# Laboratorul 4: Exerciții liste, map, filter

## Liste

1. Reamintiți-vă definirea listelor prin selecție din **Laboratorul 3**. Încercați să aflați valoarea expresiilor de mai jos (fără a folosi interpretorul), iar apoi verificați-vă răspunsurile folosind `ghci`.

```
{-
[ 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 ]
-}
```

Deși în exercițiile de mai jos vom lucra cu date de tip `Int`, este suficient să presupuneți în soluțiile voastre că lucrăm doar cu valori pozitive (definițiile pot fi adaptate ușor pentru valori oarecare folosind funcția `abs`).

2. Definiți o funcție `factori` care întoarce lista divizorilor pozitivi ai unui număr primit ca parametru. Folosiți doar metoda de definire a listelor prin selecție.

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

3. Folosind funcția `factori`, definiți predicatul `prim`, care verifică dacă un număr primit ca parametru este prim.

```
prim :: Int -> Bool
prim = undefined
```

4. Definiți funcția `numerePrime`, care pentru un număr `n` primit ca parametru, întoarce lista numerelor prime din intervalul `[2..n]`. Folosiți metoda de definire a listelor prin selecție și funcțiile definite anterior.

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

## Funcția `zip`

Evaluați expresiile de mai jos folosind interpretorul și observați diferența între cele două rezultate:

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

```
Prelude> zip [1..5] [1..3]
```

5. Definiți funcția `myzip3` ca o generalizare a funcției `zip` pentru trei argumente:

```
myzip3 [1,2,3] [1,2] [1,2,3,4] == [(1,1,1),(2,2,2)]
```

## Secțiuni

Reamintiți-vă noțiunea de ***secțiune*** definită la curs: o ***secțiune*** este aplicarea parțială a unui operator; se obține dintr-un operator prin fixarea unui argument. De exemplu:

p `(*3)` este o funcție cu un singur parametru, rezultatul fiind parametrul înmulțit cu `3`,

`(10-)` este o funcție cu un singur parametru, rezultatul fiind diferența dintre `10` și parametru.

## Lambda expresii

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 parametru 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 parametri 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 aflați valoarea expresiilor de mai jos, iar apoi verificați răspunsul găsit de voi folosind interpretorul:

```
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 parametri o proprietate (predicat) 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 următoarele exerciții folosind `map` și `filter` (fără a folosi funcții recursive sau metoda de definire a listelor prin selecție). Pentru fiecare funcție scrieți și prototipul acesteia.

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

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

7. Scrieți funcția `sumList` care are ca parametru 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]
```

8. Scrieți o funcție `prel2` care are ca parametru o listă de întregi (`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]
```

9. Scrieți o funcție care primește ca parametri un caracter și o listă de șiruri de caractere, și întoarce lista șirurilor care conțin caracterul primit ca argument (hint: folosiți funcția `elem`).

10. Scrieți o funcție care are ca parametru o listă de întregi și întoarce lista pătratelor numerelor impare din acea listă.

11. Scrieți o funcție care primește ca argument o listă de întregi și întoarce lista pătratelor elementelor din poziții impare. Hint: folosiți `zip` pentru a avea acces la poziția elementelor.
12. Scrieți o funcție care primește ca parametru o listă de șiruri de caractere și întoarce lista obținută prin eliminarea consoanelor din fiecare șir.

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

13. Definiți recursiv funcțiile `mymap` și `myfilter` cu aceeași funcționalitate ca a funcțiilor `map` și `filter` predefinite.

## Extra

Un joc *turn-based* constă dintr-o succesiune de runde, fiecare dintre ele corespunzând mutării unui jucător. Pentru acest exercițiu suplimentar, veți implementa un mecanism simplu al jocului X și 0, care este *turn-based*.

- Definiți funcția `step` care are drept parametri un caracter `p` (`'X'` sau `'0'`) și o configurație `c` a jocului (o listă de "celule" ce pot fi goale, marcate cu `'X'` sau cu `'0'`) și care întoarce lista tuturor configurațiilor ce pot fi obținute din `c` adăugând caracterul `p` într-una din celulele goale.
- Definiți o funcție `next` care are drept parametri un caracter `p` și o listă de configurații `cs` și produce lista tuturor configurațiilor care pot fi obținute din `cs` și `p` folosind funcția `step`. Hint: folosiți `map` și concatenarea de liste `concat`.
- Scrieți o funcție `win` care are ca parametri un caracter `p` și o listă de configurații `cs` și produce lista tuturor configurațiilor din `cs` pentru care `p` ar fi câștigător. Hint: folosiți `filter`.
- Combinați `next` și `win` pentru a simula un joc complet de X și 0. Hint: recursivitate.
- Generalizați jocul de X și 0 pentru table de joc nxn și tridimensionale.