```haskell
-- Laboratorul 1

-- Observam ca myInt nu se mai poate modifica, e constanta.
-- Definirea unei functii:
----- (a -> a): de la un anume tip de date a la acelasi tip de date
----- folosim =

import Data.List
myInt = 55555555555555555555555555555555555555555555555555
double :: Integer -> Integer
double x = x + x

-- Scrieti o functie ce determina maximul dintre doua numere, respectiv trei.

maxim :: Integer -> Integer -> Integer
maxim x y = if (x > y)
              then x
          else y

max3 x y z = let
             u = maxim x y
             in (maxim u z)

-- 6. Să se scrie următoarele functii:

-- a) functie cu 2 parametri care calculeaza suma pătratelor celor două numere;

patrate :: Integer -> Integer -> Integer
patrate x y = x*x + y*y

-- b) functie cu un parametru ce întoarce mesajul “par” dacă parametrul este par
-- si “impar” altfel;

paritate :: Integer -> String
paritate p = if mod p 2 == 1
               then "impar"
             else "par"

-- c) functie care calculează factorialul unui număr;

factorial :: Integer -> Integer
factorial x = if x == 0
                then 1
            else x*factorial(x-1)

-- d) functie care verifică dacă un primul parametru este mai mare decât dublul
-- celui de-al doilea parametru.

dublu :: Integer -> Integer -> Bool
dublu x y = if x>2*y
                then True
            else False
```

```haskell
-- Laboratorul 1 (Macovei)

-- 1. Jocul Piatra Foarfeca Hartie

data Alegere = Piatra | Foarfeca | Hartie
               deriving (Eq, Show)

-- Folosim Eq pentru a preciza ca putem stabili egalitate intre doua date.
-- Folosim Show pentru a putea afisa datele sub forma de caractere.

data Rezultat = Victorie | Infrangere | Egalitate
                deriving (Eq, Show)

partida :: Alegere -> Alegere -> Rezultat
partida Piatra Foarfeca = Victorie
partida Piatra Hartie = Infrangere
partida Piatra Piatra = Egalitate
partida Foarfeca Piatra = Infrangere
partida Foarfeca Hartie = Victorie
partida Foarfeca Foarfeca = Egalitate
partida Hartie Hartie = Egalitate
partida Hartie Foarfeca = Infrangere
partida Hartie Piatra = Victorie

-- Pentru a scapa de trei cazuri, putem pune la final un caz general, de pattern
-- matching. Pe acest caz se va intra dupa ce toate celelate cazuri au fost
-- verificate. De aceea este foarte important ca acesta sa fie pus la final:
---- partida Piatra Foarfeca = Victorie
---- partida Piatra Hartie = Infrangere
---- partida Foarfeca Piatra = Infrangere
---- partida Foarfeca Hartie = Victorie
---- partida Hartie Foarfeca = Infrangere
---- partida Hartie Piatra = Victorie
---- partida _ _ = Egalitate
-- Daca am pune cazul la inceput, am avea o generalizare care va genera eroare.

-- 2. Compunerea functiilor

f :: Int -> Int
f x = x + 1

g :: Int -> Int
g x = 2*x
```

