fromIntegral

averageThree :: Int -> Int -> Int -> Double

averageThree x y z = (fromIntegral (x+y+z)) / 3

Funktionen:

fst :: (a, b) -> a

snd :: (a, b) -> b

head :: [a] -> a

tail :: [a] -> [a]

(++) :: [a] -> [a] -> [a]

(!!) :: [a] -> Int -> a (a[i])

take :: Int -> [a] -> [a]

drop :: Int -> [a] -> [a]

(!!) :: [a] -> Int -> a

last :: [a] -> a

init :: [a] -> [a] (alle ausser letztes)

reverse :: [a] -> [a]

length :: Foldable t => t a -> Int

map :: (a -> b) -> [a] -> [b]

filter :: (a -> Bool) -> [a] -> [a]

elem :: Eq a => a -> [a] -> Bool (contains)

maximum, minimum :: Ord a => [a] -> a

sum, product :: Num a => [a] -> a

zip :: [a] -> [b] -> [(a,b)]

zip ['a','b'] [12,19,22]~> [('a',12),('b',19)]

concat :: [[a]] -> [a]

concat :: Foldable t => t [a] -> [a]

and :: Foldable t => t Bool -> Bool

sum :: (Foldable t, Num a) => t a -> a

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith (+) [1,2,3] [10,11,12]~>[11,13,15]

Präzedenz:

((f a) b) infixl

(.) infixr

(^) **infixr** 2^3^4 => 2^(3^4)

(\*) (/) **infixl** 4/2/2 => (4/2)/2

(+) (-) infixl

(++) (:) infixr

(==) (/=) (<) (>) (<=) (>=) infix

(&&) infixr

(||) infixr

Lambda expressions:

f c acc = L c : acc vs. (\c acc -> L c : acc)

(\a b -> (a . a) b) (+1) 5 ~> 7

incAll :: [Int] -> [Int]

incAll xs = map (\x -> x + 1) xs

incAll,incAll2,incAll3 :: [Int] -> [Int]

incAll = map (\x -> x+1)

incAll2 a = map (\x -> x+1) a

incAll3 = \a -> map (\x -> x+1) a

addToAll …addToAll4 :: Int -> [Int] -> [Int]

addToAll a = map (\x -> a+x)

addToAll2 a b = map (\x -> a+x) b

addToAll3 = \a -> \b -> map (\x -> a+x) b

addToAll4 = \a -> map (\x -> a+x)

keepOld …keepOld3 :: [Int] -> [Int]

keepOld = filter (\x -> x >= 90)

keepOld2 a = filter (\x -> x >= 90) a

keepOld3 = \a -> filter (\x -> x >= 90) a

dropShort …dropShortSS :: [String] -> [String]

dropShort = \a -> filter (\x -> length x > 1) a

dropShortS ss = filter (\x -> length x > 1) ss

dropShortSS = filter (\x -> length x > 1)

Evaluation:

pyth (square 2) 3

~> pyth (2 \* 2) 3 -- using (sqr)

~> pyth 4 3 -- arithmetic

~> square 4 + square 3 -- using (py)

~> 4 \* 4 + square 3 -- using (sqr)

~> 4 \* 4 + 3 \* 3 -- using (sqr)

~> 16 + 3 \* 3 -- arithmetic

~> 16 + 9 -- arithmetic

~> 25 -- arithmetic

reverse' :: [a] -> [a]

reverse' [] = [] --(rev.0)

reverse' (a:as) = (reverse' as) ++ [a] --(rev.1)

Evaluieren Sie schrittweise reverse' [1,2,3]

reverse' [1,2,3]

~> (reverse' [2,3]) ++ [1] -- by (rev.1)

~> ((reverse' [3]) ++ [2]) ++ [1] -- by (rev.1)

~> (((reverse' []) ++ [3]) ++ [2]) ++ [1] -- by (rev.1)

~> ((([]) ++ [3]) ++ [2]) ++ [1] -- by (rev.0)

~> (([3]) ++ [2]) ++ [1] -- by (++)

~> ([3,2]) ++ [1] -- by (++)

~> [3,2,1] -- by (++)

Partial Application:

replicate' :: a -> Int -> [a]

replicate' a b = take b (repeat a)

replicateThreeTimes = replicate' 3

Dollar: Schwache Funktionsapplikation statt Klammern

f1 = map even (map inc (filter (>2) [1,2,3]))

f1’ = map even $ map inc $ filter (>2) [1,2,3]

