```
-- Laboratorul 4

-- 0. Ce afiseaza fiecare?

-- a. [ x^2 |x <- [1..10], x `rem` 3 == 2]
-- [4, 25, 64]

-- b. [(x,y)| x<- [1..5], y <- [x..(x+2)]]
--
[(1,1),(1,2),(1,3),(2,2),(2,3),(2,4),(3,3),(3,4),(3,5),(4,4),(4,5),(4,6),(5,5),(5
,6),(5,7)]

-- c. [(x,y)| x<-[1..3], let k = x^2, y <- [1..k]]
--
[(1,1),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4),(3,5),(3,6),(3,7),(3,8),(3
,9)]

-- d. [ x | x<- "Facultatea de Matematica si Informatica", elem x ['A'..'Z']]
-- "FMI"

-- e. [[x..y]| x <- [1..5], y <- [1..5], x < y]
--
[[1,2],[1,2,3],[1,2,3,4],[1,2,3,4,5],[2,3],[2,3,4],[2,3,4,5],[3,4],[3,4,5],[4,5]]

-- 1. Folosind numai metoda prin selectie definiti o functie
-- atfel încât functia factori n întoarce lista divizorilor pozitivi ai lui n.

factori :: Int -> [Int]
factori n = [x| x<-[1,n], n`mod`x==0]

-- 2. Folosind functia factori, definiti predicatul prim n care întoarce
-- True dacă si numai dacă n este număr prim.

prim :: Int -> Bool
prim n = length(factori n) == 2

--  3. Folosind numai metoda prin selectie si functiile definite anterior,
-- definiti functia numerePrime astfel încât numerePrime n întoarce lista
-- numerelor prime din intervalul [2..n].

numerePrime :: Int -> [Int]
numerePrime n = [x | x<-[2..n], prim x]

-- 4. Definiti functia myzip3 care se comportă asemenea lui zip
-- dar are trei argumente:
```

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

myzip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
myzip3 x y z = [(p, n, m) | ((p, n), m) <- zip (zip x y) z]

myzip33 :: [a] -> [b] -> [c] -> [(a, b, c)]
myzip33 [] _ _ = []
myzip33 _ [] _ = []
myzip33 _ _ [] = []
myzip33 (x:xs) (y:ys) (z:zs) = (x, y, z) : myzip33 xs ys zs

myzip333 :: [a] -> [b] -> [c] -> [(a, b, c)]
myzip333 x y z = if (length x == 0 || length y == 0 || length z == 0) then []
                 else [(head x, head y, head z)] ++ myzip333 (tail x) (tail y)
(tail z)

-- 0.1. Ce afiseaza fiecare?

-- a. map (\x -> 2 * x) [1..10]
-- [2,4,6,8,10,12,14,16,18,20]

-- b. map (1 `elem`) [[2,3], [1,2]]
-- [False,True]

-- c. map (`elem` [2,3]) [1,3,4,5]
-- [False,True,False,False]

-- 5. Scrieti o functie generică firstEl care are ca argument o listă de
-- perechi de tip (a,b) si întoarce lista primelor elementelor
-- din fiecare pereche:

firstEl :: [(a, b)] -> [a]
firstEl = map fst

firstEl2 :: [(a, b)] -> [a]
firstEl2 = map (\(x,y) -> x)

-- 6. Scrieti functia sumList care are ca argument o listă de liste de
-- valori Int si întoarce lista sumelor elementelor din fiecare listă
-- (suma elementelor unei liste de întregi se calculează cu functia sum):

sumList :: [[Int]] -> [Int]
sumList = map sum

-- 7. Scrieti o functie prel2 care are ca argument o listă de Int
```

```
-- si întoarce o listă în care elementele pare sunt înjumătătite,
-- iar cele impare sunt dublate:

prel2 :: [Int] -> [Int]
prel2  = map (\x -> if odd x then 2*x
                    else x `div` 2)

prel22 :: [Int] -> [Int]
prel22 [] = []
prel22 (h:t)
    | odd h = 2 * h : prel22 t
    | otherwise = h `div` 2 : prel22 t

-- 8. Scrieti o functie care primeste ca argument un caracter
-- si o listă de siruri, rezultatul fiind lista sirurilor care contin
-- caracterul respectiv (folositi functia elem).

functie8 :: Char -> [String] -> [String]
functie8 c = filter (elem c)

-- 9. Scrieti o functie care primeste ca argument o listă de
-- întregi si întoarce lista pătratelor numerelor impare.

functie9 :: [Int] -> [Int]
functie9 lista = map (\x -> x*x) (filter odd lista)

-- 10. Scrieti o functie care primeste ca argument o listă de întregi
-- si întoarce lista pătratelor numerelor din pozitii impare. Pentru a
-- avea acces la pozitia elementelor folositi zip.

functie10 :: [Int] -> [Int]
functie10 lista = map (\(x, y) -> y*y) (filter (\(x, y) -> odd x) (zip [0..]
lista))

-- 11. Scrieti o functie care primeste ca argument o listă de siruri de caractere
-- si întoarce lista obtinută prin eliminarea consoanelor din fiecare sir.

numaiVocale :: [String] -> [String]
numaiVocale = map (filter (`elem` "aeiouAEIOU"))

-- 12. Definiti recursiv functiile mymap si myfilter cu aceeasi
-- functionalitate ca si functiile predefinite.

mymap :: (a -> b) -> [a] -> [b]
mymap _ [] = []
```

```
mymap f (h:t) = f h : mymap f t

myfilter :: (a -> Bool) -> [a] -> [a]
myfilter _ [] = []
myfilter f (h:t) = if f h then h : myfilter f t
                   else myfilter f t
```