ПМИ 111 группа

Вторцев К.Д.

**Лабораторная работа 1**

--Найдем максимум из 3-х чисел

max3 :: Int -> Int -> Int -> Int

max3 x y z = max ( max x y ) z

--Найдем минимум из 3-х чисел

min3 :: Int -> Int -> Int -> Int

min3 x y z = min ( min x y ) z

--Функция sort2

sort2 :: Int -> Int -> ( Int, Int )

sort2 x y = (min x y, max x y)

--Функция bothTrue1

bothTrue1 :: Bool -> Bool -> Bool

bothTrue1 a b = min a b

--Функция bothTrue2

bothTrue2 :: Bool -> Bool -> Bool

bothTrue2 True True = True

bothTrue2 True False = False

bothTrue2 False True = False

bothTrue2 False False = False

--Функция bothTrue3

bothTrue3 :: Bool -> Bool -> Bool

bothTrue3 True x = x

bothTrue3 \_ \_ = False

--Функция solve2

solve2 :: Double -> Double -> (Bool, Double)

solve2 a b = if a == 0 then ( False, 0.0) else ( True, ((-b)/a))

--Функция solve22

solve22 :: Double -> Double -> (Bool, Double)

solve22 0 \_ = (False, 0.0)

solve22 a b = (True, ((-b)/a))

--Функция isParallel

isParallel :: (Double, Double) -> (Double, Double) -> (Double, Double) -> (Double, Double) -> Bool

isParallel (a, b) (c, d) (e, f) (g, h) = (c - a) / (g - e) == (d - b) / (h - f)

--функция isIncluded

isIncluded :: (Double, Double) -> Double -> (Double, Double) -> Double -> Bool

isIncluded (x', y') a (x'', y'') b = sqrt((x''-x')^2 + (y''-y')^2) + b <= a

--функция isRectangular

isRectangular :: (Double, Double) -> (Double, Double) -> (Double, Double) -> Bool

isRectangular (x', y') (x'', y'') (x''', y''') = (x'' - x') \* (x''' - x') + (y'' - y') \* (y''' - y') == 0 || (x' - x'') \* (x''' - x'') + (y' - y'') \* (y''' - y'') == 0 || (x' - x''')\*(x'' - x''') + (y' - y''') \* (y'' - y''') == 0

--функция isTriangle

isTriangle :: Double -> Double -> Double -> Bool

isTriangle x y z = x^2 + y^2 == z^2 || x^2 + z^2 == y^2 || y^2 + z^2 == x^2

--функция isSorted

isSorted :: Double -> Double -> Double -> Bool

isSorted a b c = a >= b && b >= c || a <= b && a <= c

fnat :: Int->[Int]

fnat 0 = []

fnat n = if n<0 then fnat(n+1)++[n] else fnat(n-1)++[n]

fnat1 :: Int -> [Int]

fnat1 0 = []

fnat1 n = fnat(n-1)++[n]

**Лабораторная работа 2**

{-№1

Функция, принимающая на вход целое число n и возвращающее список, содержащий n

элементов, упорядоченных по возрастанию

1)Список натуральных чисел

2)Список нечетных натуральныз чисел

3) Список квадратов натуральных чисел

4) Список факториалов

5) Список степени двойки

7) Список треугльных чисел

8) Список пирамидальных чисел-}

f1 :: Integer -> [Integer]

f1 0 = []

f1 n = f1(n - 1) ++ (n : [])

f2 :: Integer -> [Integer]

f2 0 = []

f2 n = f2(n - 1) ++ (2 \* n - 1 : [])

f3 :: Integer -> [Integer]

f3 0 = []

f3 n = f3(n - 1) ++ (2 \* n : [])

f4 :: Integer -> [Integer]

f4 0 = []

f4 n = f4(n - 1) ++ (n \* n : [])

f5 :: Integer -> [Integer]

factorial 0 = 1

factorial n = n \* factorial(n - 1)

f5 0 = []

f5 n = f5(n - 1) ++ [factorial(n)]

f6 :: Integer -> [Integer]

f6 0 = []

f6 n = f6(n - 1) ++ (2^n : [])

f7 :: Integer -> [Integer]

t 1 = 1

t(n) = n + t(n - 1)

f7 0 = []

f7 n = f7(n - 1) ++ [t(n)]

f8 :: Integer -> [Integer]

p1 = 1

p(n) = t(n) + p(n-1)

f8 0 = []

f8 n = f8(n -1) ++ [p(n)]

