Haskell

Contents

[Befehle 3](#_Toc67510949)

[map 3](#_Toc67510950)

[fold 3](#_Toc67510951)

[foldr (right) 3](#_Toc67510952)

[foldl (left) 3](#_Toc67510953)

[Beispiele 3](#_Toc67510954)

[filter 4](#_Toc67510955)

[iterate 4](#_Toc67510956)

[zipWith 4](#_Toc67510957)

[FAQ 4](#_Toc67510958)

[Allgemein 4](#_Toc67510959)

[Ein Intervall erstellen 4](#_Toc67510960)

[Head of emtpy list 5](#_Toc67510961)

[cons (:) vs append (++) 5](#_Toc67510962)

[Wie konvertiert man Zahl nach String? 5](#_Toc67510963)

[Unendliche Listen 5](#_Toc67510964)

[Wie erstellt man am besten eine unendliche Liste mit einer Zahl/Symbol? 5](#_Toc67510965)

[Wie erstellt man eine unedliche Liste als Folge? 5](#_Toc67510966)

[Wann wird ein Akkumulator benötight? 6](#_Toc67510967)

[Theorie 6](#_Toc67510968)

[Kontrollflus 6](#_Toc67510969)

[If‘s 6](#_Toc67510970)

[Guard 6](#_Toc67510971)

[Pattern Matching 6](#_Toc67510972)

[Lokale Bindung 6](#_Toc67510973)

[let 6](#_Toc67510974)

[where 6](#_Toc67510975)

[Polymorphe Datentypen 7](#_Toc67510976)

[Maybe: Optionale Werte 7](#_Toc67510977)

[Either: Summen-Typ 7](#_Toc67510978)

[Algebraische Datentypen 7](#_Toc67510979)

[Shape 7](#_Toc67510980)

[Rekursive Datentypen 7](#_Toc67510981)

[Stack 7](#_Toc67510982)

[Binary Tree 8](#_Toc67510983)

[Endrekursion 8](#_Toc67510984)

[Akkumulator 8](#_Toc67510985)

[Aufgaben 8](#_Toc67510986)

[Breitensuche für Binary Tree mit Queue 8](#_Toc67510987)

[Queue front back 8](#_Toc67510988)

[Breitensuche 8](#_Toc67510989)

[Merge Listen 9](#_Toc67510990)

[Das einfache merge 9](#_Toc67510991)

[Undendliche merge 9](#_Toc67510992)

# Befehle

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]

## 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]

# 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“]

## 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..]

## Wann wird ein Akkumulator benötight?

# 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)

## Endrekursion

## Akkumulator

# 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)