`-- *Main > f (g 2)`
`-- 5`
`-- *Main > (f . g) 2`
`-- 5`
`-- *Main > f $ g 2`
`-- 5`
	`-- *Main > g (f 2)`
	`-- 6`
	`-- *Main > (g . f) 2`
	`-- 6`
	`-- *Main > g $ f 2`
	`-- 6`
		`-- *Main > f (f (g (f (g 2))))`
		`-- 12`
		`-- *Main > (f . f . g . f . g) 2`
		`-- 12`
		`-- *Main > f $ f $ g $ f $ g 2`
		`-- 12`

```haskell
-- 3. Recursivitate. Suma elementelor dintr-o lista

suma :: [Int] -> Int
suma [] = 0
suma (head:tail) = head + suma tail


-- Laboratorul 2

-- 1. Să se scrie o functie poly2 care are patru argumente de tip Double, a,b,c,x
-- si calculează a*x^2+b*x+c. Scrieti si signatura functiei (poly :: ceva).

poly :: Double -> Double -> Double -> Double -> Double
poly a b c x = a*x*x+b*x+c

-- 2. Să se scrie o functie eeny care întoarce “eeny” pentru input par si “meeny”
-- pentru input impar.
-- Hint: puteti folosi functia even (căutati pe https://hoogle.haskell.org/).

eeny :: Integer -> String
eeny x = if mod x 2 == 0 then "eeny" else "meeny"

eeny2 :: Integer -> String
eeny2 x = if even x then "eeny" else "meeny"

eeny3 :: Integer -> String
eeny3 x
    | even x = "eeny"
    |otherwise = "meeny"

-- 3. Să se scrie o functie fizzbuzz care întoarce “Fizz” pentru numerele
-- divizibile cu 3, “Buzz” pentru numerele divizibile cu 5 si “FizzBuzz” pentru
-- numerele divizibile cu ambele. Pentru orice alt număr se întoarce sirul vid.
-- Pentru a calcula modulo a două numere puteti folosi functia mod. Să se scrie
-- această functie în 2 moduri: folosind if si folosind gărzi (conditii).
```

```haskell
fizbuzz :: Integer -> String
fizbuzz x
    | mod x 3 == 0 && mod x 5 == 0 = "fizbuzz"
    | mod x 3 == 0 = "fizz"
    | mod x 5 == 0 = "buzz"
    | otherwise = ""

fizbuzz2 :: Integer -> String
fizbuzz2 x
    | mod x 15 == 0 = "fizbuzz"
    | mod x 3 == 0 = "fizz"
    | mod x 5 == 0 = "buzz"
    | otherwise = ""

fizbuzz3 :: Integer -> String
fizbuzz3 x = if mod x 15 == 0 then "fizbuzz"
else if mod x 3 == 0 then "fizz"
else if mod x 5 == 0 then "buzz"
else ""

-- Un foarte simplu exemplu de recursie este acela al calculării
-- unui element de index dat din secventa numerelor Fibonacci.

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

fibonacciEcuational :: Integer -> Integer
fibonacciEcuational 0 = 0
fibonacciEcuational 1 = 1
fibonacciEcuational n = fibonacciEcuational (n - 1) + fibonacciEcuational (n - 2)

-- 4. Numerele tribonacci sunt definite de ecuatia
--- Tn = 1 dacă n = 1
-------- 1 dacă n = 2
-------- 2 dacă n = 3
-------- Tn-1 + Tn-2 + Tn-3 dacă n > 3
-- Să se implementeze functia tribonacci atât cu cazuri cât si ecuational.

tribonacci :: Integer -> Integer
tribonacci 1 = 1
tribonacci 2 = 1
tribonacci 3 = 2
tribonacci n = tribonacci(n-1) + tribonacci(n-2) + tribonacci(n-3)
```

```haskell
tribonacci2 :: Integer -> Integer
tribonacci2 n
    | n == 1 = 1
    | n == 2 = 1
    | n == 3 = 2
    | otherwise = tribonacci2(n-1) + tribonacci2(n-2) + tribonacci2(n-3)

-- 5. Să se scrie o functie care calculează coeficientii binomiali, folosind
-- recursivitate. Acestia sunt determinati folosind urmatoarele ecuatii.
-- 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 n k
    | n == 0 = 0
    | k == 0 = 1
    | otherwise = binomial (n-1) k + binomial (n-1) k-1

binomial2 :: Integer -> Integer -> Integer
binomial2 n 0 = 1
binomial2 0 k = 1
binomial2 n k = binomial (n-1) k + binomial2 (n-1) k+1

---------------- FUNCTII -----------------
-- 1. head -- primul element al unei liste
----- tail -- restul listei
----- head :: [a] -> a
----- head(h:t) = h
----- tail:: [a] -> a
----- tail(h:t) = t
----- "abc" --> head-ul este a si tail-ul este restul listei
-- 2. take -- primele n elemente din lista l
----- take :: [a] -> a
----- take n l (apel)
-- 3. drop -- scapam de primele k elemente din lista
----- drop :: [a] -> [a]
----- drop 2 "abcd" -> "cd" (am scapat de primele 2)
-- 4. length -- lungimea listei
----- length "abcd" -> 4

-- Să se implementeze următoarele functii folosind liste:

-- a) verifL - verifică dacă lungimea unei liste date ca parametru este pară
```

```haskell
verifL :: [Int] -> Bool
verifL a = if even(length a) then True else False

verifL2 :: [Int] -> Bool
verifL2 a
    | even (length a) = True
    |otherwise = False

-- b) takefinal - pentru o listă dată ca parametru si un număr n, întoarce lista
-- cu ultimele n elemente. Dacă lista are mai putin de n elemente, se intoarce
-- lista nemodificată.
-- Cum trebuie să modificăm prototipul functiei pentru a putea fi folosită si
-- pentru siruri de caractere?

takefinal :: [Int] -> Int -> [Int]
takefinal l n = if (length l) < n then l else drop ((length l) - n) l

takefinall :: [Int] -> Int -> [Int]
takefinall l n
    | length l < n = l
    | otherwise = drop(length l - n) l

-- c) remove - pentru o listă si un număr n se întoarce lista din care se sterge
-- elementul de pe pozitia n. (Hint: puteti folosi functiile take si drop).
-- Scrieti si prototipul functiei.
-- ++ este folosit pentru a aduna doua liste

remove :: [Int] -> Int -> [Int]
remove l n = take(n-1) l ++ drop n l

-- 7.Să se scrie urmatoarele functii folosind recursivitate:

-- a) myreplicate - pentru un întreg n si o valoare v întoarce lista de lungime n
-- ce are doar elemente egale cu v. Să se scrie si prototipul functiei.

myreplicate :: Int -> val -> [val]
myreplicate n v
    | n == 0 = []
    | otherwise = myreplicate(n-1) v ++ [v]

-- b) sumImp - pentru o listă de numere întregi, calculează suma valorilor
-- impare. Să se scrie si prototipul functiei.

sumImp :: [Int] -> Int
sumImp [] = 0
```

```haskell
sumImp (h:t)
    | even h = sumImp t
    | otherwise = h + sumImp t

-- c) totalLen - pentru o listă de siruri de caractere, calculează suma
-- lungimilor sirurilor care încep cu caracterul ‘A’

totalLen :: [String] -> Int
totalLen [] = 0
totalLen (h:t)
    | take 1 h == "A" = totalLen t + length h
    | otherwise = totalLen t


-- Laboratorul 2 (Macovei)

-- 1. Integer -> numerele pot fi oricat de mari, putem face
-- operatii cu numere oricat de mari
-- ex: myInt = 99999...999

-- 2. Haskell nu suporta abordari iterative.
-- Implementarea operatiilor pentru structuri repetitive se face prin recursie.

-- 3. Prelucrarea listelor de elemente

-- [Int] -> lista de Int-uri
-- [Char] -> lista de Char-uri, echivalent cu un String

-- [1, 2, 3, 4] are un head [1] si un tail [2, 3, 4] si se noteaza (h:t)

-- Denumim functia lenght deoarece daca o denumim corect, length, aceasta
-- va coincide cu functia predefinita din Haskell si va genera
-- eroare deoarece nu o putem suprascrie.

lenght :: [Integer] -> Integer
lenght [] = 0
lenght (h:t) = 1 + lenght t

-- lenght [1, 2, 3, 4] =
--    1 + lenght [2, 3, 4]
--        1 + lenght [3, 4]
--            1 + lenght [4]
--                1 + lenght []
--                    0
```

```haskell
-- 4. Scrieti functia semiPare care primeste o lista de intregi ca parametru.
-- Aceasta elimina numerele impare si imparte la 2 numerele pare.
-- semiPare [0, 2, 1, 7, 8, 56, 17, 18] = [0, 1, 4, 28, 9]

-- Functia odd -> daca un elemnt este impar
-- Functia even -> daca un element este par

semiPare :: [Integer] -> [Integer]
semiPare [] = []
semiPare (h:t)
    | odd h = semiPare t
    | otherwise = div h 2 : semiPare t

-- [0, 2, 1, 7, 8, 56, 17, 18]
-- 0/2 = 0 : semiPare [2, 1, 7, 8, 56, 17, 18]
--         2/2 = 1 : semiPare [1, 7, 8, 56, 17, 18]
--                 : semiPare [7, 8, 56, 17, 18]
--                 : semiPare [8, 56, 17, 18]
--                 8/2 = 4 : semiPare [56, 17, 18]
--                           56/2 = 28 : semiPare [17, 18]
--                                       semiPare [18]
--                                       18/2 = 9 : semiPare[]
--                                                  []

semiPare2 :: [Integer] -> [Integer]
semiPare2 list
    | null list = list
    | even h = div h 2 : t'
    | otherwise = t'
    where h = head list
          t = tail list
          t' = semiPare2 t

-- Listele definite prin Comprehensiune
-- {0, 2, 4, 6, 8, 10} => {x | x apartine {1, .. 10}, x par}
-- [0, 2, 4, 6, 8, 10] = [x | x<-[0..10], even x]
--                     = [2*x | x<-[0..5]]

-- Pentru a accesa elementul de pe o pozitie dintr-o lista: lista !! pozitie
-- De exemplu, pentru a extrage mijlocul unei liste folosim:
-- lista !! (div (lenght lista) 2)

semiPareCom :: [Integer] -> [Integer]
semiPareCom list = [div x 2 | x<-list, even x]  -- div x 2 = x `div` 2
```

```haskell
-- Laboratorul 3

import Data.Char

-- 1. Sa se scrie o functie nrVocale care pentru o lista de siruri de caractere,
-- calculeaza numarul total de vocale ce apar în cuvintele palindrom. Pentru a
-- verifica daca un sir e palindrom, puteti folosi functia reverse, iar pentru a
-- cauta un element într-o lista puteti folosi functia elem. Puteti define
-- oricâte functii auxiliare.

-- Varianta 1

nrVocale :: [String] -> Int
nrVocale [] = 0
nrVocale (h:t)
    | h == reverse h = nrVocale t + countVocale h
    | otherwise = nrVocale t