{-№2 Определить следующие функции-}

-- Функция, принимающая на входе список вещественных чисел и вычисляющую их арифметическое среднее.

quantity :: [Double] -> Double

quantity [] = 0.0

quantity [x] = 1.0

quantity (x:xs) = 1.0 + (quantity xs)

sum\_ :: [Double] -> Double

sum\_ [] = 0.0

sum\_ [x] = 1.0

sum\_ (x:xs) = ((x) + (sum\_ xs))

f9 n = (sum\_ n) / (quantity n)

-- Функция 2 вычленения n-го элемента из заданного списка

f10 :: Integer -> [Integer] -> [Integer]

f10 n (x:xs) = if n == 1 then xs else x: f10 (n-1) xs

{-Функция сложения элементов двух списков. Возвращает список, составленый из сумм элементов

списков-параметров Учесть, что переданные списки могут быть разной длины -}

add2List :: [Integer] -> [Integer] -> [Integer]

add2List [][] = []

add2List [](n1:nx) =(n1:nx) --учёт длины

add2List (l1:lx)[] = (l1:lx)

add2List (l1:lx) (n1:nx) = (l1+n1): (add2List lx nx)

{-Функция перестановки местами соседних четных и нечетных элеметов в заданном списке

[1,2,3,...,n]-}

f11 :: [Integer] -> [Integer]

f11 [] = []

f11 [x] = [x]

f11 (x:y:xs) = y:x:(f11 xs)

--Функция twopow n

twopow :: Integer -> Integer

twopow n | (n == 1) = 2

| (even n) = w \* w

| otherwise = (w \* w) + (w \* w) where w = twopow (n `div` 2)

{-Функция removeOdd, которая удаляет из заданного списка целые числа все нечетные числа.

Например: removeOdd [1,4,5,6,10] должен возвращать [4,10]-}

removeOdd :: [Integer] -> [Integer]

removeOdd [] = []

removeOdd [x] = if odd x == True then [] else [x]

removeOdd (x:xs) = if odd x == True then removeOdd xs else x:removeOdd xs

{-Функция removeEmpty, которая удаляет пустые строки из за-

данного списка строк.

Например: removeEmpty ["", "Hello", "", "", "World!"] возвращает ["Hello","World!"].-}

removeEmpty :: [String]->[String]

removeEmpty [] = []

removeEmpty [x] = if x == "" then [] else [x]

removeEmpty (x:xs) = if x == "" then removeEmpty xs else x:removeEmpty xs

-- Функция countTrue :: [Bool] -> Integer, возвращающая количество элементов списка, равных True.

countTrue :: [Bool] -> Integer

countTrue [] = 0

countTrue [x] = if x == True then 1 else 0

countTrue (x:xs) = if x == True then 1 + (countTrue xs) else countTrue xs

--makePositive меняет знаки - на +

makePositive :: [Integer]->[Integer]

makePositive [x] = if x < 0 then [(-1)\*x] else [x]

makePositive (x:xs) = if x < 0 then ((-1)\*x): (makePositive xs) else x: (makePositive xs)

{-Функция delete :: Char -> String -> String, которая принимает на вход стоку, в которой

удалены все взожения символа.