Curry / Uncurry:

aadd :: (Int, Int) -> Int

aadd (a,b) = a+b

curry :: ((a,b) -> c) -> a -> b -> c

curry f = \a -> \b -> f(a,b)

add :: Int -> Int -> Int

add a b = a+b

uncurry :: (a -> b -> c) -> (a,b) -> c

uncurry f = \(a,b) -> f a b

Lambda & Currying:

prod1 x y z = x \* y \* z

prod2 x y = \z -> x \* y \* z

prod3 x = \y -> \z -> x \* y \* z

prod4 = \x -> \y -> \z -> x \* y \* z

test1 :: Eq b => (a -> b) -> [(a,b)] -> Bool

test1 f [] = True

test1 f (x:xs) = (f (fst x) == snd x) && test1 f xs

-- test1 f x = and (map (\(i,o) -> (f i == o) x)

test4 :: Eq a => (a -> [a] -> Bool) -> [ ( (a,[a]) , Bool ) ] -> Bool

test4 f x = test1 (uncurry f) x

Conditionals:

maxGuard :: Int -> Int -> Int

maxGuard a b | a > b = a

| otherwise = b

maxIf :: Int -> Int -> Int

maxIf a b = if a > b

then a

else b

maxCase :: Int -> Int -> Int

maxCase a b = case a > b of

True -> a

False -> b

Pattern Matching & List conditional

descList :: [a] -> String  
descList xs = "list is " ++ case xs of

[] -> "empty."  
 [x] -> "a singleton list."

xs -> "a longer list.“

switchFirstTwo [] : leere Liste

switchFirstTwo [x] : exakt 1

switchFirstTwo (x:y:[]) : exakt 2

switchFirstTwo (x:y:zs) : min 2

type M22 = ((Int,Int),(Int,Int))

add :: M22 -> M22 -> M22

add ((a1,a2),(a3,a4)) ((b1,b2),(b3,b4))=

((a1+b1,a2+b2),(a3+b3,a4+b4))

mulS :: M22 -> Int -> M22

mulS ((a1,a2),(a3,a4)) s =

((a1\*s,a2\*s),(a3\*s,a4\*s))

Where bindings

brainpower :: Int -> Int -> String

brainpower mentalAge age

| iq < low = "lower than avg"

| iq < avg = "average"  
 | iq < high = "higher than avg"

| otherwise = " top 1 %)"

where iq = (mentalAge / age) \* 100

(low,avg,high) = (85,115,135)

cylinder r h = 2 \* topArea + sideArea

where sideArea = 2 \* pi \* r \* h

topArea = pi \* r ^ 2

Lambda Types:

(\x -> x > 9) 6 :: Bool

(\x -> tail x) :: [a] -> [a]

(\(a,b) -> b ++ a) :: ([a], [a]) -> [a]

(\t -> fst) :: p -> (a, b) -> a

(\(x:xs) -> x) :: [a] -> a

(\x y -> head y) 2 :: [a] -> a

\(a,b) -> fst a ++ b :: (([a], b), [a]) -> [a]

filter (\x -> True) :: [a] -> [a]

map (\a -> 1) [] :: Num b => [b]

(\a -> \b -> (a,b,b)) 1 :: Num a => c -> (a, c, c)

(\(x,y) -> \(x,z) -> (x,z+y))

:: Num b => (a1, b) -> (a2, b) -> (a2, b)

(\f -> f 2 == "WOW") :: Num t => (t -> [Char]) -> Bool

Map (\(a:as) -> a ++ as) :: Ungültiger Ausdruck

CompareWithCase:

comparecase :: (Ord a) => a -> a -> Ordering

comparecase x y = case x > y of

True -> GT

False -> case y > x of

True -> LT

False -> EQ

Let&Where:

cuboid :: Float -> Float -> Float -> Float

cuboid l w h = 2\*sideArea + 2\*frontArea + 2\* baseArea

where

baseArea = l\*w

sideArea = h\*w

frontArea = h\*l

cuboid' :: Float -> Float -> Float -> Float

cuboid' l w h = let

baseArea = l\*w

sideArea = h\*w

frontArea = h\*l

in 2\*sideArea + 2\*frontArea + 2\* baseArea

doSomething :: [Int] -> [Int] -> Int

doSomething xs ys = if x < y then

let (\_:sx:\_) = xs; (\_:sy:\_) = ys

in sx + sy

else

x + y

where

(x:\_) = xs; (y:\_) = ys

Element in der Mitte einer Liste einfügen:

let (ys,zs) = splitAt n xs in  ys ++ [new\_element] ++ zs

Implementierung nach Signatur

f :: (a,b) -> (b -> c) -> (a -> c -> d) -> d

f1 f2

f (a,b) f1 f2 = f2 a (f1 b)

f2 a c

f5 :: a -> (a -> b) -> b

f5 x f = f x

Aufzählungstyp / Pattern match:

data Op = Add | Sub

calc :: Op -> Int -> Int -> Int

calc Add a b = a + b

calc Sub a b = a - b

Let:

letsNest :: Int -> Int

letsNest x = let

c = let b = 6

in (x + b)

in

let d = 4

in c + d

letsNest 3 ~> 13

letsShadow :: Int -> Int

letsShadow x = let a = 5

in

let a = 6

in a + x

letsShadow 3 ~> 9

Rekursionsfunktionen:

countDown :: Int -> [Int]

countDown 0 = [0]

countDown a = a : countDown (a-1)

countUp :: Int -> [Int]

countUp 0 = [0]

countUp a = countUp (a-1) ++ [a]

countDownUp :: Int -> [Int]

countDownUp 0 = [0]

countDownUp a = a : countDownUp(a-1) ++ [a]

len :: [a] -> Int

len [] = 0

len (i:is) = 1 + len is

allTrue :: [Bool] -> Bool

allTrue [] = True

allTrue (i:is) = i && allTrue is

sublist :: Int -> Int -> [a] -> [a]

sublist \_ 0 \_ = []

sublist \_ \_ [] = []

sublist 0 b (x:xs) = x : sublist 0 (b-1) xs

sublist a b (\_:xs) = sublist (a-1) b xs

(+++) :: [a] -> [a] -> [a]

(+++) [] b = b

(+++) (a:as) b = a:(as +++ b)

take' :: Int -> [a] -> [a]

take' \_ [] = []

take' 0 a = []

take' n (a:as) = a:(take' (n-1) as)

zip' :: [a] -> [b] -> [(a,b)]

zip' [] \_ = []

zip' \_ [] = []

zip' (a:as) (b:bs) = (a,b):(zip' as bs)

elem' :: Eq a => a -> [a] -> Bool

elem' \_ [] = False

elem' a (b:bs) = if a == b

then True

else elem' a bs

eqList :: Eq a => [a] -> [a] -> Bool //Vergl 2Lists

eqList [] [] = True

eqList \_ [] = False

eqList [] \_ = False

eqList (a:as) (b:bs) = if a == b

then eqList as bs

else False

(|>) :: (a -> b) -> (b -> c) -> (a -> c)

(|>) f1 f2 = f2.f1

(|>) :: (a -> b) -> (b -> c) -> a -> c

f |> f = \a -> g (f a)

f |> g = g.f

maxl :: (Ord a) => [a] -> a

maxl [] = error "maximum of empty list"

maxl (a:[]) = a

maxl (a:b:cs) = maxl ((max' a b):cs)

quicksort' :: (Ord a) => [a] -> [a]

quicksort' [] = []

quicksort' (x:xs) = quicksort' small ++ (x : quicksort' large)

where small = [y | y <- xs, y <= x]

large = [y | y <- xs, y > x]

alternate :: [a] -> [a] -> [a]

alternate [] [] = []

alternate [] b = b

alternate a [] = a

alternate (x:ys) (b:cs) = [x,b] ++ alternate ys cs

-- entfernt alle doppelten

diy :: Eq a => [a] -> [a]

diy [] = []

diy (x:xs) = x : diy (diy' x (xs))

oder

diy' a (y:xs)

| a == y = diy' a xs

| otherwise = y : diy' a xs

main = do

putStrLn “Program started”

loop

loop :: IO ()

loop = do

putStrLn "Text or ':quit'”

line <- getLine

if line == ":quit"

then return ()

else **do**

print (evalLine line)

loop

instance **Show** JSON where

show JNull = "null"

show (JBool b) = map toLower (show b)

show (JNum n) = show n

show (JStr s) = "\"" ++ s ++ "\""

show (JSeq js) = "[" ++ intercalate ", " (map show js) ++ "]"

show (JObj bs) = "{"++intercalate "," (map binding bs) ++ "}"

where binding (key, json) =

"\"" ++ key ++ "\": " ++ show json

Type Classes: an interface

class Movable a where

move :: Vector -> a -> a

instance Movable Figure where

move v f = moveFigure v f

instance (Movable a) => Movable [a] where

move v as = map (move v) as

Own Impl. of Ord instead Deriving Ord

instance **Ord** Point where

(<=) (XY x1 y1) (XY x2 y2)

| x1 == x2 = (y1 <= y2)

| otherwise = (x1 <= x2)