countVocale :: [Char] -> Int
countVocale [] = 0
countVocale (h:t)
    | h == 'a' = countVocale t + 1
    | h == 'e' = countVocale t + 1
    | h == 'i' = countVocale t + 1
    | h == 'o' = countVocale t + 1
    | h == 'u' = countVocale t + 1
    | h == 'A' = countVocale t + 1
    | h == 'E' = countVocale t + 1
    | h == 'I' = countVocale t + 1
    | h == 'O' = countVocale t + 1
    | h == 'U' = countVocale t + 1
    | otherwise = countVocale t

-- Varianta 2

nrVocale3 :: [String] -> Int
nrVocale3 l = sum[countVocale3 x | x<-l, x == reverse x]

countVocale3 :: [Char] -> Int
countVocale3 s = sum [1 | c<-s, c `elem` "aeiouAEIOU"]

-- Varianta 3

nrVocale2 :: [String] -> Int
nrVocale2 [] = 0
```

```haskell
nrVocale2 (h:t)
    | h == reverse h = nrVocale2 t + countVocale2 h
    | otherwise = nrVocale2 t

countVocale2 :: [Char] -> Int
countVocale2 [] = 0
countVocale2 (h:t)
    | elem h "aeiouAEIOU" = countVocale2 t + 1
    | otherwise = countVocale2 t

-- 2. Sa se scrie o functie care primeste ca parametru un numar si o lista de
-- întregi, si adauga elementul dat dupa fiecare element par din lista. Sa se
-- scrie si prototipul functiei.

f :: Int -> [Int] -> [Int]
f n [] = []
f n (h:t)
    | even h = h : n : f n t
    | otherwise = h : f n t

-- 3. Sa se scrie o functie care are ca parametru un numar întreg si determina
-- lista de divizori ai acestui numar. Sa se scrie si prototipul functiei.

divizori :: Int -> [Int]
divizori n = [x | x <- [1..n], mod n x == 0]

-- 4. Sa se scrie o functie care are ca parametru o lista de numere întregi si
-- calculeaza lista listelor de divizori.

listadiv :: [Int] -> [[Int]]
listadiv l = [divizori x | x <- l]

-- 5. Scrieti o functie care date fiind limita inferioara si cea superioara
-- (întregi) a unui interval închis si o lista de numere întregi, calculeaza
-- lista numerelor din lista care apartin intervalului.

-- a. Folositi doar recursie. Denumiti functia inIntervalRec.

inIntervalRec :: Int -> Int -> [Int] -> [Int]
inIntervalRec x y [] = []
inIntervalRec x y (h:t)
    | elem h [x..y] = [h] ++ inIntervalRec x y t
    | otherwise = inIntervalRec x y t
```

```haskell
inIntervalRec :: Int -> Int -> [Int] -> [Int]
inIntervalRec _ _ [] = []
inIntervalRec inf sup (h:t)
    | h >= inf && h <= sup = h : inIntervalRec inf sup t
    | otherwise = inIntervalRec inf sup t

-- b. Folositi descrieri de liste. Denumiti functia inIntervalComp.

inIntervalComp :: Int -> Int -> [Int] -> [Int]
inIntervalComp s d l = [x | x <- l, x >= s && x <= d]

-- 6. Scrieti o functie care numara câte numere strict pozitive sunt într-o lista
-- data ca argument.

-- a. Folositi doar recursie. Denumiti functia pozitiveRec.

pozitiveRec :: [Int] -> Int
pozitiveRec [] = 0
pozitiveRec (h:t)
    | h > 0 = pozitiveRec t + 1
    | otherwise = pozitiveRec t

pozitiveRec :: [Int] -> Int
pozitiveRec [] = 0
pozitiveRec (h:t) = if h >= 0 then 1 + pozitiveRec t else pozitiveRec t

-- b. Folositi descrieri de liste. Denumiti functia pozitiveComp.
-- Nu puteti folosi recursie, dar veti avea nevoie de o functie de agregare.

pozitiveComp :: [Int] -> Int
pozitiveComp l = sum[1 | x<-l, x>0]

pozitiveComp :: [Int] -> Int
pozitiveComp list = length [x | x<-list, x>=0]

-- 7. Scrieti o functie care data fiind o lista de numere calculeaza lista
-- pozitiilor elementelor impare din lista originala.

-- a. Folositi doar recursie. Denumiti functia pozitiiImpareRec. Indicatie:
-- folositi o functie ajutatoare, cu un argument în plus reprezentând pozitia
-- curenta din lista.

pozitiiImpareRec :: Int -> [Int] -> [Int]
pozitiiImpareRec _ [] = []
pozitiiImpareRec i (h:t)
```

```haskell
    | odd h = i : pozitiiImpareRec (i+1) t
    | otherwise = pozitiiImpareRec (1+i) t

-- pozitiiImpare [0, 1, 2, 3] 0
--     pozitiiImpare [1, 2, 3] 1
--         1 : pozitiiImpare [2, 3] 2
--                 pozitiiImpare [3] 3
--                     3 : pozitiiImpare [] 4
--                             []

-- b. Folositi descrieri de liste. Denumiti functia pozitiiImpareComp.

pozitiiImpareComp :: [Int] -> [Int]
pozitiiImpareComp l = [poz |(elem, poz) <- zip l [0..], odd elem]

-- 8. Scrieti o functie care calculeaza produsul tuturor cifrelor care apar în
-- sirul de caractere dat ca intrare. Daca nu sunt cifre în sir, raspunsul
-- functiei trebuie sa fie 1.
-- De exemplu:
-- -- multDigits "The time is 4:25" == 40
-- -- multDigits "No digits here!" == 1

-- Indicatie: Veti avea nevoie de functia isDigit care verifica daca un caracter
-- e cifra si functia digitToInt care transforma un caracter in cifra. Cele 2 în
-- functii se afla pachetul Data.Char.

-- a. Folositi doar recursie. Denumiti functia multDigitsRec.

multDigitsRec :: String -> Int
multDigitsRec "" = 1
multDigitsRec (h:t)
    | isDigit h = digitToInt h * multDigitsRec t
    | otherwise = multDigitsRec t

-- Tratam String ca [Char] si folosim tot (h:t) pentru a parcurge.
-- In cazul in care aveam [Char] in signatura functiei, "" devenea [].

-- b. Folositi descrieri de liste. Denumiti functia multDigitsComp

multDigitsComp :: String -> Int
multDigitsComp s = product[digitToInt x| x <- s, isDigit x]
```

```haskell
-- Laboratorul 3 (Macovei)

-- 1. Functiile de nivel inalt

-- Nu este suficient sa spunem f :: a -> a deoarece trebuie sa precizam ce tip
-- de clasa este a pentru a stii cum sa se comporte +.

f :: (Num a) => a -> a
f x = x + 1

aplica2 :: (a -> a) -> a -> a
aplica2 f n = f (f n)

-- 2. Doua functii de nivel inalt sunt map si filter.

-- Map aplica functia data pe fiecare element al listei.
-- map :: (a -> b) -> [a] -> [b]
-- map f list = [f x | x <- list]

-- Toate rezultatele de mai jos sunt echivalente -> [2, 3, 4]
rezultat = map f [1, 2, 3]
rezultat2 = map (\x -> x + 1) [1, 2, 3]   -- lambda expresie
rezultat3 = map (+1) [1, 2, 3]            -- sectiune

-- Va calcula fiecare functie din lista cu argumentul 3:
-- map ($3) [(4+), (10*), (^2), sqrt]
-- [7.0,30.0,9.0,1.7320508075688772]

-- Filter filtreaza lista dupa o annumita proprietate data ca functie.
-- filter :: (a -> Bool) -> [a] -> [b]

-- filter (\x -> x>2) [1, 2, 3, 4] => [3, 4]

-- 3. Ordonare folosind comprehensiunea (verificati daca o lista este ordonata).

-- and [True, False, False] => False
-- and [True, True] => True
-- and [1<2, 2<3, 3<4] => True

ordonataNat :: [Int] -> Bool
ordonataNat [] = True
ordonataNat [x] = True
ordonataNat (h:t) = (and [h <= y | y <- t]) && ordonataNat t

-- Aceeasi functie, dar fara comprehensiune, doar recursiv.
```

```haskell
-- Extragem primele doua elemente ale listei pentru a le compara doua cate doua
-- pana ajungem la ultimul element.

ordonataNat2 :: [Int] -> Bool
ordonataNat2 [] = True
ordonataNat2 [x] = True
ordonataNat2 (h1 : h2 : t) = (h1 <= h2) &&  ordonataNat2 (h2:t)

-- Scrieti aceasta functie de ordonare pe cazuri generale.
-- ordonataNat = ordonata list(<=)
-- Functia primeste o lista, un predicat si un raspuns boolean.

ordonata :: [a] -> (a -> a -> Bool) -> Bool
ordonata [] _ = True
ordonata [x] _ = True
ordonata (h:t) rel = (and [rel h y | y <- t]) && ordonata t rel

-- ordonata (reverse [1..10]) (>) = True

-- 4. Definiti operatorul *<* pe tupluri.

(*<*) :: (Integer, Integer) -> (Integer, Integer) -> Bool
(*<*)(a, b)(c, d)
    |(a < c) || (a == c && b <= d) = True
    | otherwise = False


-- 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"
```

```haskell
-- 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]

factori :: Int -> [Int]
factori n = [x | x <- [1,n], rem n 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)

zip3333 :: [a] -> [b] -> [c] -> [(a, b, c)]
```

```haskell
zip3333 list1 list2 list3 = [(x, y, z) | (x, p1) <- zip list1 [0..], (y, p2) <-
zip list2 [0..], (z, p3) <- zip list3 [0..], (p1 == p2 && p2 == p3)]

-- 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]
-- Verifica daca 1 este element in listele mici din lista mare

-- c. map (`elem` [2,3]) [1,3,4,5]
-- [False,True,False,False]
-- Verifica fiecare element din lista [1, 3, 4, 5] daca se afla in lista [2, 3]

-- 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 list = map fst list

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

-- 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 list = map sum list

-- 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 list = map (\x -> if odd x then 2*x
                   else x `div` 2) list

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 list = filter (elem c) list

-- 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)

functie9 :: [Int] -> [Int]
functie9 lista = map (^2)(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))

-- lista = [1, 2, 3, 4, 5]
-- Aplic zip [0..] lista => [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)]
-- Filtrez dupa pozitiile impare => [(1, 2), (3, 4)]
-- Fac map pe lista aceasta ca al doilea elem sa fie selectat si sa fie ridicat
-- la patrat => [4, 16]

-- 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
```

