Laboratorul 2: Funcții

Exerciții

1. Să se scrie o funcție `poly2` care are patru argumente de tip `Double`, `a,b,c,x` și calculează `a*x^2+b*x+c`. Scrieți și signatura funcției (`poly :: ceva`).
2. Să se scrie o funcție `eeny` care întoarce "eeny" pentru input par și "meeny" pentru input impar. Hint: puteti folosi funcția `even` (puteți căuta pe https://hoogle.haskell.org/).

```haskell
eeny :: Integer -> String
eeny = undefined
```

3. Să se scrie o funcție `fizzbuzz` care întoarce "Fizz" pentru numerele divizibile cu 3, "Buzz" pentru numerele divizibile cu 5 și "FizzBuzz" pentru numerele divizibile cu ambele. Pentru orice alt număr se întoarce șirul vid. Pentru a calcula modulo a două numere puteți folosi funcția `mod`. Să se scrie această funcție în 2 moduri: folosind `if` și folosind gărzi (condiții).

```haskell
fizzbuzz :: Integer -> String
fizzbuzz = undefined
```

Recursivitate

Una dintre diferențele dintre programarea declarativă și cea imperativă este modalitatea de abordare a problemei iterării: în timp ce in programarea imperativă acesta este rezolvată prin bucle (`while`, `for`, ...), în programarea declarativă rezolvarea iterării se face prin conceptul de recursie.

Un avantaj al recursiei față de bucle este acela că usurează sarcina de scriere și verificare a corectitudinii programelor prin raționamente de tip inductiv: construiește rezultatul pe baza rezultatelor unor subprobleme mai simple (aceeași problemă, dar pe o dimensiune mai mică a datelor).

Un foarte simplu exemplu de recursie este acela al calculării unui element de index dat din secvența numerelor Fibonacci, definită recursiv de:

$$F_n = \begin{cases} n & \text{dacă } n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{dacă } n > 1 \end{cases}$$

Putem transcrie această definiție direct in Haskell:

```haskell
fibonacciCazuri :: Integer -> Integer
fibonacciCazuri n
    | n < 2     = n
    | otherwise = fibonacciCazuri (n - 1) + fibonacciCazuri (n - 2)
```

Alternativ, putem folosi o definiție în stil ecuațional (cu șabloane):

```haskell
fibonacciEcuational :: Integer -> Integer
fibonacciEcuational 0 = 0
fibonacciEcuational 1 = 1
```

```
fibonacciEcuational n =
    fibonacciEcuational (n - 1) + fibonacciEcuational (n - 2)
```

4. Numerele `tribonacci` sunt definite de ecuatția

$$T_n = \begin{cases} 1 & \text{dacă } n = 1 \\ 1 & \text{dacă } n = 2 \\ 2 & \text{dacă } n = 3 \\ T_{n-1} + T_{n-2} + T_{n-3} & \text{dacă } n > 3 \end{cases}$$

Să se implementeze functia `tribonacci` atât cu cazuri cât și ecuațional.

```
tribonacci :: Integer -> Integer
tribonacci = undefined
```

5. Să se scrie o funcție care calculează coeficienții binomiali, folosind recursivitate. Aceștia sunt determinați folosind urmatoarele ecuații.

```
B(n,k) = B(n-1,k) + B(n-1,k-1)

B(n,0) = 1

B(0,k) = 0
```

```
binomial :: Integer -> Integer -> Integer
binomial = undefined
```

Liste

Funcții utile: head, tail, take, drop, length

6. Să se implementeze următoarele funcții folosind liste:

a) `verifL` - verifică dacă lungimea unei liste date ca parametru este pară

```
verifL :: [Int] -> Bool
verifL = undefined
```

b) `takefinal` - pentru o listă dată ca parametru și un număr `n`, întoarce lista cu ultimele `n` elemente. Dacă lista are mai putin de `n` elemente, se intoarce lista nemodificată.

```
takefinal :: [Int] -> Int -> [Int]
takefinal = undefined
```

Cum trebuie să modificăm prototipul funcției pentru a putea fi folosită și pentru șiruri de caractere?

c) `remove` - pentru o listă și un număr `n` se întoarce lista din care se șterge elementul de pe poziția `n`. (Hint: puteți folosi funcțiile `take` și `drop`). Scriți si prototipul functiei.

Recursivitate pe Liste

Listele sunt definite inductiv: - vida `[]` - construită prin adăugarea unui element `head` unei liste existente `tail` (head:tail)

Recursivitatea pe liste se bazeaza pe definiția inductivă a lor.

Exemplu: Dată fiind o listă de numere întregi, să se scrie o funcție 'semiPareRec care elimină numerele impare și le injumătățește pe cele pare. De exemplu:

```
-- semiPareRec [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
```

```haskell
semiPareRec :: [Int] -> [Int]
semiPareRec [] = []
semiPareRec (h:t)
 | even h    = h `div` 2 : t'
 | otherwise = t'
 where t' = semiPareRec t
```

7. Exerciții: să se scrie urmatoarele funcții folosind recursivitate:

a) `myreplicate` - pentru un întreg `n` si o valoare `v` întoarce lista de lungime `n` ce are doar elemente egale cu `v`. Să se scrie și prototipul functiei.

b) `sumImp` - pentru o listă de numere întregi, calculează suma valorilor impare. Să se scrie și prototipul functiei.

c) `totalLen` - pentru o listă de șiruri de caractere, calculează suma lungimilor șirurilor care încep cu caracterul 'A'.

```haskell
totalLen :: [String] -> Int
totalLen = undefined
```