Recursive Data Types

(:) heisst Cons und hängt ein Element vorne an eine Liste. Der Datentyp Liste ist rekursiv definiert und entspricht strukturell:

data List a = Cons a (List a) | Nil

listSize :: List a -> Int

listSize Empty = 0

listSize (Cons \_ t) = 1 + listSize t

Modules

import Data.List (nub, sort)

:m + Data.List Data.Map Data.Set

not import to overwrite:

import Data.List hiding (nub)

Alles exportieren:

module Tasks.Console where

nur bestimmte Funktionen:

module Geometry

(circleArea

,circlePerimeter

,squareArea

,squarePerimeter

) where

IO is actions happening in sequence

IO a is an IO action of type a, it will return an a

getLine :: IO String

putStrLn :: String -> IO ()

return :: a -> IO a

getArgs :: IO [String] (aus System.Environment)

print = putStrLn . show

Die Type-Klasse Monad setzt den Binding Operator (>>==) und die return Operation voraus.

Der Binding Operator nimmt eine Monadic value, entpackt den Wert, führt darauf eine Funktion aus und packt das Resultat wieder in den selben Monad als Rückgabewert:

(>>=) :: m a -> (a -> m b) -> m b

(>>=) :: IO a -> (a -> IO b) -> IO b

m ist der Monad, m wrapt a, sodass zb. IO a, oder Maybe a oder eine Liste von a [a] herauskommt. (IO, Maybe und Listen sind Monads). m a ist dann eine monadic value, genauso wie m b

return :: Monad m => a -> m a

return ist der Wrapper, welcher den Wert in eine monadic value packt. Jede Impure function muss einen IO Typ zurückgeben. Wenn eine Funktion nun zb. eine Zahl einlesen, bearbeiten und zurückgeben soll, macht return a aus dem Integer-Rückgabewert einen IO Int.

Wird die Zahl direkt an der Konsole ausgegeben mit putStr, hat dieses bereits einen IO Rückgabetyp und daran müssen wir nichts ergänzen.

Im Gegensatz dazu ist (>>) nur eine Möglichkeit die Aktionen aneinander zu reihen ohne die Ergebnisse zu beachten:

(>>) :: IO a -> IO b -> IO b

import System.Environment

import Data.List

main = do args <- getArgs

putStrLn $ intercalate "\n" args

putStr :: String -> IO ()

putStr [] = return ()

putStr (x:xs) = do

    putChar x

    putStr xs

Unit Tests

data Expr = Const Int

| Add Expr Expr

| Mul Expr Expr

deriving (Show, Eq)

test1 :: Eq b => (a -> b) -> [(a,b)] -> Bool

test1 f [] = True

test1 f (x:xs) = (f (fst x) == snd x) && test1 f xs

testdata = [

((Const 1),1),

((Add (Const 2) (Const 3)),5),

((Add (Const 0) (Const 3)),3),

((Add (Const 3) (Const 0)),3),

((Mul (Const 2) (Const 3)),6),

((Add (Mul (Const 3) (Const 4)) (Add (Const 5) (Const 2))),19)

]

evalTest = test1 eval testdata

evalTest2 = test1 (eval.simpl) testdata

HigherOrderFunctions

b is das Resultat/Zwischenresultat und braucht eine Initial-Value

bis [a] leer ist wird die Funktion auf jedem Elem. angewendet

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr \_ z [] = z

foldr f z (x:xs) = f x (foldr f z xs)

foldr f z [1,2,3] = f 1 (f 2 (f 3 z))

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

foldl f z [] = z

foldl f z (x:xs) = foldl f (f z x) xs

foldl f z [1,2,3] = f (f (f z 1) 2) 3

Usage:

countChar :: Char -> String -> Int

countChar c xs = foldr step 0 xs

where step :: Char -> Int -> Int

step x cnt | x == c = cnt + 1

| otherwise = cnt

fs = [(+1), (\*3), (/2)]

ns = [1,2,3]

foldr (\(a,b) (as, bs) -> (a:as, b:bs)) ([],[]) [(1,'a'), (2,'b')] ~> ([1,2], "ab")