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


-- Laboratorul 4 (Macovei)

-- 1. Compunerea unor functii (pentru fiecare functie din lista listf
-- trebuie sa aplicam functia f).

compuneList :: (b -> c) -> [(a -> b)] -> [(a -> c)]
compuneList f listf = map (f .) listf

-- 2. Aplicarea functiilor pe un argument (aplicam argumentul a fiecarei
-- functii din lista listf)
-- aplicaList 100 [sqrt, (^2)] = [10, 10000]

aplicaList :: a -> [(a -> b)] -> [b]
aplicaList _ [] = []
aplicaList n (f:fs) = f n : aplicaList n fs

-- FUNCTIA FOLDR. PROPRIETATEA DE UNIVERSALITATE.

-- foldr :: (a -> b -> b) -> b -> [a]  -> b
-- Foldr ia o functie cu doua argumente, un elem initial, o lista de elemente
-- si intoarce un element final.
-- Practic, la fiecare pas se cumuleaza elementul initial conform functiei

-- foldr op unit [a1, a2, ... an] a1 `op` a2 `op` ... `op` an `op` unit
-- foldr (+) 0 [1, 2, 3, 4] = 1 + 2 + 3 + 4 + 0 = 10

-- Proprietatea de universalitate
-- Daca avem o functie pe care o putem scrie recursiv:
-- g [] = i
-- g (x:xs) = f x (g xs)
-- atunci functia g = foldr f i
-- Plecand de la varianta recursiva, pot face implementarea cu foldr.

suma :: [Int] -> Int
suma [] = 0
suma (x:xs) = x + suma xs        -- putem rescrie: (+) x (suma xs)

-- Avem acelasi pattern, unde f este (+), g este suma, iar 0 este elem initial.
-- Din proprietatea de universalitate, avem ca: suma = foldr (+) 0
```

```haskell
-- Daca nu ne prindeam de smecheria cu rescrierea:
-- Rescriem: g[] = 0
--           g (x:xs) = x + g xs
-- Din th. de universalitate avem ca g(x:xs) = f x (g xs).
-- Atunci: g[] = 0
--         f x (g xs) = x + g xs
-- Notam g xs = u si avem: g[] = 0
--                         f x u = x + u
-- Acum putem deduce:
--    (\x u -> x + u) care e echivalent cu (+)
--    0 element initial
-- Prin urmare:
-- suma = foldr f i
--        where
--             i = 0
--             f x u = u + x
-- Rescris:
-- suma = foldr (\x u -> x + u) 0

-- 3. Produs recursiv

produsRec :: [Integer] -> Integer
produsRec [] = 1
produsRec (x:xs) = x * produsRec xs

produsFold :: [Integer] -> Integer
produsFold lista = foldr (*) 1 lista

-- 4. And recursiv

andRec :: [Bool] -> Bool
andRec [] = True
andRec (x:xs) = x && andRec xs

andFold :: [Bool] -> Bool
andFold lista = foldl (&&) True lista

-- 5. Concatenare

-- "s" ++ "tr" = "str"
-- 's' : "tr" = "str"
-- 's' ++ 't' = eroare
-- 's' ++ 't' + 'r' + [] = "str"
-- [1, 2, 3] ++ [3, 4, 5] = [1, 2, 3, 3, 4, 5]
```

```haskell
concatRec :: [[a]] -> [a]
concatRec [] = []
concatRec (x:xs) = x ++ concatRec xs

concatFold :: [[a]] -> [a]
concatFold lista = foldr (++) [] lista

-- 6. Folosind doar recursia, scrieti o functie semn care ia ca parametru
-- o lista de intregi si intoarce un string care contine semnul numerelor
-- care apartin intervalului [-9, 9].
-- semn [-5, -10, -9, 9, 10, -3, 0] = "--+-0"

semn :: [Integer] -> String
semn [] = ""
semn (x:xs)
    | x <= -9 = semn xs
    | x < 0 = '-' : semn xs
    | x == 0 = "0" ++ semn xs
    | x <= 9 = '+' : semn xs
    | otherwise = semn xs

-- Aplicam algoritmul:
-- g [] = ""
-- g (x:xs) = if x < -9 then semn xs
--            else (if x < 0 then '-': semn xs
--                  else ...)
-- =>
-- g [] = ""
-- f x (g xs) = if x < -9 then ...
-- =>
-- g [] = ""
-- f x u = if x < -9 then ...
--     (notam (g xs) cu u)

unit :: String
unit = ""

f :: Integer -> String -> String
f = \x u -> if x < -9 then u
          else (if x < 0 then '-' : u
                else (if x == 0 then "0" ++ u
                      else (if x <= 9 then '+' : u
                            else u)))
```

```haskell
semnFoldr :: [Integer] -> String
semnFoldr = foldr f unit

semnFoldrExplicit :: [Integer] -> String
semnFoldrExplicit = foldr f i
                    where
                        i :: String
                        i = ""
                        f :: Integer -> String -> String
                        f = \x u -> if x < -9 then u
                                else (if x < 0 then '-' : u
                                        else (if x == 0 then "0" ++ u
                                            else (if x <= 9 then '+' : u
                                                    else u)))


-- Laboratorul 5

-- 1. Calculati suma pătratelor elementelor impare dintr-o listă dată ca
-- parametru.

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

-- 2. Scrieti o functie care verifică faptul că toate elementele dintr-o listă
-- sunt True, folosind foldr.

elemTrue :: [Bool] -> Bool
elemTrue = foldr (&&) True

-- 3. Scrieti o functie care verifică dacă toate elementele dintr-o listă de
-- numere întregi satisfac o proprietate dată ca parametru.

allVerifies :: (Int -> Bool) -> [Int] -> Bool
allVerifies f lista = foldr (&&) True (map (\h -> f h) lista)

-- 4. Scriet o functie care verifică dacă există elemente într-o listă de numere
-- întregi care satisfac o proprietate dată ca parametru.

anyVerifies :: (Int -> Bool) -> [Int] -> Bool
anyVerifies f lista = foldr (||) False (map (\h -> f h) lista)

-- 5. Redefiniti functiile map si filter folosind foldr. Le puteti numi mapFoldr
-- si filterFoldr.
```

```haskell
mapFoldr :: (a -> b) -> [a] -> [b]
mapFoldr f lista = foldr (\ h t -> f h : t) [] lista

filterFoldr :: (a -> Bool) -> [a] -> [a]
filterFoldr f = foldr(\ h t -> if f h then h : t else t) []

-- 6. Folosind functia foldl, definiti functia listToInt care transformă o lista
-- de cifre (un număr foarte mare stocat sub formă de listă) în numărul intreg
-- asociat. Se presupune ca lista de intrare este dată corect.

listToInt :: [Integer] -> Integer
listToInt  = foldl (\ x y -> x *10 + y) 0

-- 7.
-- (a) Scrieti o functie care elimină un caracter din sir de caractere.

