Започнат на	Sunday, 16 January 2022, 11:30
Състояние	Завършен
Приключен на	Sunday, 16 January 2022, 12:24
Изминало време	54 мин. 40 сек.
Оценка	Още не е оценен

Въпрос **1**

Отговорен

От максимално 1,00

Как се конструира списък чрез техниката на определяне на неговия обхват (list comprehension) в езика Haskell? Обяснете общия случай и дайте поне два примера, в които се използват различни възможности на тази техника (включително напишете получените резултати).

За конструиране на списък чрез тази техника трябва първо задължително да имаме генератор на списъка. Общия синтакс изглежда по следния начин:

[x | x <- xs, cond...] като xs е генератор, а x е образец. Можем и да добавяме различни условия за x-овете на мястото на cond, като ги отделяме със запетаи. Можем също и да приложим някаква функция върху x-овете, така че да променим резултата в новия списък.

getEven :: [Int] -> [Int]

getEven xs = [x | x <- xs, even x]

Тази функция връща списък от всички четни числа в даден списък. Аналогична е с вградената функция от по-висок ред filter.

getEven [1, 5, 3, 2, 7, 10] -> връща списъка [2, 10]

addOnes :: [Int] -> [Int]

addOnes xs = [x + 1 | x <- xs]

Тази функция връща списък от числа, които са увеличени с единица в сравнение с подадения списък.

addOnes [2, 10, 40, 0, -5] -> връща [3, 11, 41, 1, -4]

Въпрос **2**

Отговорен

От максимално 1,00

Как се дефинират класове в Haskell? Опишете общия случай и дайте конкретен пример за дефиниция на клас.

За да дефинираме клас ние трябва да упоменем кои методи за дефинирани за този тип клас като ограничения

Например класът Eq

class Eq where

(==) :: a -> a -> Bool

Задължително трябва да имаме дефиниран метод == за да можем да сравняваме такива типове.

Типовете които могат да се сравняват са екземпляри на Eq и го наследяват

Въпрос **3**

Отговорен

От максимално 1,00

Дайте пример за дефиниция на функция на Haskell, в която се използва обща (а не примитивна) рекурсия върху списъци. Обяснете каква задача се решава с помощта на тази функция.

Пример за обща рекурсия е дефиницията на функцията, имплементираща алгоритъма quick sort.

В нея вземаме първия елемент на списък, правим го на списък от един елемент и го конкатенираме от двете му страни с два други списъка, като този отляво е списък от рекурсивното извикване на функцията, който съдържа елементите <= x, а този вдясно съответно - по-големи от x, като x е елемента, който разглеждаме .

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs) = (qSort [l | l <- xs, l <= x]) ++ [x] ++ (qSort [r | r <- xs, r > x])
```

Примерна оценка:

```
qSort [] = []
qSort [5, 2, 0, 15, 3, 8, 7] =
qSort [2, 0, 3] ++ [5] ++ qSort [15, 8, 7] =
qSort [0] ++ [2] ++ qSort [3] ++ [5] ++ qSort [8, 7] ++ [15] ++ qSort [] =
qSort [] ++ [0] ++ qSort [] ++ [2] ++ qSort [] ++ [3] ++ qSort [] ++ [5] ++ qSort [7] ++ [8] ++ [15] ++ qSort [] =
[] ++ [0] ++ [] ++ [2] ++ [] ++ [3] ++ [] ++ [5] ++ [] ++ [7] ++ [8] ++ [15] ++ [] =
[0, 2, 3, 5, 7, 8, 15]
```

Въпрос **4**

Отговорен

От максимално 1,00

Кои от следните конструкции са коректно дефинирани (валидни) списъци в Haskell? Обосновете отговорите си. Посочете типовете на валидните списъци.

(а) [(123,”Hello”),(”123”,”Hello”)]

(б) [[”123”,”Hello”],[”123”,”Hello”,”World”]]

(в) [[1,2],[3],[4,5,6]]

(г) [[],[[3]],[[2,1]]]

Валидните списъци и съответно типовете им са:

а) – невалиден списък, понеже типът трябва да е еднозначно дефиниран, а в случая имаме първи елемент на списъка от тип (Int, String) и втори елемнет от тип (String, String) което няма как да се случи

б) - [[[Char]]] или [[String]] понеже всеки от елементите на списъка е списък и той има елементи String тоест списък от списъци от String (списък от символи)

в) - [[Int]] имаме списък от списъци от цели числа

г) - [[[Int]]] същото като във в) но този път виждаме че втория и третия елемент имат един допълнителен вложен списък следователно имаме списък от списъци от списъци от числа, а първия елемент също е валиден, понеже празният списък изпълнява условието на типа на данните

Въпрос **5**

Отговорен

От максимално 1,00

Обяснете понятието „алгебричен тип“ в езика Haskell. Дайте пример за дефиниция на полиморфен алгебричен тип.

Алгебричен тип в Haskell е един вид структура от данни (както е в C++), в която можем да наредим други по-прости типове.

Например:

data Person Name Age

type Name = String

type Age = Int

е алгебричен тип, който съставлява човек с данни име и възраст

data Shape a = Circle a | Rectangle a a | Square a | Triangle a a a

Това вече е пример за полиморфичен тип Shape който можем да представим като различни фигури, които определяме чрез конструкторите след знака за =

Така Shape може да бъде Circle, Rectangle, Square или Triangle

Въпрос **6**

Отговорен

От максимално 1,00

Обяснете понятието „образец“ (pattern) в езика Haskell. Дайте примери за поне три типа образци и обяснете кога те се съпоставят успешно със съответните аргументи.

Примери за образци са следните:

- литерали - "something", 0, False от различни типове

- променливи от различни типове

- символът "_" известен като wildcard - съответства на произволен аргумент

- конструктори на типове

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Функцията fact използва 2 типа образци, единият е с литерал 0, другият е с променлива n

Проверяват се подред както са записани

```
data BTree = Nil | Node Int BTree BTree

isEvenNodeValue :: BTree -> Bool
isEvenNodeValue Nil = False
isEvenNodeValue (Node v _ _) = even v
```

В този случай имаме дефиниция на двоично дърво с конструктори и функция която получава възел като аргумент и проверява дали стойността му е четно или нечетно число. И в двата образци използваме конструктор Nil и Node (в този случай разбиваме променливата по този начин за да вземем стойността ѝ), като във втория използваме и wildcard за децата на дървото, понеже те не са ни нужни във функцията

Въпрос **7**

Отговорен

От максимално 1,00

Обяснете понятието „дефиниране на функция на функционално ниво“ в езика Haskell. Дайте поне два пример за дефиниция на функция на функционално ниво.

Това означава когато елиминираме променливите в дефиницията на функция

filterEvenMapAddOne :: [Int] -> [Int]

filterEvenMapAddOne xs = map (+1) $ filter even xs

Тук имаме нормална дефиниция с променлива, но забелязваме, че xs е най-отдясно, затова можем да го изпуснем и да променим леко синтакса:

filterEvenMapAddOne = map (+1) . filter even

$ заменяме с . и изпускаме xs

Друг пример:

concatAndReverse :: [String] -> String

concatAndReverse xs = reverse $ concat xs

Имаме списък от списъци от символи и искаме да ги обединим и да ги обърнем затова прилагаме същия метод и получаваме:

concatAndReverse = reverse . concat

Въпрос **8**

Отговорен

От максимално 1,00

Проследете стъпка по стъпка процеса на оценяване и напишете оценката на всеки от следващите изрази на езика Haskell:

(\x y z -> x y z) (\x y -> x*x+y*y) 3 4

sum (map (\f -> f 3) [\x->x, \x->x*x, \x->x*x*x])

(\x y z -> x y z) (\x y -> x*x+y*y) 3 4 =

Първият израз със скоби приема функция и две променливи и прилага функцият върху тях, тоест се изпълнява

x y z

x е функцията (\x y -> x*x + y*y)

y е числото 3

z е числото 4

тоест свеждаме задачата до:

(\x y -> x*x + y*y) 3 4 което е равно на 3*3 + 4*4 = 9 + 16 =25

Вторият израз прави следното:

sum (map (\f -> f 3) [\x->x, \x->x*x, \x->x*x*x])

map върху списъка от ламбда функции, като заменяме всеки елемент с прилагането на функцията върху числото 3 и ни дава [3, 9, 27] и след това прилагаме sum върху този списък и получаваме числото 39

◄ Второ контролно по ФП - теоретична част

Отиди на ...

Група 3: Оценки Д1-Д3 КЗ 1-2 ►