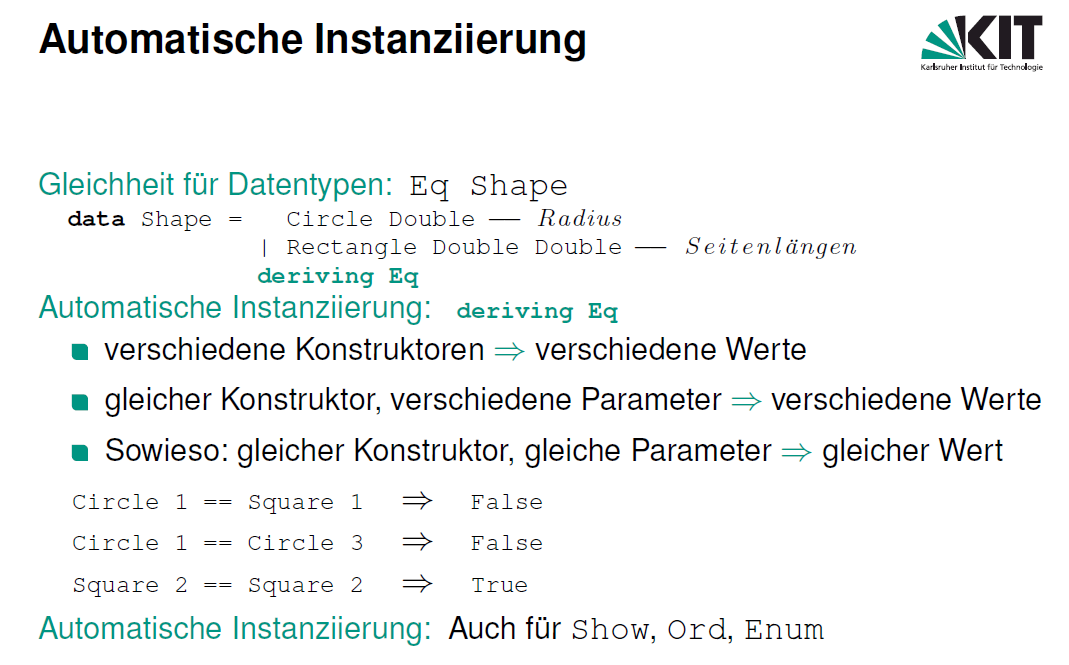


### Hierarchie



### Automatische Instanziierung

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



### 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

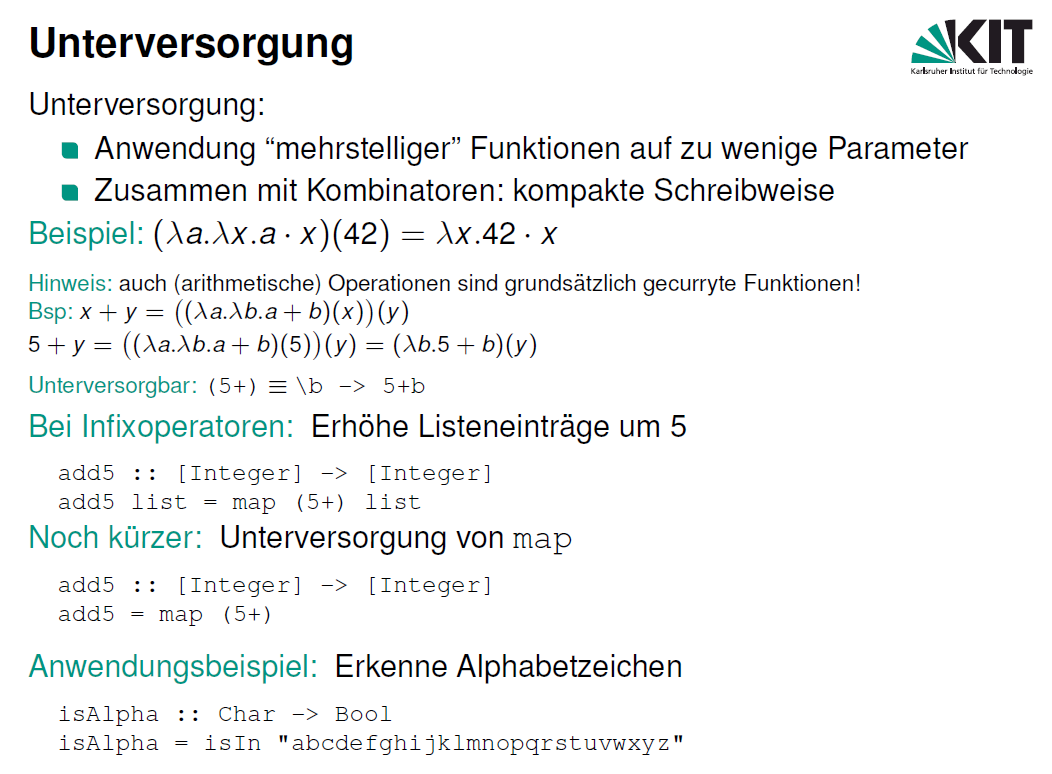
Currying: Ersetzung einer mehrstelligen Funktion durch Schachtelung einstelliger Funktionen