rmChar :: Char -> String -> String
rmChar _ [] = []
rmChar ch (x:xs)
    | x == ch = rmChar ch xs
    | otherwise = x : rmChar ch xs

rmChar :: Char -> String -> String
rmChar chr = filter(/=chr)

-- (b) Scrieti o functie recursivă care elimină toate caracterele din al doilea
-- argument care se găsesc în primul argument, folosind rmChar.

rmCharsRec :: String -> String -> String
rmCharsRec [] str = str
rmCharsRec _ [] = []
rmCharsRec str (x:xs)
    | elem x str = rmCharsRec str xs
    | otherwise = x : rmCharsRec str xs

rmCharsRec :: String -> String -> String
rmCharsRec [] str = str
rmCharsRec (h:t) str = rmCharsRec t (rmChar h str)

-- (c) Scrieti o functie echivalentă cu cea de la (b) care foloseste foldr în
-- locul recursiei si rmChar.

rmCharsFold :: String -> String -> String
rmCharsFold str = foldr (\ h t -> if h `elem` str then t else h : t) []
```

```haskell
rmCharsFold2 :: String -> String -> String
rmCharsFold2 str1 str2 = foldr rmChar str2 str1

--                              rmChar sn str2
--                rmChar s(n-1) res1
--  rmChar s(n-2) res2
-- ..............................
-- Ajungem sa fie eliminate toate caracterele din str1 care sunt in str2

-- foldr (rmChar) str2 str1
--                     [s1, s2, ... sn]
-- Se scot sn, ... s2, s1 pe rand din str2 (asociativitate la dreapta):
-- s1 `rmChar` s2 `rmChar` ... `rmChar` sn `rmChar` str2

-- Sa ne reamintim! Principiile de currying si uncurrying
-- f x y z = x + y + z
-- Se poate rescrie ca:
-- f x y = \z -> x + y + z
-- f x = \y z -> x + y + z
-- f = \x y z -> x + y + z

-- Atentie! Am inversat str1 si str2 ca sa ne folosim de proprietatea de
-- universalitate.

-- g [] = i
-- g (x:xs) = f x (g x)
-- =>
-- g [] str = []
-- g (x:xs) str = if elem x str then g xs str else x : g xs str
-- =>
-- g [] = \str -> []
-- f x (g xs) str = if elem x str then g xs str else x : g xs str
--     (am inlocut g(x:xs) cu f x (g xs))
-- =>
-- g [] = \_ -> []
-- f x u str = if elem x str then u str else x : u str
--     (notam u = g xs)
-- =>
-- g = foldr f i

rmCharsAl :: String -> String -> String
rmCharsAl = foldr (\x u str -> if elem x str then u str else x : u str)(\_ -> [])

