

**Functional Programming**

LAB #2

Topic: Laboratory work Nr 2

Student’s name: Andrey

Student’s surname: Perunov

St. code:

81049

Lecturer: Aleksejs Vesjolijs

**RIGA**

Contents

[Git repository 3](#_Toc163209740)

[Tasks 3](#_Toc163209741)

[Task 1 3](#_Toc163209742)

[Task 2. 4](#_Toc163209743)

[Task 3. 6](#_Toc163209744)

[Task 4. 6](#_Toc163209745)

# Git repository

<https://github.com/AndreyPerunov/FP-Lab2>

# Tasks

## Task 1

Create a program in HASKELL. Program should take user input and print output.

**Variant 1.**

Implement a function. Requirement: Function, for some k, m, and n, determines the number of solutions of the equation kx² + m(x –n) = 0, using case distinction.

**Answer the questions.**

What is the type of the following functions?

tail, sqrt, pi, exp, (ˆ), (/=) and function you created?

How can you query the interpreter for the type of an expression and how can you explicitly specify the types of functions in your program?

**Solution:**

What is the type of the following functions?

* tail :: [a] -> [a]
* sqrt :: Floating a => a -> a
* pi :: Floating a => a
* exp :: Floating a => a -> a
* (ˆ) :: (Num a, Integral b) => a -> b -> a
* (/=) :: Eq a => a -> a -> Bool
* solutions :: Int -> Int -> Int -> Int
* discriminant :: Int -> Int -> Int -> Int

How can you query the interpreter for the type of an expression and how can you explicitly specify the types of functions in your program?

* running ghci and typing: ":t \*your expression here\*"
* \*function\_name\* :: \*input type\* -> \*return type\*
* \*input type\* can be repeated for multiple arguments

*-- Determines the number of solutions of the equation kx² + m(x –n) = 0*

solutions :: Int -> Int -> Int -> Int

solutions k m n *-- kx² + m(x – n) = 0*

*-- kx² + m(x – n) = 0 == kx² + mx - mn) = 0*

*-- a == k, b == m, c == -mn*

  | discriminant k m (m\*(-n)) > 0 = 2 *-- D > 0 - two real roots*

  | discriminant k m (m\*(-n)) == 0 = 1 *-- D = 0 - one real root*

  | discriminant k m (m\*(-n)) < 0 = 0 *-- D < 0 - no real roots*

  | otherwise = -1 *-- error*

*-- ax² + bx + c = 0*

*-- D = b² - 4ac*

discriminant :: Int -> Int -> Int -> Int

discriminant a b c = b^2 - 4 \* a \* c

main :: IO ()

main = do

  putStrLn "kx^2 + m(x - n) = 0"

  putStrLn "Enter k: "

  kString <- getLine

  let k = read kString :: Int

  putStrLn "Enter m: "

  mString <- getLine

  let m = read mString :: Int

  putStrLn "Enter n: "

  nString <- getLine

  let n = read nString :: Int

  putStrLn "Number of solutions: "

  putStrLn (show (solutions k m n))

**Output:**

A screen shot of a computer

Description automatically generated

## Task 2.

**Create a program in HASKELL. Program should take user input and print output.**

**Variant 1.**

**Subtask 1** (10 pts). Define a function that determines the “run-length encoding” of a list: [1, 2, 2, 3, 2, 4] is mapped to [(1, 1),(2, 2),(1, 3),(1, 2),(1, 4)]. That is, the list is mapped to a list of pairs whose first element says how many times the second component of the pair appears in adjacent positions in the list.

**Subtask 2** (15 pts). Define a function that groups successive duplicate elements in a list into sublists: [1, 2, 2, 3, 2, 4] is mapped to [[1], [2, 2], [3], [2], [4]].

Note: Program should iterate through the same user input. Output should be meaningful and clear to which function it relates.

**Solution:**

*-- subtask 1*

encode :: [Int] -> [(Int, Int)]

encode [] = []

encode (head:tail) = encode' tail head 1

  where

    encode' :: [Int] -> Int -> Int -> [(Int, Int)]

*-- if the list is empty, return the last element and its count*

    encode' [] h count = [(count, h)]

    encode' (head:tail) h count

*-- if the next element is the same as the current one*

*-- start counting*

      | head == h = encode' tail h (count + 1)

*-- if the next element is different, add the current count and element to the list*

*-- start counting next number*

      | otherwise = (count, h) : encode' tail head 1

*-- subtask 2*

*-- switched back to (x:xs) instead of (head:tail) because of head function*

group :: [Int] -> [[Int]]

group [] = []

group (x:xs) = group' xs [x]

  where

    group' :: [Int] -> [Int] -> [[Int]]

*-- if the list is empty, return the last element*

    group' [] h = [h]

    group' (x:xs) h

*-- if the next element is the same as the current one*

*-- add it to the current sublist*

      | x == head h = group' xs (x:h)