Пример: delete 'l' "Hello world!" должно возвращать "Heo word!"-}

delete :: Char -> String -> String

delete c [] = ""

delete c cs = if c == (head cs) then delete c (tail cs) else (head cs): (delete c (tail cs))

-- Заменить один символ в строке на другой

substitute :: Char->Char->String->String

substitute c1 c2 [] = ""

substitute c1 c2 str = if c1 == (head str) then c2:substitute c1 c2 (tail str) else (head str): (substitute c1 c2 (tail str))

--Я не увидел порядок выполнения лаюораторной работы, поэтому сделал все варианты. Мое задание 6,11, т.к. вар 6

**Лабораторная работа 3**

{-задание 6

В некотором языке программирования существуют следующие типы данных:

- Простые типы: целые, вещественные и строки

- Сложные типы: структуры. Структура имеет название и состоит из нескольких

полей каждое из которых в свою очередь имеет название и простой тип.

База данных идентификаторов программы представляет собой список пар,

состоящих из имени идентификатора и его типа. Разработайте тип данных,

представляющий описанную информацию. Определите след. ф-ции:

1) isStructured, проверяющая, что её аргумент является сложным типом.

2) getType, по заданному имени и списку идентификаторов (базе данных)

возвращающая тип идентификатора с указанным именем (помните о том,

что такого идентификатора в базе может и не оказаться).

3) getFields, по заданному имени возвращающая список полей идентификатора,

если он имеет тип структуры.

4) getByType, возвращающая список идентификаторов указанного типа

из базы данных.

5) getByTypes, аналогичная предыдущей, но принимающей вместо одного

типа список типов (с помощью этой ф-ции можно получить, например,

список всех идентификаторов с числовым типом).-}

data MyType = Null |

MyInt |

MyFloat |

MyString |

MyStruct [(String,MyType)] deriving (Show,Eq)

db = [("x",MyInt),

("y",MyFloat),

("z",MyString),

("u",MyInt),

("s",MyStruct [("s1",MyInt),("s2",MyFloat)])]

isStructured :: MyType -> Bool

isStructured (MyStruct b) = True

isStructured (\_) = False

getType :: String -> [(String,MyType)] -> MyType

getType \_ [] = Null

getType s (h:t) = if fst(h) == s then snd(h) else getType s t

getFieldsFromStruct :: MyType -> [(String,MyType)]

getFieldsFromStruct (MyStruct a) = a

getFieldsFromStruct \_ = []

getFields :: String -> [(String,MyType)] -> [(String,MyType)]

getFields \_ [] = []

getFields s (h:t) = if fst(h) == s && isStructured (snd h) then

getFieldsFromStruct (snd h)

else

getFields s t

getByType :: MyType -> [(String,MyType)] -> [String]

getByType \_ [] = []

getByType a (h:t) = if a == (snd h) then

(fst h) : getByType a t

else

getByType a t

getByTypes :: [MyType] -> [(String,MyType)] -> [String]

getByTypes [] \_ = []

getByTypes \_ [] = []

getByTypes (h:t) db = (getByType h db) ++ getByTypes t db

**Лабораторная работа 4**

{-

 Работа с типом Expr, реализуйте следующее функции

      1) Определите  корректную функцию diff, которая принимает в качестве дополнительного аргумента имя переменной, по которой необходимо осуществлять дифференцирование

      2) Определите функцию simplify, которая упрощает выражения типа Expr, применяя очевидные правила вида (х+0 = 0 + х = х) и т.п.

      3) Определите функцию toString, преобразующую выражение типа Expr в строку.