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

```haskell
-- Laboratorul 5 (Macovei)

import Numeric.Natural

-- 1. Definirea lui MAP cu foldr

-- Primul pas este sa avem functia intr-o forma recursiva, cat mai apropiata de
-- proprietatea de universalitate: g[] = i
--                                 g (x:xs) = op x (g xs)
-- map :: (a -> b) -> [a] -> [b]
-- map _ [] = []
-- map f (x:xs) = f x : map f xs
-- =>
-- Rescriu functia map, aducand lista pe prima pozitie (inversez parametrii).
-- g :: [a] -> (a -> b) -> [b]
-- g [] _ = []
-- g (x:xs) f = f x : g xs f
-- =>
-- Transformam cazul de oprire cu un lambda function pentru a ajunge mai aproape
-- de forma standard si a obtine i (elementul initial) de forma (a -> b) -> [b].
-- g :: [a] -> (a -> b) -> [b]
-- g [] = \_ -> []
-- g (x:xs) f = f x : g xs f
-- =>
-- Din proprietatea de universalitate, inlocuim g (x:xs) cu op x (g xs).
-- g :: [a] -> (a -> b) -> [b]
-- g [] = \_ -> []
-- op x (g xs) f = f x : g xs f
-- =>
-- Notez g xs := u si inlocuiesc.
-- g :: [a] -> (a -> b) -> [b]
-- g [] = \_ -> []
-- op x u f = f x : u f

-- Am obtinut elementul initial:
i :: (a -> b) -> [b]
i = \_ -> []

-- Am obtinut functia:
op :: a -> ((a -> b) -> [b]) -> (a -> b) -> [b]
op = \x u f -> f x : u f

-- Putem scrie functia MAP:
mapFold :: [a] -> (a -> b) -> [b]
mapFold = foldr op i
```

```haskell
-- ghci> (foldr op i) [1, 2, 3] (+2)
-- [3,4,5]

-- 2. Evaluarea lenesa: Haskell nu evalueaza ceea ce nu ii trebuie mai departe

logistic :: Num a => a -> a -> Natural -> a
logistic rate start = f
                      where
                        f 0 = start
                        f n = rate * f (n-1) * (1 - f (n-1))

logistic0 :: Fractional a => Natural -> a
logistic0 = logistic 3.741 0.00079

-- Cu cat argumentul pe care care il dam lui logistic0 este mai mare, cu atat
-- va creste timpul de asteptare al evaluarii

ex1 :: Natural
ex1 = 20

-- Daca vrem sa afisam lista, va calcula toate elementele
ex20 :: Fractional a => [a]
ex20 = [1, logistic0 ex1, 3]

-- Daca vrem sa afisam head-ul, nu va calcula logistic0 ex1 si va afisa doar
-- primul element.
ex21 :: Fractional a => a
ex21 = head ex20

-- Nu va evalua toata lista, va afisa doar elementul de pe pozitia a doua.
ex22 :: Fractional a => a
ex22 = ex20 !! 2

-- Nu va evalua toata lista, va lua elem de pe pozitia 2 si il va impacheta
-- intr-o alta lista.
ex23 :: Fractional a => [a]
ex23 = drop 2 ex20

-- Va evalua doar tail-ul, adica ultimele doua elemente.
ex24 :: Fractional a => [a]
ex24 = tail ex20

-- Pentru 5, va da direct True fara a calcula si a doua parte.
ex31 :: Natural -> Bool
ex31 x = x < 7 || logistic0 (ex1 + x) > 2
```

```haskell
-- Pentru 5, o sa dureze deoarece calculeaza primul caz, nu merge direct la al
-- doilea.
ex32 :: Natural -> Bool
ex32 x = logistic0 (ex1 + x) > 2 || x < 7

-- 3. Matrice ca lista de liste

matrice :: Num a => [[a]]
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

-- 3.1 Scrieti o functie care verifica daca o matrice este corecta (listele au
-- lungime egala). Luam prima si a doua linie si vedem daca sunt egale, apoi
-- mergem cu doua cate doua recursiv pentru a verifica in continuare.

corect :: [[a]] -> Bool
corect [] = True
corect [_] = True
corect (x:y:xs) = length  x == length y && corect(y:xs)

-- 3.2 Scrieti o functie care returneaza elementul de pe pozitia i si j.

el :: [[a]] -> Int -> Int -> a
el matrix line column = (matrix !! line) !! column

-- 3.3 Scrieti o functie care genereaza o lista de forma:
-- [(elem1, i_elem1, j_elem1), (elem2, i_elem1, j_elem2)...]

enumera :: [a] -> [(a, Int)]
enumera list = zip list [0..]

insereazaPozitie :: ([(a, Int)], Int) -> [(a, Int, Int)]
insereazaPozitie (lista, linie) = map (\(x, coloana) -> (x, linie, coloana))
lista

transforma :: [[a]] -> [(a, Int, Int)]
transforma matrix = concat(map insereazaPozitie (enumera (map enumera matrix)))

transforma' :: [[a]] -> [(a, Int, Int)]
transforma' = concat . map insereazaPozitie . enumera . map enumera

-- ghci> map enumera matrice
-- [[(1,0), (2,1), (3,2)], [(4,0), (5,1), (6,2)], [(7,0), (8,1), (9,2)]]
-- [[(element, coloana sa), ....], .....]
```

```haskell
-- ghci> enumera (map enumera matrice)
-- [([(1,0), (2,1), (3,2)], 0), ([(4,0), (5,1), (6,2)], 1), ([(7,0), (8,1),
-- (9,2)], 2)]
-- [[[(element, coloana sa), ....], linia elementelor], .....]

-- ghci> map insereazaPozitie (enumera (map enumera matrice))
--[[(1,0,0), (2,0,1), (3,0,2)], [(4,1,0), (5,1,1), (6,1,2)], [(7,2,0), (8,2,1),
-- (9,2,2)]]
-- inserez linia in tuplul fiecarui element

-- ghci> concat (map insereazaPozitie (enumera (map enumera matrice)))
-- [(1,0,0), (2,0,1), (3,0,2), (4,1,0), (5,1,1), (6,1,2), (7,2,0), (8,2,1),
-- (9,2,2)]
-- concatenez toate listele


-- Laboratorul 6

-- Exercitiul 1

-- Vom începe prin a scrie câteva functii definite folosind tipul de date Fruct:
data Fruct
  = Mar String Bool
  | Portocala String Int

-- O expresie de tipul Fruct este fie un Mar String Bool sau o Portocala String
-- Int. Vom folosi un String pentru a indica soiul de mere sau portocale, un Bool
-- pentru a indica dacă mărul are viermi si un Int pentru a exprima numărul de
-- felii dintr-o portocală.

-- De exemplu:

ionatanFaraVierme = Mar "Ionatan" False
goldenCuVierme = Mar "Golden Delicious" True
portocalaSicilia10 = Portocala "Sanguinello" 10
listaFructe = [Mar "Ionatan" False,
               Portocala "Sanguinello" 10,
               Portocala "Valencia" 22,
               Mar "Golden Delicious" True,
               Portocala "Sanguinello" 15,
               Portocala "Moro" 12,
               Portocala "Tarocco" 3,
               Portocala "Moro" 12,
               Portocala "Valencia" 2,
               Mar "Golden Delicious" False,
```

```
               Mar "Golden" False,
               Mar "Golden" True]

-- a) Scrieti o functie care indică dacă un fruct este o portocală de Sicilia sau
-- nu. Soiurile de portocale din Sicilia sunt Tarocco, Moro si Sanguinello.
-- test_ePortocalaDeSicilia1 = ePortocalaDeSicilia (Portocala "Moro" 12) == True
-- test_ePortocalaDeSicilia2 = ePortocalaDeSicilia (Mar "Ionatan" True) == False

ePortocalaDeSicilia :: Fruct -> Bool
ePortocalaDeSicilia (Mar soi viermi) = False
ePortocalaDeSicilia (Portocala soi felii) = soi `elem` ["Moro", "Tarocco",
"Sanguinello"]

ePortocalaDeSicilia2 :: Fruct -> Bool
ePortocalaDeSicilia2 (Mar _ _) = False
ePortocalaDeSicilia2 (Portocala s _)
    | s `elem` ["Tarocco","Moro","Sanguinello"] = True
    | otherwise = False

-- b) Scrieti o functie care calculează numărul total de felii ale portocalelor
-- de Sicilia dintr-o listă de fructe.
-- test_nrFeliiSicilia = nrFeliiSicilia listaFructe == 52

nrFeliiPortocala :: Fruct -> Int
nrFeliiPortocala(Mar soi viermi) = 0
nrFeliiPortocala(Portocala soi felii) = felii

nrFeliiSicilia :: [Fruct] -> Int
nrFeliiSicilia []  = 0
nrFeliiSicilia (h:t) = if ePortocalaDeSicilia h then nrFeliiPortocala h +
nrFeliiSicilia t
                       else nrFeliiSicilia t

-- O singura functie (pattern matching):

nrFeliiSicilia :: [Fruct] -> Int
nrFeliiSicilia [] = 0
nrFeliiSicilia ((Mar _ _) : fs) = nrFeliiSicilia fs
nrFeliiSicilia ((Portocala soi nr) : fs)
    | ePortocalaDeSicilia (Portocala soi nr) = nr + nrFeliiSicilia fs
    | otherwise = nrFeliiSicilia fs

-- c) Scrieti o functie care calcuelază numărul de mere care au viermi dintr-o
-- lista de fructe.
-- test_nrMereViermi = nrMereViermi listaFructe == 2
```

```haskell
areViermi :: Fruct -> Bool
areViermi(Mar soi viermi) = viermi
areViermi(Portocala soi felii) = False

nrMereViermi :: [Fruct] -> Int
nrMereViermi [] = 0
nrMereViermi (h:t) = if areViermi h then nrMereViermi t + 1
                     else nrMereViermi t

-- O singura functie (pattern matching):

nrMereViermi :: [Fruct] -> Int
nrMereViermi [] = 0
nrMereViermi ((Portocala _ _) : fs) = nrMereViermi fs
nrMereViermi ((Mar soi viermi) : fs)
    | viermi == True = 1 + nrMereViermi fs