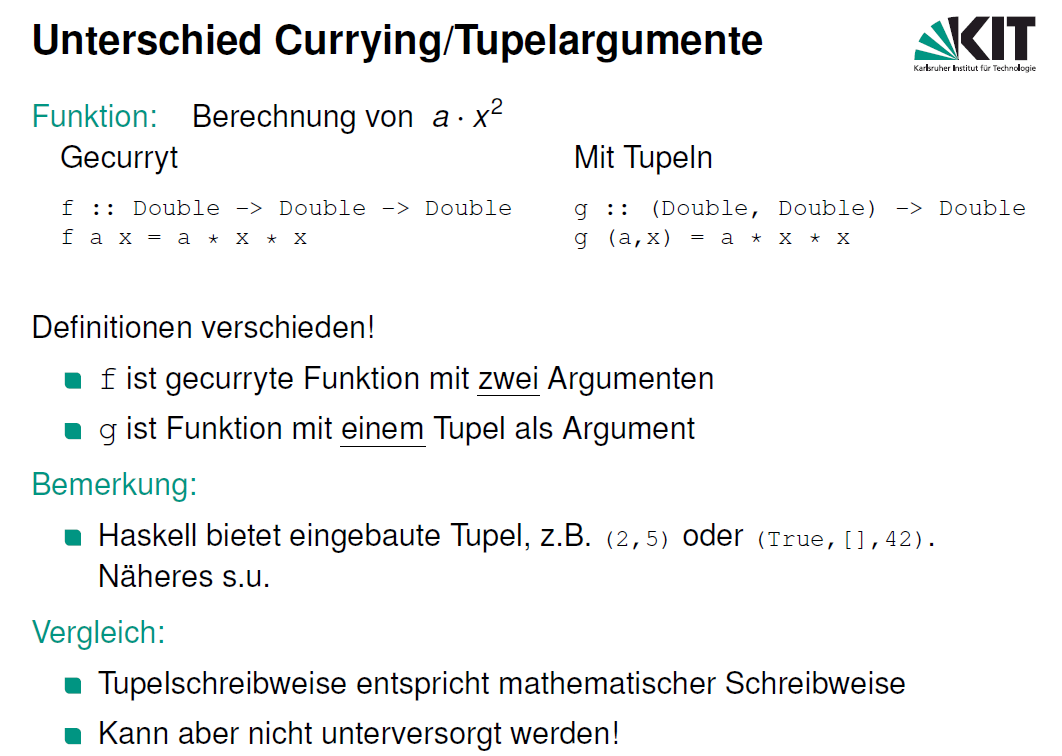
Funktionsanwendung ist links-assoziativ:

f 3 7 === (f 3) 7

Konsequenz: Funktionstypen -> sind rechts-assoziativ:

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





## 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] а [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

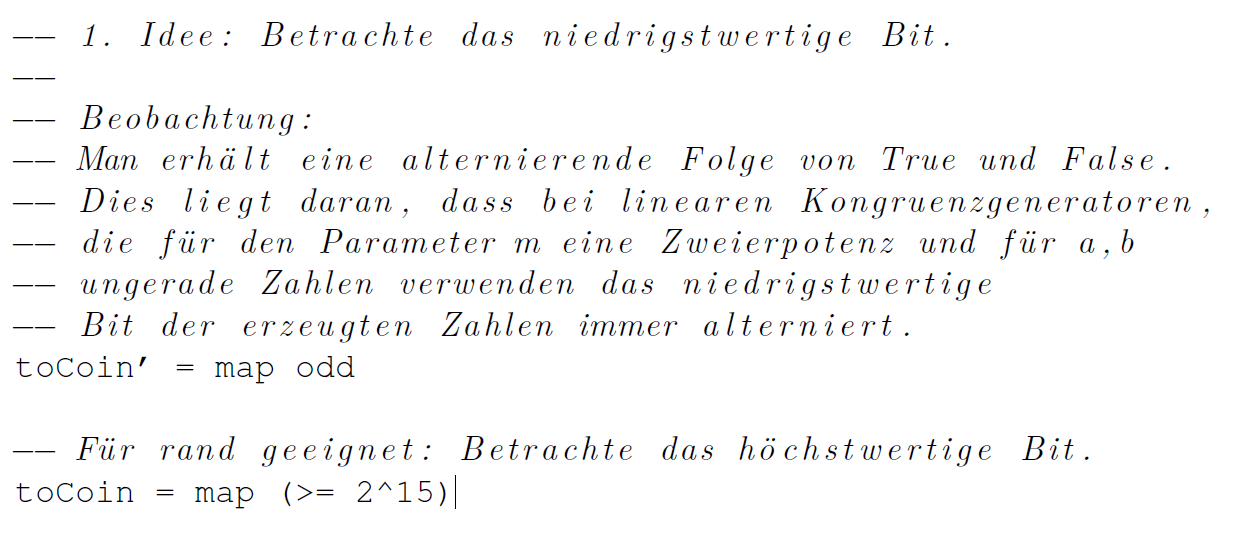
gen a b m x0 = iterate (\x -> (a \* x + b) ‘mod‘ m) x0

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]



## 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]

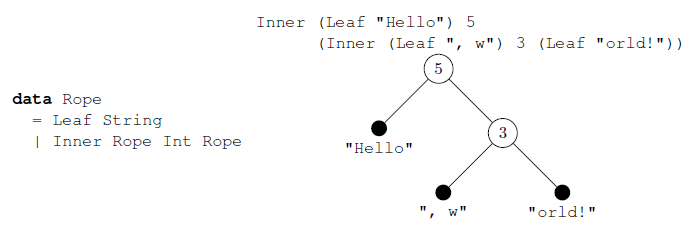
toList lb = toListAcc lb []

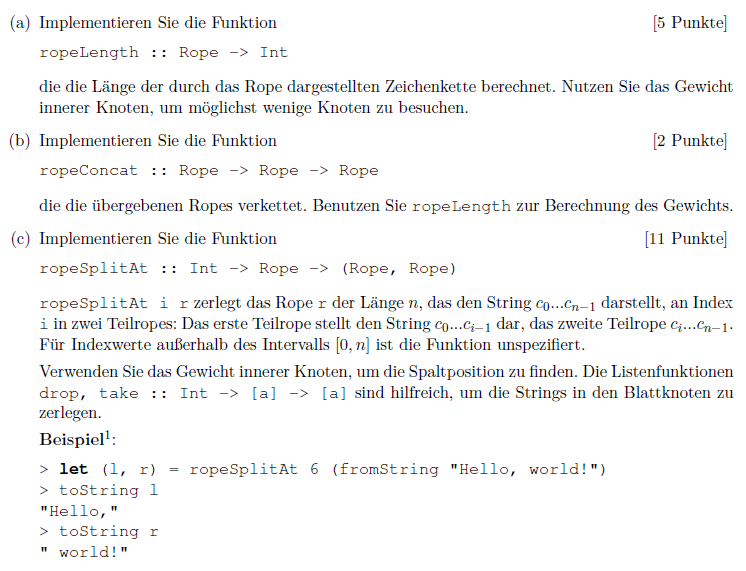
toListAcc Nil acc = acc

toListAcc (Cons x xs) acc = x : toListAcc xs acc

toListAcc (Append xs ys) acc = toListAcc xs (toListAcc ys acc)