      4) Определите функцию eval, которая принимает два параметра: выражение типа Expr и список пар типа (String,Integer), задающий соответствие имен переменных и их значений

-}

-- Exp

data Expr = Const Int

| Var String

| Add Expr Expr

| Mult Expr Expr deriving (Eq, Show)

-- функция diff

diff :: Expr -> String -> Expr

diff (Const \_) \_ = Const 0

diff (Var x) c = if x == c then Const 1 else Const 0

diff (Add x y) c = Add (diff x c) (diff y c)

diff (Mult x y) c = Add (Mult (diff x c) y) (Mult x (diff y c))

--функция simplify

simplify :: Expr -> Expr

simplify (Var x) = Var x

simplify (Const a) = Const a

simplify (Add x (Const 0)) = (simplify x)

simplify (Add (Const 0) y) = (simplify y)

simplify (Add (Const x) (Const y)) = Const (x+y)

simplify (Add x y) = (Add (simplify x) (simplify y))

simplify (Mult (Const a) (Const b)) = Const (a\*b)

simplify (Mult (Var x) (Var y) )= (Mult (simplify(Var x)) (simplify (Var y)))

simplify (Mult x (Const 1)) = (simplify x)

simplify (Mult (Const 1) y) = (simplify y)

simplify (Mult \_(Const 0)) = Const 0

simplify (Mult (Const 0 )\_) = Const 0

simplify (Mult x y) = (Mult (simplify x) (simplify y))

-- intToString

intToString :: Int -> String

intToString a | a == 1 = "1" |a==2 = "2"| a==3 = "3"| a==4 = "4"|a==5="5"|a==6="6"|a==7="7"|a==8="8"|a==9="9"| a==0="0"

|otherwise =

let ost = mod a 10

ch = div a 10

in (intToString ch) ++(intToString ost)

toString :: Expr -> String

toString (Const a) = intToString a

toString (Var x) = x

toString (Add x y) = "(" ++ ((toString x) ++ "+" ++ (toString y)) ++ ")"

toString (Mult x y) = ((toString x) ++ "\*" ++ (toString y))

--choosePar

choosePar :: String -> [(String,Int)] -> Int

choosePar \_ [] = error "Error argument"

choosePar a (x:xs) = if a == (fst x) then snd x else (choosePar a xs)

--eval

eval :: Expr -> [(String,Int)] -> Int

eval (Var x) xs = (choosePar x xs)

eval (Const x) \_ = x

eval (Add x y) xs = (eval x xs) + (eval y xs)

eval (Mult x y) xs = (eval x xs) \* (eval y xs)

**Лабораторная работа 5**

{-

Определить следующие функции с использованием функций высшего порядка

1) Функция midAr вычисления арифметического среднего элементов списка выщественных чисел с использованием функции foldr

2) Функция skMult, вычисляющая скаляраное произведение двух список (используя функции foldr и zipWith)

3) Функция countEven, возвращающая количество четных элементов в списке

4) Функция quicksort, осуществляющая быструю сортировку списка

5) Определенная в пред пункте функция quicksort сортирует список в пордке возрастания. Обобщить ее: пусть она принимает еще один аргумент - функцрю сравнения типа a-> a -> Bool и сортирует список в соответсвие с ней

-}

--midAr

midAr :: [Double] -> Double

midAr (x) = (foldr (+) 0 x) / (fromIntegral (length x))

--zipWith

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

zipWiths z (a:as) (b:bs) = z a b : zipWiths z as bs

zipWiths \_ \_ \_ = []

--skMult

skMult :: [Integer] -> [Integer] -> Integer

skMult xs ys = (foldr (+) 0 (zipWiths (\*) xs ys))

--countEven

countEven :: [Int] -> Int

countEven xs = counter even xs

counter :: (Int -> Bool) -> [Int] -> Int

counter f xs = foldl (\ acc y -> if (f y) then acc+1 else acc) 0 xs

--quickSorts

quickSorts :: [Int] -> [Int]

quickSorts [] = []

quickSorts (x:xs)= quickSorts(filter (<x) xs) ++ [x] ++ quickSorts(filter (>=x) xs)

--quickSorts2

quickSorts2 :: (a -> a -> Bool) -> [a] -> [a]

quickSorts2 f [] = []

quickSorts2 f (x:xs) = quickSorts2 f (filter (f x) xs) ++ [x] ++ (quickSorts2 f (filter (\ y -> not $ f x y) xs))