    | otherwise = nrMereViermi fs

-- Exercitiul 2

type NumeA = String
type Rasa = String
data Animal = Pisica NumeA | Caine NumeA Rasa
    deriving Show

-- a) Scrieti o functie care întoarce "Meow!" pentru pisică si "Woof!" pentru
-- câine.

vorbeste :: Animal -> String
vorbeste (Pisica numePisica) = "Meow!"
vorbeste (Caine numeCaine rasaCaine) = "Woof!"

-- Vă reamintiti tipul de date predefinit Maybe:
-- data Maybe a = Nothing | Just a
-- b) Scrieti o functie care întoarce rasa unui câine dat ca parametru sau
-- Nothing dacă parametrul este o pisică

rasaCaine :: Animal -> Rasa
rasaCaine (Caine numeCaine rasaCaine) = rasaCaine

rasa :: Animal -> Maybe String
rasa (Pisica numePisica) = Nothing
rasa (Caine numeCaine rasaCaine) = Just rasaCaine
```

```haskell
-- Exercitiul 3

-- Se dau urmatoarele tipuri de date ce reprezintă matrici cu linii de lungimi
-- diferite:

data Linie = L [Int]
   deriving Show        -- ca sa putem afisa linie
data Matrice = M [Linie]
   deriving Show

-- a) Scrieti o functie care verifica daca suma elementelor de pe fiecare linie
-- este egala cu o valoare n. Rezolvati cerinta folosind foldr.
-- test_verif1 = verifica (M[L[1,2,3], L[4,5], L[2,3,6,8], L[8,5,3]]) 10 == False
-- test_verif2 = verifica (M[L[2,20,3], L[4,21], L[2,3,6,8,6], L[8,5,3,9]]) 25 ==
-- True

sumaLinii :: Matrice -> [Int]
sumaLinii (M []) = []
sumaLinii (M (L h:t)) = sum h : sumaLinii(M t)

verifica :: Matrice -> Int -> Bool
verifica (M mat) n = foldr (&&) True [if x == n then True else False | x <-
                                                                    sumaLinii(M mat)]

-- O singura functie (pattern matching):

verifica :: Matrice -> Int -> Bool
verifica (M []) _ = True
verifica (M (L x : xs)) n
    | sum x == n = verifica (M xs) n
    | otherwise = False

-- b) Scrieti o functie doarPozN care are ca parametru un element de tip Matrice
-- si un numar intreg n, si care verifica daca toate liniile de lungime n din
-- matrice au numai elemente strict pozitive.
-- testPoz1 = doarPozN (M [L[1,2,3], L[4,5], L[2,3,6,8], L[8,5,3]]) 3 == True
-- testPoz2 = doarPozN (M [L[1,2,-3], L[4,5], L[2,3,6,8], L[8,5,3]]) 3 == False

extragereLinii :: Matrice -> [[Int]]
extragereLinii (M []) = []
extragereLinii (M (L h:t)) = h : extragereLinii(M t)

doarPozN :: Matrice -> Int -> Bool
doarPozN (M mat) n = foldr (&&) True [if length(filter(>0) linie) == n then True
else False | linie <- extragereLinii(M mat), length linie == n]
```

```haskell
-- c) Definiti predicatul corect care verifică dacă toate liniile dintr-o matrice
-- au aceeasi lungime.
-- testcorect1 = corect (M[L[1,2,3], L[4,5], L[2,3,6,8], L[8,5,3]]) == False
-- testcorect2 = corect (M[L[1,2,3], L[4,5,8], L[3,6,8], L[8,5,3]]) == True

extragereDimensiune :: Matrice -> Int
extragereDimensiune (M []) = 0
extragereDimensiune (M (L h:t)) = length h

corect :: Matrice -> Bool
corect (M mat) = foldr (&&) True [if length linie == extragereDimensiune(M mat)
then True else False | linie <- extragereLinii (M mat)]


-- Laboratorul 6 (Macovei)

-- 1. Definirea lui MAP cu foldr

-- Primul pas este sa avem functia intr-o forma recursiva, cat mai apropiata de
-- proprietatea de universalitate: g[] = i
--                                 g (x:xs) = op x (g xs)
-- map :: (a -> b) -> [a] -> [b]
-- map _ [] = []
-- map f (x:xs) = f x : map f xs
-- =>
-- Rescriu functia map, aducand lista pe prima pozitie (inversez parametrii).
-- g :: [a] -> (a -> b) -> [b]
-- g [] _ = []
-- g (x:xs) f = f x : g xs f
-- =>
-- Transformam cazul de oprire cu un lambda function pentru a ajunge mai aproape
-- de forma standard si a obtine i (elementul initial) de forma (a -> b) -> [b].
-- g :: [a] -> (a -> b) -> [b]
-- g [] = \_ -> []
-- g (x:xs) f = f x : g xs f
-- =>
-- Din proprietatea de universalitate, inlocuim g (x:xs) cu op x (g xs).
-- g :: [a] -> (a -> b) -> [b]
-- g [] = \_ -> []
-- op x (g xs) f = f x : g xs f
-- =>
-- Notez g xs := u si inlocuiesc.
-- g :: [a] -> (a -> b) -> [b]
-- g [] = \_ -> []
-- op x u f = f x : u f
```

```haskell
-- Am obtinut elementul initial:
i :: (a -> b) -> [b]
i = \_ -> []

-- Am obtinut functia:
op :: a -> ((a -> b) -> [b]) -> (a -> b) -> [b]
op = \x u f -> f x : u f

mapFold :: [a] -> (a -> b) -> [b]
mapFold = foldr op i

-- 2. Functia ordonata rescrisa cu foldr.
-- Scrieti o functie care verifica daca o lista este ordonata dupa un anumit
-- criteriu.

ordonata :: [a] -> (a -> a -> Bool) -> Bool
ordonata [] _ = True
ordonata [x] _ = True
ordonata (h:t) rel = (and [rel h y | y <- t]) && ordonata t rel

ordonata' :: [a] -> (a -> a -> Bool) -> Bool
ordonata' [] _ = True
ordonata' [x] _ = True
ordonata' (x:y:xs) rel
    | rel x y = ordonata' (y:xs) rel
    | otherwise = False

-- e suficient sa se compare un x cu succesorul lui
-- list = [1, 2, 3, 4, 5, 6]
-- [(1, 2), (2, 3), (3, 4), (4, 5), (5, 6)] === [(x succ x)]
--                                          === zip list (tail list)
-- Cum?
-- zip[1, 2, 3, 4, 5, 6][2, 3, 4, 5, 6] => (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)
--        length = 6       length = 5
-- Trebuie sa verific de fapt ca orice tuplu din lista zip list (tail list)
-- respecta relatia

ordonataT :: [(a, a)] -> (a -> a -> Bool) -> Bool
ordonataT [] _ = True
ordonataT ((x, y) : xs) rel
    | rel x y  = ordonataT xs rel
    | otherwise = False

-- g :: [(a, a)] -> (a -> a -> Bool) -> Bool
-- g [] = \_ -> True
```

```haskell
-- g ((x, y) : xs) rel
--     | rel x y  = g xs rel
--     | otherwise = False
-- =>
-- g :: [(a, a)] -> (a -> a -> Bool) -> Bool
-- g [] = \_ -> True
-- g ((x, y) : xs) rel =
--     if rel x y then g xs rel
--     else False
-- =>
-- g :: [(a, a)] -> (a -> a -> Bool) -> Bool
-- g [] = \_ -> True
-- g (x : xs) rel =
--     if rel (fst x) (snd x) then g xs rel
--     else False
-- =>
-- Inlocuim g (x:xs) cu f x (g xs) rel = ...
-- =>
-- Notam u := g xs
-- =>
-- Solutia

iord ::  (a -> a -> Bool) -> Bool
iord = \_ -> True

f :: (a, a) -> ((a -> a -> Bool) -> Bool) -> ((a -> a -> Bool) -> Bool)
f = \x u rel ->  if rel (fst x) (snd x) then u rel else False

ordonataTF :: [a] -> (a -> a -> Bool) -> Bool
ordonataTF list = foldr f iord (zip list (tail list))

-- 3. TIPURI DE DATE

-- Stim ca o lista contine doar elemente de acelasi tip. Cum facem daca vrem sa
avem o lista
-- care contine si char-uri si int-uri?

data CharInt = I Int | C Char
-- atentie, constructorul de date si cel de tip trebuie scris cu litera mare

list :: [CharInt]
list = [C 'c', I 4, I 6, I 89, C 'y']
```

```haskell
-- Laboratorul 7

-- Se dau următoarele tipuri de date reprezentând expresii si arbori de expresii:

data Expr = Const Int -- integer constant
          | Expr :+: Expr -- addition
          | Expr :*: Expr -- multiplication
    deriving Eq

data Operation = Add | Mult deriving (Eq, Show)

data Tree = Lf Int -- leaf
          | Node Operation Tree Tree -- branch
    deriving (Eq, Show)

-- 1.1. Să se instantieze clasa Show pentru tipul de date Expr, astfel încât să
-- se afiseze mai simplu expresiile.

exp1 = ((Const 2 :*: Const 3) :+: (Const 0 :*: Const 5))
exp2 = (Const 2 :*: (Const 3 :+: Const 4))
exp3 = (Const 4 :+: (Const 3 :*: Const 3))
exp4 = (((Const 1 :*: Const 2) :*: (Const 3 :+: Const 1)) :*: Const 2)

instance Show Expr where
    show (Const x) =  show x
    show (a :+: b) =  "(" ++ show a ++ "+" ++ show b ++ ")"
    show (a :*: b) =   "(" ++ show a ++ "*" ++ show b ++ ")"

-- 1.2. Să se scrie o functie evalExp :: Expr -> Int care evaluează o expresie
-- determinând valoarea acesteia.
-- test11 = evalExp exp1 == 6
-- test12 = evalExp exp2 == 14
-- test13 = evalExp exp3 == 13
-- test14 = evalExp exp4 == 16

evalExp :: Expr -> Int
evalExp (Const a) = a
evalExp (a :+: b) = evalExp a + evalExp b
evalExp (a :*: b) = evalExp a * evalExp b

-- 1.3. Să se scrie o functie evalArb :: Tree -> Int care evaluează o expresie
-- modelată sub formă de arbore, determinând valoarea acesteia.