## Ropes





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

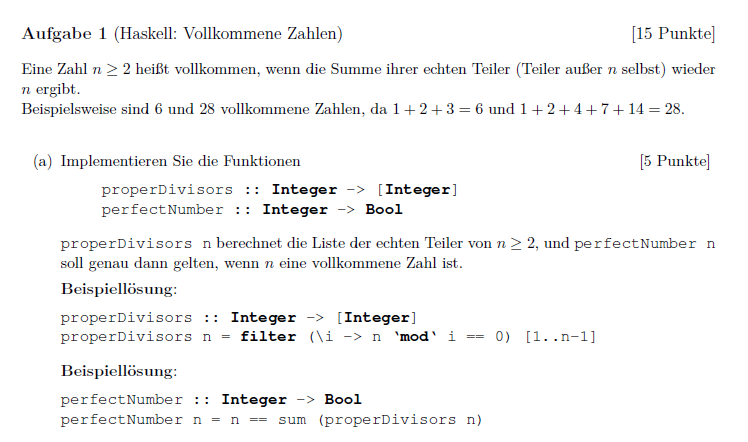
= let (rl, rr) = ropeSplitAt (i-w) r

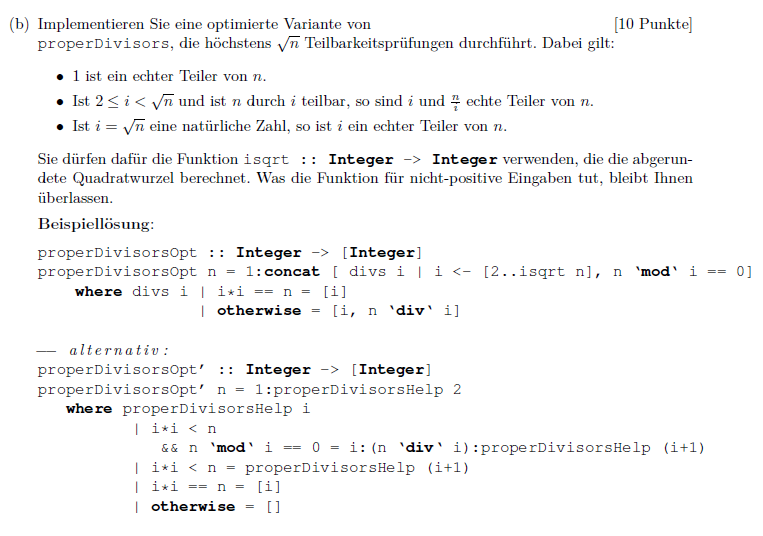
in (ropeConcat l rl, rr)

| otherwise

= (l, r)