Positionsbhängiges Anwenden von fs auf ns:

zipWith ($) fs ns ~> [2.0,6.0,1.5]

IO Module

module Tasks.Console where

import Tasks.Core

import System.IO

import System.Directory

taskFile :: FilePath

taskFile = "TasksDB.txt"

main :: IO ()

main = do

hSetBuffering stdin LineBuffering

hSetBuffering stdout NoBuffering

putStrLn "FProg Task List Manager"

loop

loop :: IO ()

loop = do

putStr "> "

line <- getLine

let input = words line

if null input

then loop

else

let (cmd:args) = input

in do cont <- dispatch cmd (unwords args)

if cont then loop else return ()

dispatch :: String -> String -> IO Bool

dispatch "help" \_ = listCommands >> return True

dispatch "list" \_ = listTasksAction taskFile >> return True

dispatch "add" t = addTaskAction taskFile t >> return True

dispatch "done" nr = markDoneAction taskFile (read nr) >> return True

dispatch "remove" nr = removeTaskAction taskFile (read nr) >> return True

dispatch "quit" \_ = return False

dispatch \_ \_ = return True

listCommands :: IO()

listCommands = do

putStrLn ("help, list, add Tasktitle, done nr, remove nr")

listTasksAction :: FilePath -> IO ()

listTasksAction file = do

tasks <- readTasks file

putStrLn (concat (showTasks tasks))

Aus Tasks.Core:

type Task = (Bool,String)

type TaskList = [Task]

readTasks :: FilePath -> IO TaskList

readTasks file = do

exists <- doesFileExist file

if not exists then return []

else do content <- readFile file

let tasks = if null content then [] else (read content)

length tasks `seq` return tasks

numberTasks :: TaskList -> [(Int,Task)]

numberTasks tasks = zip [0..] tasks

showTasks :: TaskList -> [String]

showTasks [] = ["Horray! You've got nothing to do."]

showTasks ts = map showTask (numberTasks ts)

showTask :: (Int, Task) -> String

showTask (nr,(done,desc)) = nrStr nr ++ " " ++ doneStr done ++ " " ++ desc ++ "\n"

where doneStr True = "[X]"

doneStr False = "[ ]"

nrStr nr = show nr ++ "."

type Person = (String, String, Int)

name :: Person -> String

name (n, \_, \_) = n

teacher :: Person

teacher = ("Daniel Kröni", "056217", 38)

data Student = Student { email :: String, grade :: Float }

let stu = Student "hui@fhnw.ch" 5.6

Typename Value Field

data Shape = Circle Float

| Rectangle Float Float

deriving (Show)

circumference :: Shape -> Float

circumference (Circle r) = 2 \* pi \* r

circumference (Rectangle a b) = 2\*a + 2\*b

:t Rectangle

Rectangle :: Float -> Float -> Shape

Own Types:

data Size = Small | Large deriving Show

data Currency = USD|EUR

toString :: Currency -> String

toString USD = "$"

toString EUR = "€"

Usage:

type Cash = (Int,Currency)

cash1 :: Cash

cash1 = (100,EUR)

Maybe

data Maybe x = Just x

| Nothing

return x = Just x

Nothing >>= f = Nothing

Just x >>= f = f x

safeHead :: [a] -> Maybe a

safeHead [] = Nothing

safeHead (a:as) = Just a

Record-Syntax

=> Named components:

data PersonType = Person { firstName :: String,

lastName :: String,

email :: String,

yob :: Int

} deriving (Show)

> firstName (Person "Haskell" "Curry" "unknown" 1900)

> "Haskell"

Basic Classes:

**Eq –** equality types

– methods: (==) , (/=)

**Ord –** ordered types

– methods: (<) , (<=) , (>) , (>=) , min, max

**Show –** showable types

– method show :: a -> String

**Num –** numeric types

– methods: (+) , (-) , (\*) , negate, abs,signum

Kompositionen

toLower :: Char -> Char

toLower x = x

normalize :: String -> [String]

normalize = (filter (\a -> (length a > 2))) . words . (map toLower)

spam = ["viagra", "buy", "spam"]

rateWord :: String -> Int

rateWord a | elem a spam = -10

| otherwise = 2

rateWords :: [String] -> Int

rateWords = sum . map rateWord

isSpam :: String -> Bool

isSpam x = rateWords (normalize x) < 0