Laboratorul 5: Funcții de nivel înalt

În Haskell, funcțiile sunt *valori*. Putem să trimitem funcții ca argumente și sa le întoarcem ca rezultat.

Să presupunem că vrem să definim o funcție `aplica2` care primește ca argument o funcție `f` de tip `a -> a` și o valoare `x` de tip `a`, rezultatul fiind `f (f x)`. Tipul funcției `aplica2` este

```
aplica2 :: (a -> a) -> a -> a
```

Se pot da mai multe definiții:

```
aplica2 f x = f (f x)
aplica2 f = f . f
aplica2 =\f x -> f (f x)
aplica2 f = \x -> f (f x)
```

MAP

Funcția `map` are ca argumente o funcție de tip `a -> b` și o listă de elemente de tip `a`, rezultatul fiind lista elementelor de tip `b` obținute prin aplicarea funcției date pe fiecare element de tip `a`:

```
map :: (a -> b) -> [a] -> [b]
map f xs =[f x | x <- xs]
```

Exemple:

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
Prelude> map ($ 3) [ ( 4 +) , (10 * ) , ( ^ 2) , sqrt ]
[7.0,30.0,9.0,1.7320508075688772]
```

Încercați să găsiți valoarea expresiilor de mai jos și verificați răspunsul găsit de voi în interpretor:

```
map (\x -> 2 * x) [1..10]
map (1 `elem`) [[2,3], [1,2]]
map (`elem` [2,3]) [1,3,4,5]
```

FILTER

Funcția `filter` are ca argument o proprietate si o listă de elemente, rezultatul fiind lista elementelor care verifică acea proprietate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

Prelude> filter (>2) [3,1,4,2,5]
[3,4,5]
Prelude> filter odd [3,1,4,2,5]
[3,1,5]
```

FOLDR

Funcția `foldr` este folosită pentru agregarea unei colecții. O definiție intuitivă a lui `foldr` este:

```
foldr op unit [a1, a2, a3, ... , an] == a1 `op` a2 `op` a3 `op` .. `op` an `op` unit

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op i []     = i
foldr op i (x:xs) = x `op` (foldr f i xs)

Prelude> foldr (+) 0 [1..5]
15
Prelude> foldr (*) 1 [2,3,4]
24
Prelude> foldr (++) [] ["abc","def","ghi"]
"abcdefghi"
```

Exercitii

Rezolvați următoarele exerciții folosind map, filter și fold (fara recursivitate sau selectie). Pentru fiecare functie scrieti si prototipul acesteia.

1. Scrieți o funcție generică `firstEl` care are ca argument o listă de perechi de tip `(a,b)` și întoarce lista primelor elementelor din fiecare pereche:

```
firstEl [('a',3),('b',2), ('c',1)]
"abc"
```

2. Scrieți funcția `sumList` care are ca argument o listă de liste de valori `Int` și întoarce lista sumelor elementelor din fiecare listă (suma elementelor unei liste de întregi se calculează cu funcția `sum`):

```
sumList [[1,3], [2,4,5], [], [1,3,5,6]]
```

```
[4,11,0,15]
```

3. Scrieți o funcție `prel2` care are ca argument o listă de `Int` și întoarce o listă în care elementele pare sunt înjumătățite, iar cele impare sunt dublate:

```
*Main> prel2 [2,4,5,6]
[1,2,10,3]
```

4. Scrieți o funcție care primește ca argument un caracter și o listă de șiruri, rezultatul fiind lista șirurilor care conțin caracterul respectiv (folosiți funcția `elem`).
5. Scrieți o funcție care primește ca argument o listă de întregi și întoarce lista pătratelor numerelor impare.
6. Scrieți o funcție care primește ca argument o listă de întregi și întoarce lista pătratelor numerelor din poziții impare. Pentru a avea acces la poziția elementelor folosiți `zip`.
7. Scrieți o funcție care primește ca argument o listă de șiruri de caractere și întoarce lista obținută prin eliminarea consoanelor din fiecare șir.

```
numaiVocale ["laboratorul", "PrgrAmare", "DEclarativa"]
["aoaou","Aae","Eaaia"]
```

8. Definiți recursiv funcțiile `mymap` și `myfilter` cu aceeași funcționalitate ca și funcțiile predefinite.
9. Calculați suma pătratelor elementelor impare dintr-o listă dată ca parametru.
10. Scrieți o funcție care verifică faptul că toate elementele dintr-o listă sunt `True`, folosind foldr.
11.

(a) Scrieți o funcție care elimină un caracter din șir de caractere.

```haskell
rmChar :: Char -> String -> String
rmChar = undefined
```

(b) Scrieți o funcție recursivă care elimină toate caracterele din al doilea argument care se găsesc în primul argument, folosind `rmChar`.

```haskell
rmCharsRec :: String -> String -> String
rmCharsRec = undefined

-- rmCharsRec ['a'..'l'] "fotbal" == "ot"
```

(c) Scrieți o funcție echivalentă cu cea de la (b) care folosește `foldr` în locul recursiei si `rmChar`.

```haskell
rmCharsFold :: String -> String -> String
rmCharsFold = undefined
```