arb1 = Node Add (Node Mult (Lf 2) (Lf 3)) (Node Mult (Lf 0)(Lf 5))
arb2 = Node Mult (Lf 2) (Node Add (Lf 3)(Lf 4))
arb3 = Node Add (Lf 4) (Node Mult (Lf 3)(Lf 3))
```

```haskell
arb4 = Node Mult (Node Mult (Node Mult (Lf 1) (Lf 2)) (Node Add (Lf 3)(Lf 1)))
       (Lf 2)

-- test21 = evalArb arb1 == 6
-- test22 = evalArb arb2 == 14
-- test23 = evalArb arb3 == 13
-- test24 = evalArb arb4 == 16

evalArb :: Tree -> Int
evalArb (Lf f) = f
evalArb (Node Add a b) = evalArb a + evalArb b
evalArb (Node Mult a b) = evalArb a * evalArb b

-- 1.4. Să se scrie o functie expToArb :: Expr -> Tree care transformă o expresie
-- în arborele corespunzător.

expToArb :: Expr -> Tree
expToArb (Const f) = Lf f
expToArb (a :+: b) = Node Add (expToArb a) (expToArb b)
expToArb (a :*: b) = Node Mult (expToArb a) (expToArb b)

-- In acest exercitiu vom exersa manipularea listelor si tipurilor de date prin
-- implementarea catorva colectii de tip tabela asociativa cheie-valoare.
-- Aceste colectii vor trebui sa aiba urmatoarele facilitati
-- • crearea unei colectii vide
-- • crearea unei colectii cu un element
-- • adaugarea/actualizarea unui element intr-o colectie
-- • cautarea unui element intr-o colectie
-- • stergerea (marcarea ca sters a) unui element dintr-o colectie
-- • obtinerea listei cheilor
-- • obtinerea listei valorilor
-- • obtinerea listei elementelor

class Collection c where
    empty :: c key value
    singleton :: key -> value -> c key value
    insert :: Ord key => key -> value -> c key value -> c key value
    clookup :: Ord key => key -> c key value -> Maybe value
    delete :: Ord key => key -> c key value -> c key value
    keys :: c key value -> [key]
    values :: c key value -> [value]
    toList :: c key value -> [(key, value)]
    fromList :: Ord key => [(key,value)] -> c key value

-- 2.1. Adaugati definitii implicite (in functie de functiile celelalte) pentru:
```

```haskell
-- a. keys
    keys c = map fst (toList c)

-- b. values
    values c = map snd (toList c)

-- c. fromList
    fromList [] = empty
    fromList ((key, value) : xs) = insert key value (fromList xs)

-- 2.2. Fie tipul listelor de perechi de forma cheie-valoare:

newtype PairList k v
    = PairList {getPairList :: [(k, v)]}

-- Faceti PairList instanta a clasei Collection.

instance Collection PairList where

    empty :: PairList key value
    empty = PairList []

    singleton :: key -> value -> PairList key value
    singleton key value = PairList[(key, value)]

    insert :: Ord key => key -> value -> PairList key value -> PairList key value
    insert key value (PairList list) = if key `elem` keys (PairList list)
                                       then PairList list
                                       else PairList ((key, value) : list )

    delete :: Ord key => key -> PairList key value -> PairList key value
    delete key (PairList list) = PairList [(good_key, good_value) |
                                (good_key, good_value) <- list, good_key /= key]

    keys :: PairList key value -> [key]
    keys (PairList list) = map fst list

    values :: PairList key value -> [value]
    values (PairList list) = map snd list

    fromList :: Ord key => [(key, value)] -> PairList key value
    fromList [] = empty
    fromList ((key, value) : xs) = insert key value (fromList xs)
```

```haskell
    toList :: PairList key value -> [(key, value)]
    toList = getPairList

    clookup :: Ord key => key -> PairList key value -> Maybe value
    clookup key (PairList list) = lookup key list


-- 2.3. Fie tipul arborilor binari de cautare (ne-echilibrati):

data SearchTree key value
    = Empty
    | BNode
        (SearchTree key value) -- elemente cu cheia mai mica
        key -- cheia elementului
        (Maybe value) -- valoarea elementului
        (SearchTree key value) -- elemente cu cheia mai mare

-- Observati ca tipul valorilor este Maybe value. Acest lucru se face pentru a
-- reduce timpul operatiei de stergere prin simpla marcare a unui nod ca fiind
-- sters. Un nod sters va avea valoarea Nothing.

-- Faceti SearchTree instanta a clasei Collection.

instance Collection SearchTree where

    empty :: SearchTree key value
    empty = Empty

    singleton :: key -> value -> SearchTree key value
    singleton key value = BNode Empty key (Just value) Empty

    insert :: Ord key => key -> value -> SearchTree key value -> SearchTree key
                                                                 value
    insert key value Empty = singleton key value
    insert key value (BNode littleTree current_key current_value biggerTree)
        | key > current_key =
         BNode littleTree current_key current_value (insert key value biggerTree)
        | key < current_key =
         BNode (insert key value littleTree) current_key current_value biggerTree
        | otherwise =
         BNode littleTree current_key current_value biggerTree

    clookup :: Ord key => key -> SearchTree key value -> Maybe value
    clookup key Empty = Nothing
    clookup key (BNode littleTree current_key current_value biggerTree)
```

```
        | key == current_key = current_value
        | key > current_key = clookup key littleTree
        | otherwise = clookup key biggerTree

    toList :: SearchTree key value -> [(key, value)]
    toList Empty = []
    toList (BNode littleTree key Nothing biggerTree) =
                                        (toList littleTree) ++ (toList biggerTree)
    toList (BNode littleTree key (Just value) biggerTree) =
                    (toList littleTree) ++ [(key, value)] ++ (toList biggerTree)

    delete :: Ord key => key -> SearchTree key value -> SearchTree key value
    delete key (BNode littleTree current_key current_value biggerTree)
        | key == current_key = BNode littleTree current_key Nothing biggerTree
        | key > current_key = delete key biggerTree
        | otherwise = delete key littleTree

    keys :: SearchTree key value -> [key]
    keys (BNode littleTree key (Just value) biggerTree) =
             [fst l | l <- toList (BNode littleTree key (Just value) biggerTree)]

    values :: SearchTree key value -> [value]
    values (BNode littleTree key (Just value) biggerTree) =
             [snd l | l <- toList (BNode littleTree key (Just value) biggerTree)]

    fromList :: Ord key => [(key, value)] -> SearchTree key value
    fromList [] = empty
    fromList ((current_key, current_value) : xs) =
                                    insert current_key current_value (fromList xs)


-- Laboratorul 7 (Macovei)

import Data.Char

-- 1. Functia de tip lookup (actioneaza precum dictionarele din Python)
-- Functia afiseaza stringul (valoarea) unei chei.

myLookUp :: Int -> [(Int, String)] -> Maybe String
myLookUp _ [] = Nothing
myLookUp element (x:xs)
    | element == fst x = Just (snd x)
    | otherwise        = myLookUp element xs

myLookUpFoldr :: Foldable t => t (Int, String) -> Int -> Maybe String
```

```haskell
myLookUpFoldr list = foldr f i list
    where
        i :: Int -> Maybe String
        i = \_ -> Nothing
        f :: (Int, String) -> (Int -> Maybe String) -> Int -> Maybe String
        f = \x u element -> if element == fst x then Just (snd x) else u element

myLookUpFold :: Int -> [(Int, String)] -> Maybe String
myLookUpFold element = foldr (\x found -> if fst x == element then Just (snd x)
else found) Nothing

-- 2. Sa se scrie o functie myLookUp' cu aceeasi signatura ca myLookUp care,
atunci cand gaseste valoarea
-- capitalizeaza prima litera. (i.e. transforma prima litera in majuscula).

myLookUp' :: Int -> [(Int, String)] -> Maybe String
myLookUp' _ [] = Nothing
myLookUp' element (x:xs)
    | element == fst x = Just (toUpper(head(snd x)) : tail(snd x))
    | otherwise = myLookUp' element xs

myLookUp1' :: Int-> [(Int, String)] -> Maybe String
myLookUp1' _ [] = Nothing
myLookUp1' element (x:xs)
    | element == fst x = Just (toUpper x1 : x2)
    | otherwise = myLookUp1' element xs
    where x1 : x2 = snd x
```