

## Laborator 3

### Din nou liste. Funcții de nivel înalt

**Atenție!** Înainte de a continua acest laborator terminați exercițiile din Laboratorul 2!

În rezolvarea exercițiilor folosiți fișierul lab3.hs, ce conține prototipurile funcțiilor pe care trebuie să le implementați și exemple de rulat.

**(L3.1) [Definirea listelor prin comprehensiune]** Reamintiți-vă definirea listelor prin comprehensiune din Laboratorul 2. Încercați să găsiți valoarea expresiilor de mai jos și verificați răspunsul găsit de voi în interpretor:

```
[ x^2 | x <- [1..10] , x `rem` 3 == 2]
[(x,y) | x<- [1..5] , y <- [x..(x+2)]]
[(x,y) | x<-[1..3] , let k = x^2, y <- [1..k]]
[ x | x<- "Facultatea de Matematica si Informatica" , elem x ['A'..'Z']]
[[x..y] | x <- [1..5] , y <- [1..5] , x < y]
```

Pentru a putea fi ușor copiate în interpretor expresiile sunt și în fișierul sursă.

**Exerciții** Deși în aceste exerciții vom lucra cu date de tip `Int`, rezolvați exercițiile de mai jos astfel încât rezultatul să fie corect pentru valori pozitive. Definițiile pot fi adaptate ușor pentru valori oarecare folosind funcția `abs`.

1. Folosind numai comprehensiunea definiți o funcție

```
factorsi :: Int -> [Int]
```

atfel încât `factorsi n` întoarcă lista divizorilor pozitivi ai lui `n`.

2. Folosind funcția `factorsi`, definiți predicatul `prim n` care întoarcă `True` dacă și numai dacă `n` este număr prim.

3. Folosind numai comprehensiunea și funcțiile definite anterior, definiți funcția

```
numerePrime :: Int -> [Int]
```

astfel încât `numerePrime n` întoarce lista numerelor prime din intervalul  $[2..n]$ .

**(L3.2) [Funcția zip]** Testați și sesizați diferența:

```
Prelude> [(x,y) | x <- [1..5], y <- [1..3]]
Prelude> zip [1..5] [1..3]
```

Definiți funcția `myzip3` care se comportă asemenea lui `zip` dar are trei argumente:

```
*Main> myzip3 [1,2,3] [1,2] [1,2,3,4]
[(1,1,1),(2,2,2)]
```

**Funcții de nivel înalt** În Haskell, funcțiile sunt *valori*. Putem să trimitem funcții ca argumente și să 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)
```

**(L3.3) [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]
```

Reamintiți-vă noțiunea de *secțiune* definită la curs: o *secțiune* este aplicarea parțială a unui operator, adică se obține dintr-un operator prin fixarea unui argument. De exemplu `(*3)` este o funcție cu un singur argument, rezultatul fiind argumentul înmulțit cu 3, `(10-)` este o funcție cu un singur argument, rezultatul fiind diferența dintre 10 și argument. Următoarele exemple - discutate la curs - folosesc secțiuni și funcția `map`:

```
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]
```

**Exerciții** Rezolvați exercițiile folosind **map**. În fiecare caz scrieți tipul funcției respective.

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 **Integer** 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 **Integer** 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]
```

**(L3.4) [map, filter]** Funcția **filter** are ca argument o proprietate și 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]
```

**Exerciții** Rezolvați aceste exerciții fără recursie, folosind `filter` și `map`.

1. 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`).
2. Scrieți o funcție care primește ca argument o listă de întregi și întoarce lista pătratelor numerelor impare.
3. 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`.
- 4\*. Scrieți o funcție care primește ca argument o listă de șiruri de caractere și întoarce lista obținută prin eliminarea consoanelor din fiecare șir. Rezolvați exercițiul folosind numai `filter`, `map` și `elem`.

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

**(L3.5) [mymap, myfilter]** Definiți recursiv funcțiile `mymap` și `myfilter` cu aceeași funcționalitate ca și funcțiile predefinite.

- Citiți capitolul *Higher order functions* din  
M. Lipovaca, Learn You a Haskell for Great Good!  
<http://learnyouahaskell.com/higher-order-functions>