Laboratorul 2: Funcții

Exerciții

1. Scrieți o funcție `poly` cu patru argumente de tip `Double` (`a,b,c,x`) care calculează `a*x^2+b*x+c`. Scrieți și signatura funcției (`poly :: ??`).
2. Scrieți o funcție `eeny` care întoarce stringul "eeny" atunci când primește ca input un număr par și "meeny" când primeste ca input un număr impar. Hint: puteți folosi funcția `even`, despre care puteți citi pe https://hoogle.haskell.org/).

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

3. Scrieți 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 întoarce șirul vid. Scrieți două definiții pentru funcția `fizzbuzz`: una folosind `if` și una folosind gărzi (condiții). Hint: pentru a calcula restul împărțirii unui număr la un alt număr puteți folosi funcția `mod`.

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

Recursivitate

Una din diferențele dintre programarea declarativă și cea imperativă este modalitatea de abordare a problemei iterării: în timp ce în programarea imperativă folosim bucle (`while`, `for`, ...), în programarea declarativă folosim conceptul de recursivitate.

Un avantaj al folosirii recursivității este acela că ușurează sarcina de scriere și de verificare a corectitudinii programelor prin raționamente de tip inductiv: construim rezultatul pe baza rezultatelor unor subprobleme mai simple (aceeași problemă, dar pentru date de dimensiune mai mică).

Un exemplu simplu este calcularea unui element de la o poziție dată din secvența numerelor Fibonacci, definită recursiv astfel:

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

Putem transcrie această definiție în Haskell:

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

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

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

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

4. Numerele `tribonacci` sunt definite astfel:

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

Implementați funcția `tribonacci` dând o definiție bazată pe cazuri și una ecuațională, cu șabloane.

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

5. Scrieți o funcție recursivă care calculează coeficienții binomiali. Coeficienții sunt determinați folosind următoarele 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. Implementați 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 `l` dată ca parametru și un număr `n`, întoarce o listă care conține ultimele `n` elemente ale listei `l`. Dacă lista are mai puțin de `n` elemente, întoarce lista nemodificată.

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

Cum trebuie să modificăm prototipul funcției pentru a putea folosi funcția și pentru șiruri de caractere?

c) `remove` - pentru o listă și un număr `n`, întoarce lista primită ca parametru din care se șterge elementul de pe poziția `n`. (Hint: puteți folosi funcțiile `take` și `drop`). Scrieți și prototipul funcției.

Recursivitate și liste

Listele sunt definite inductiv:

- lista vidă: `[]`
- lista construită prin adăugarea unui element `head` unei liste `tail` deja existente: `(head:tail)`

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. Scrieți următoarele funcții folosind conceptul de recursivitate:

   a) `myreplicate` - pentru un întreg `n` și o valoare `v`, întoarce lista ce conține `n` elemente egale cu `v`. Să se scrie și prototipul funcției.

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

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

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