## Proper Divisors





## 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]

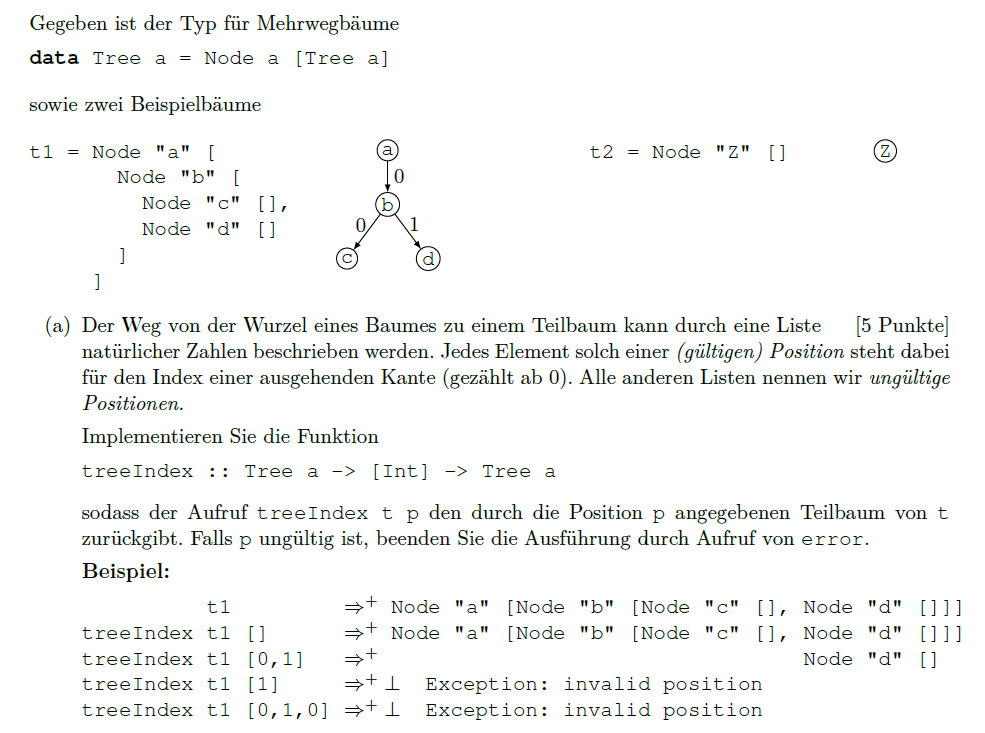
backtrack conf =

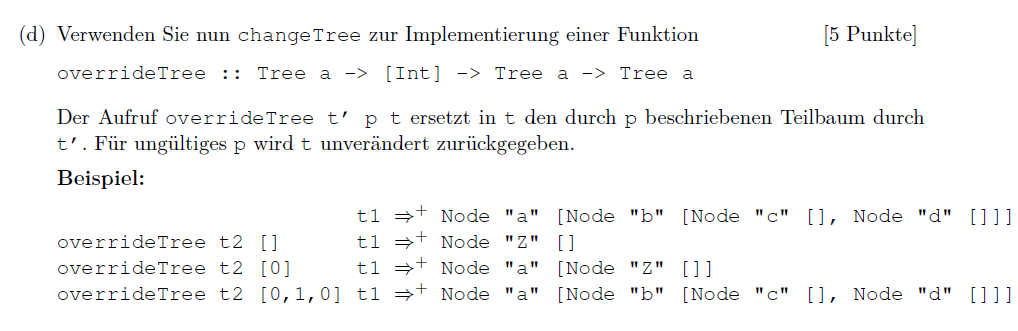
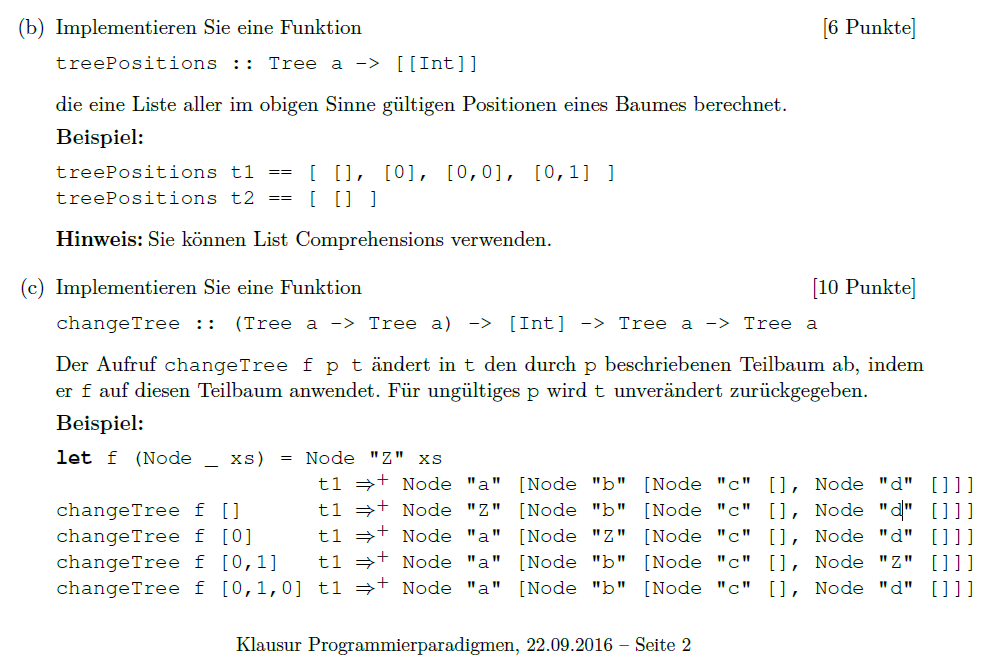
if (solution conf)

then [conf]

else flatten (map backtrack (filter legal (successors conf)))

## Mehrwegbäume





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)