*-- if the next element is different, add the current sublist to the list*

*-- start a new sublist with the next element*

      | otherwise = h : group' xs [x]

main :: IO ()

main = do

  putStrLn "Enter a list of numbers: "

  input <- getLine

  let strings = words input

  let ints = [read s :: Int | s <- strings]

  let encoded = encode ints

  putStrLn "Subtask 1: "

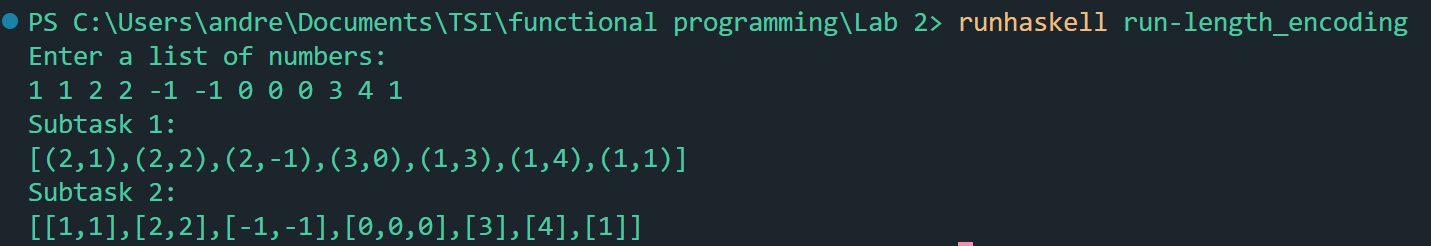
  putStrLn (show encoded)

  let grouped = group ints

  putStrLn "Subtask 2: "

  putStrLn (show grouped)

**Output:**



## Task 3.

**Answer the questions below.**

**Variant 1.**

Given the following definitions:

thrice a = [a, a, a]

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

What does the following expression evaluate to?

map thrice (sums [0 .. 4])

**Answer:**

[[0,0,0],[1,1,1],[3,3,3],[6,6,6],[10,10,10]]

**Explanation:**

1. sums [0 .. 4]
2. 0 : sums [0 + 1, 2, 3, 4]
3. 0 : 1 : sums [1 + 2, 3, 4]
4. 0 : 1 : 3 : sums [3 + 3, 4]
5. 0 : 1 : 3 : 6 : sums [6 + 4]
6. 0 : 1 : 3 : 6 : 10 : sums []
7. 0 : 1 : 3 : 6 : 10 : []
8. [0, 1, 3, 6, 10]
9. map thrice [0, 1, 3, 6, 10]
10. [thrice 0, thrice 1, thrice 3, thrice 6, thrice 10]
11. [[0, 0, 0], [1, 1, 1], [3, 3, 3], [6, 6, 6], [10, 10, 10]]

## Task 4.

**Answer the questions below**

**Variant 1-2.**

Consider the following datatype definition for binary trees that we shall want to use to implement binary search trees:

data Tree a = Branch a (Tree a) (Tree a) | Leaf

Write a function isSearchTree :: Tree a −> Bool that verifies that its argument is a binary search tree.

Then test the property that given a binary search tree t, inserting a value into thetree results in yet another binary search tree. The code for inserting a new value into the tree is:

insertTree :: Ord a => a −> Tree a −> Tree a insertTree e Leaf = Branch e Leaf Leaf insertTree e (Branch x li re)

| e <= x = Branch x (insertTree e li) re

| e > x = Branch x li (insertTree e re)

Experiment with mutating the implementation of insertTree to find out whether your property can in fact discover that the mutated implementation no longer maps binary search trees to binary search trees.

**Solution:**

*-- Leaf == NIL == NULL*

data Tree a = Branch a (Tree a) (Tree a) | Leaf

isSearchTree :: Ord a => Tree a -> Bool

*-- if it is NIL (Leaf) then it's binary search tree*

isSearchTree Leaf = True

isSearchTree (Branch node leftSubTree rightSubTree) = isLeftSmaller node leftSubTree && isRightLarger node rightSubTree && isSearchTree leftSubTree && isSearchTree rightSubTree

  where

*-- checking that left node is smaller then parent*

    isLeftSmaller parentNode Leaf = True

    isLeftSmaller parentNode (Branch node leftSubTree rightSubTree) = parentNode >= node

*-- checking that right node is larger then parent*

    isRightLarger parentNode Leaf = True

    isRightLarger parentNode (Branch node leftSubTree rightSubTree) = parentNode < node

insertTree :: Ord a => a -> Tree a -> Tree a

insertTree e Leaf = Branch e Leaf Leaf

insertTree e (Branch x li re)

  | e <= x = Branch x (insertTree e li) re

  | e > x = Branch x li (insertTree e re)

main :: IO ()

main = do

  let t = Branch 5 (Branch 3 Leaf Leaf) (Branch 7 Leaf Leaf)

  print $ isSearchTree t

  print $ isSearchTree $ insertTree 8 t

  print $ isSearchTree $ insertTree 2 t

  print $ isSearchTree $ insertTree 4 t

**Output:**

